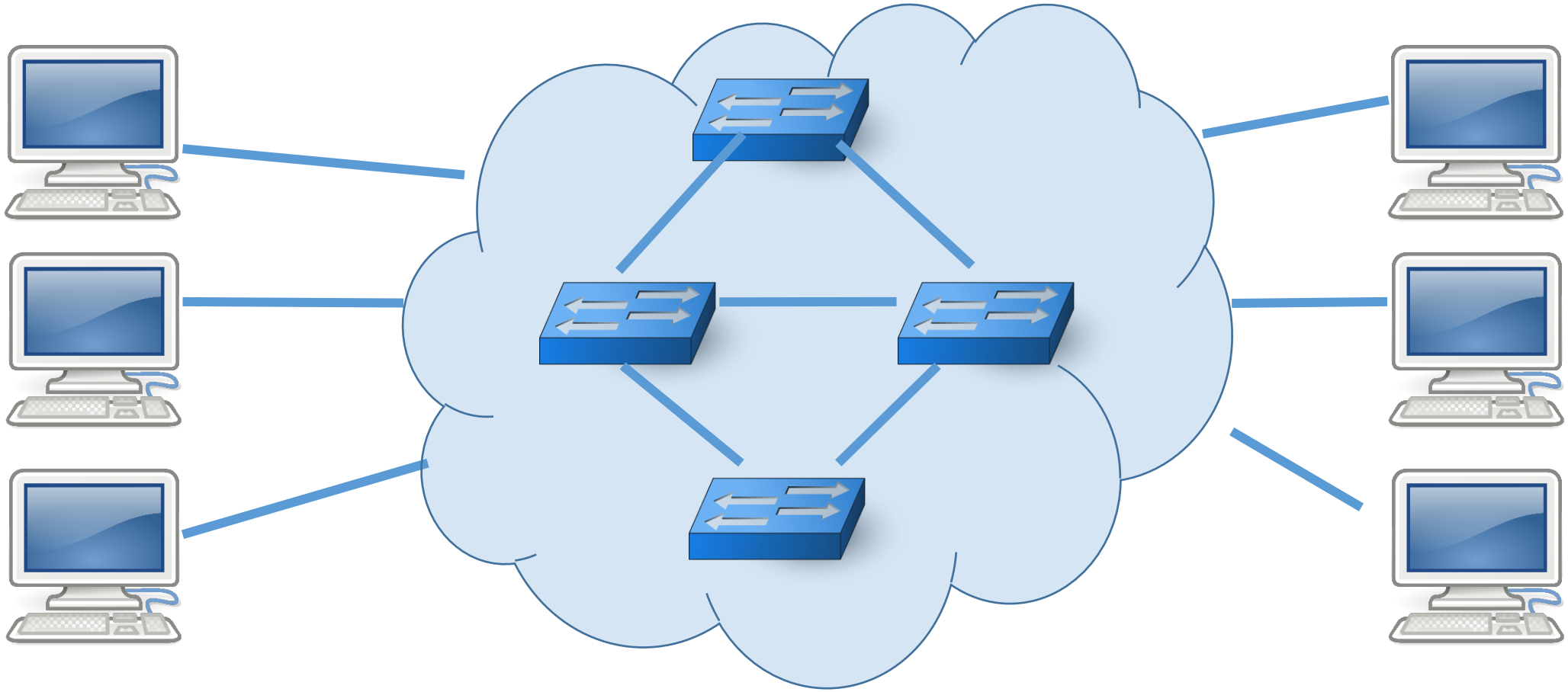
Programming Line-Rate Switches

Anirudh Sivaraman

Joint work with

- MIT: Suvinay Subramanian, Hari Balakrishnan, Mohammad Alizadeh
- Barefoot Networks: Changhoon Kim, Anurag Agrawal, Steve Licking, Mihai Budiu
- Cisco Systems: Shang-Tse Chuang, Sharad Chole, Tom Edsall
- Microsoft Research: George Varghese
- Stanford University: Sachin Katti, Nick McKeown
- University of Washington: Alvin Cheung

Traditional networking



Fixed (simple) switches and programmable (smart) end points

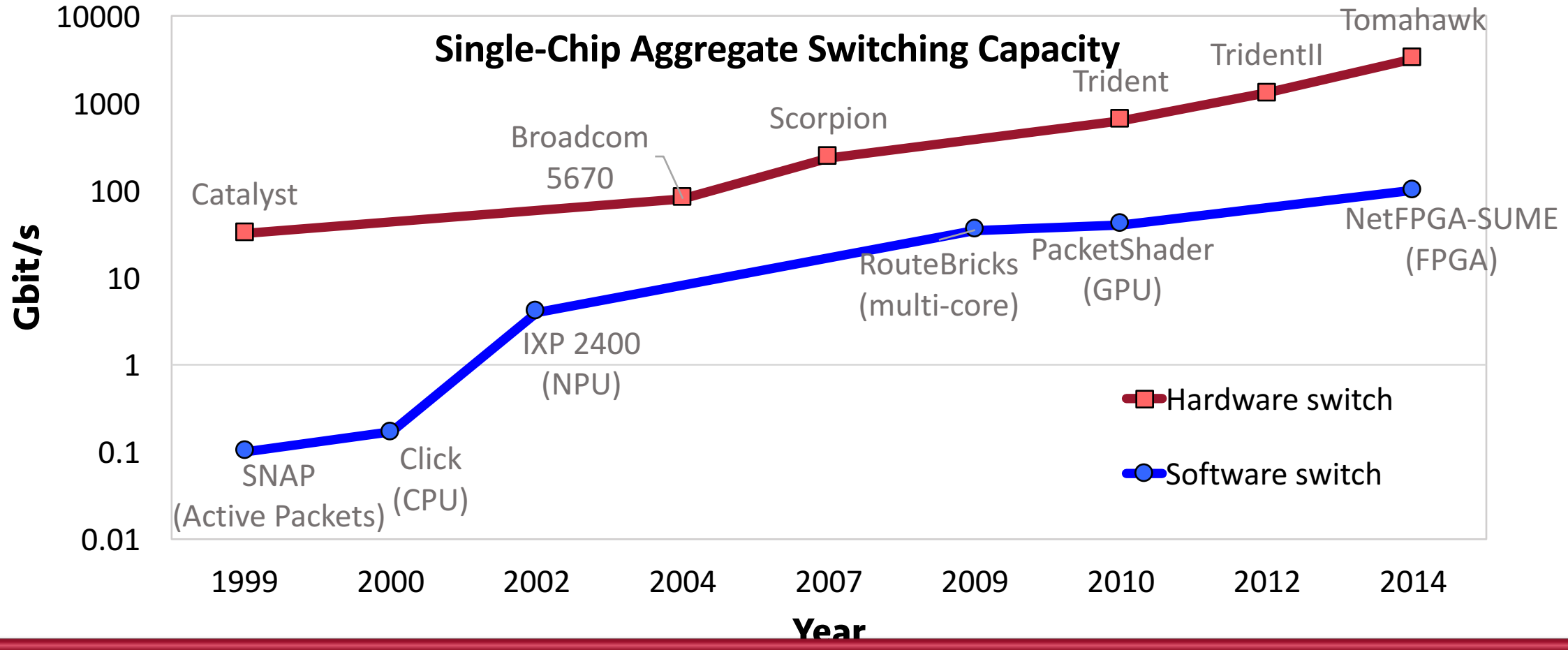
This is showing signs of age …

- Switch features tied to ASIC design cycles (2-3 years)
  - Long lag time for new protocol formats (IPv6, VXLAN)
- Operators (esp. in datacenters) need more control over switches
  - Access control, load balancing, bandwidth sharing, measurement
- Many switch algorithms never make it to production

The quest for programmable switches

- Early switches built out of minicomputers, which were sufficient
  - IMPs (1969): Honeywell DDP-516
  - Fuzzball (1971): DEC LSI-11
  - Stanford multiprotocol switch (1981): DEC PDP 11
  - Proteon / MIT C gateway (1980s): DEC MicroVAX II

The quest for programmable switches



Software switches (CPUs, NPUs, GPUs, FPGAs) are 10—100x slower

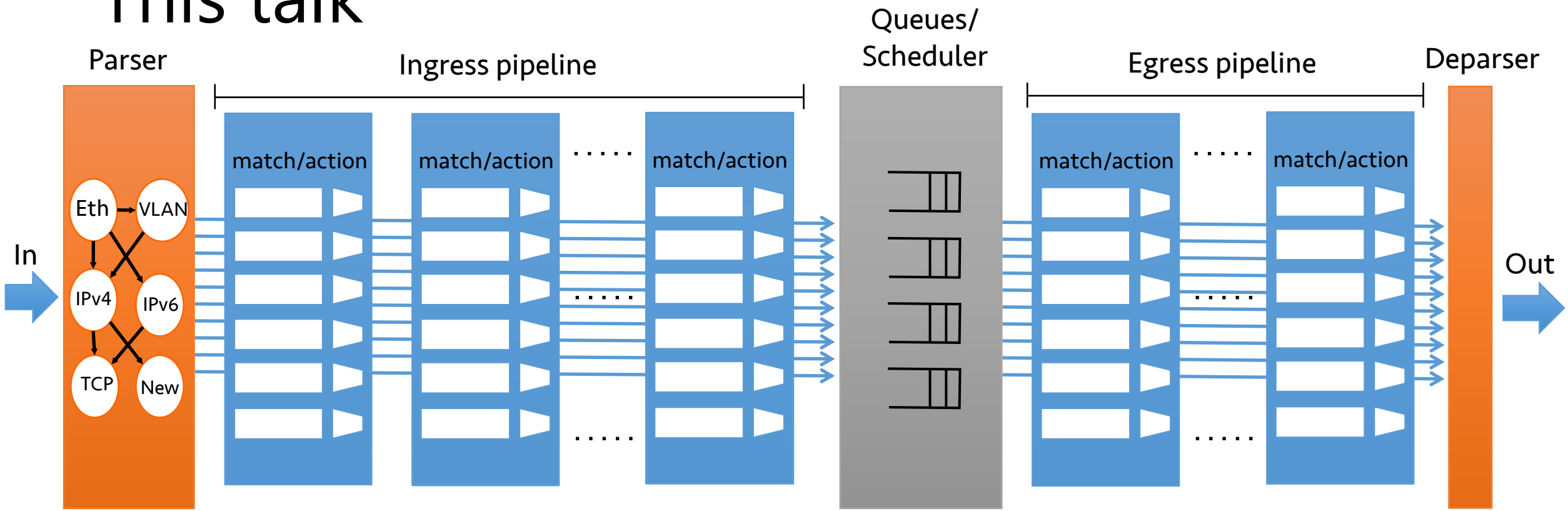
The vision: programmability at line rate

- Performance of fastest, fixed-function switches (> 1 Tbit/s)

- More programmable than fixed-function switches
  - Much more than OpenFlow/SDN, which only programs routing/control plane.
  - …, but less than software switches

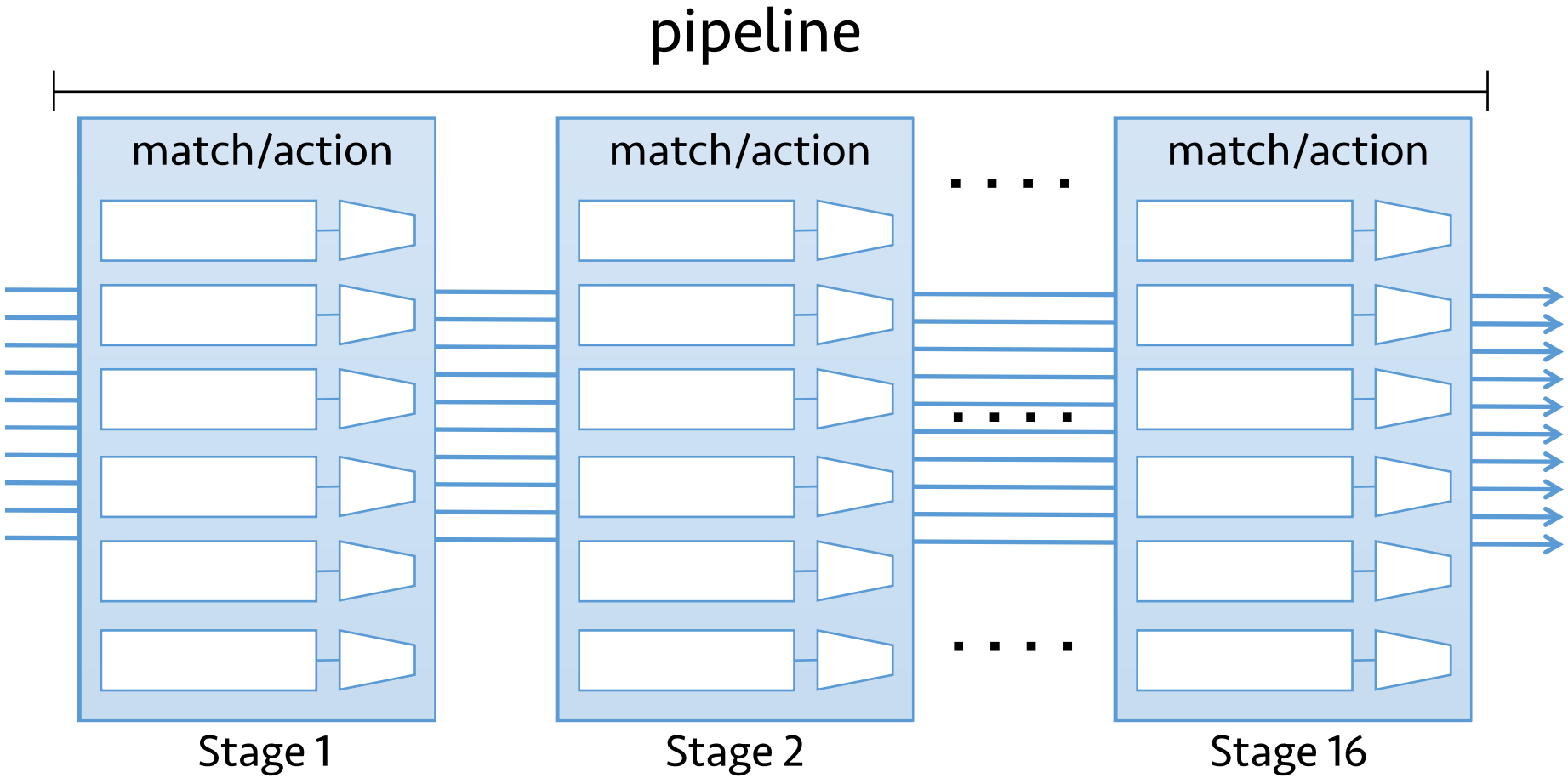
- Such programmable chips are emerging: Tofino, FlexPipe, Xpliant
  - As are languages such as P4 to program them

This talk

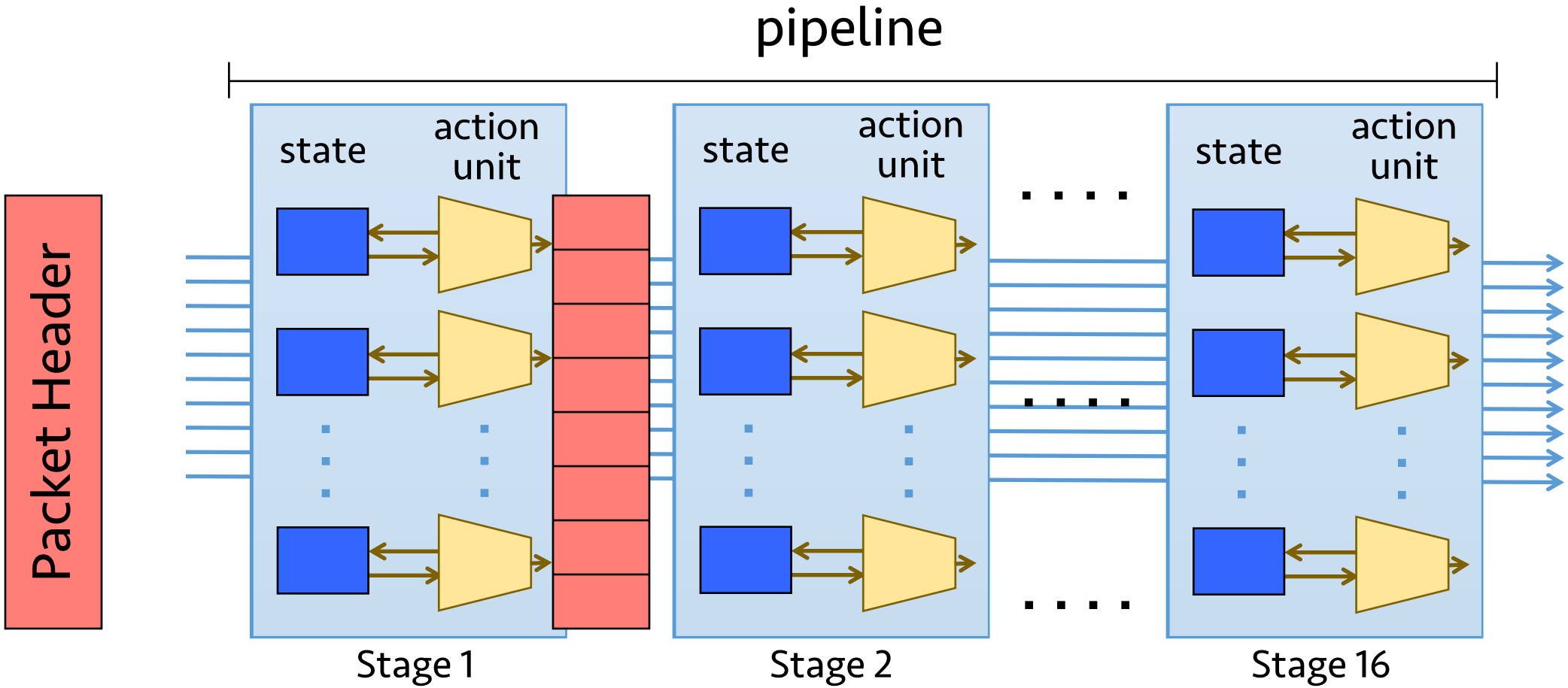


- The machine model: Formalizes the computational capabilities of line-rate switches
- Packet transactions: High-level programming for the switch pipeline
- Push-In First-Out Queues: Programming the scheduler

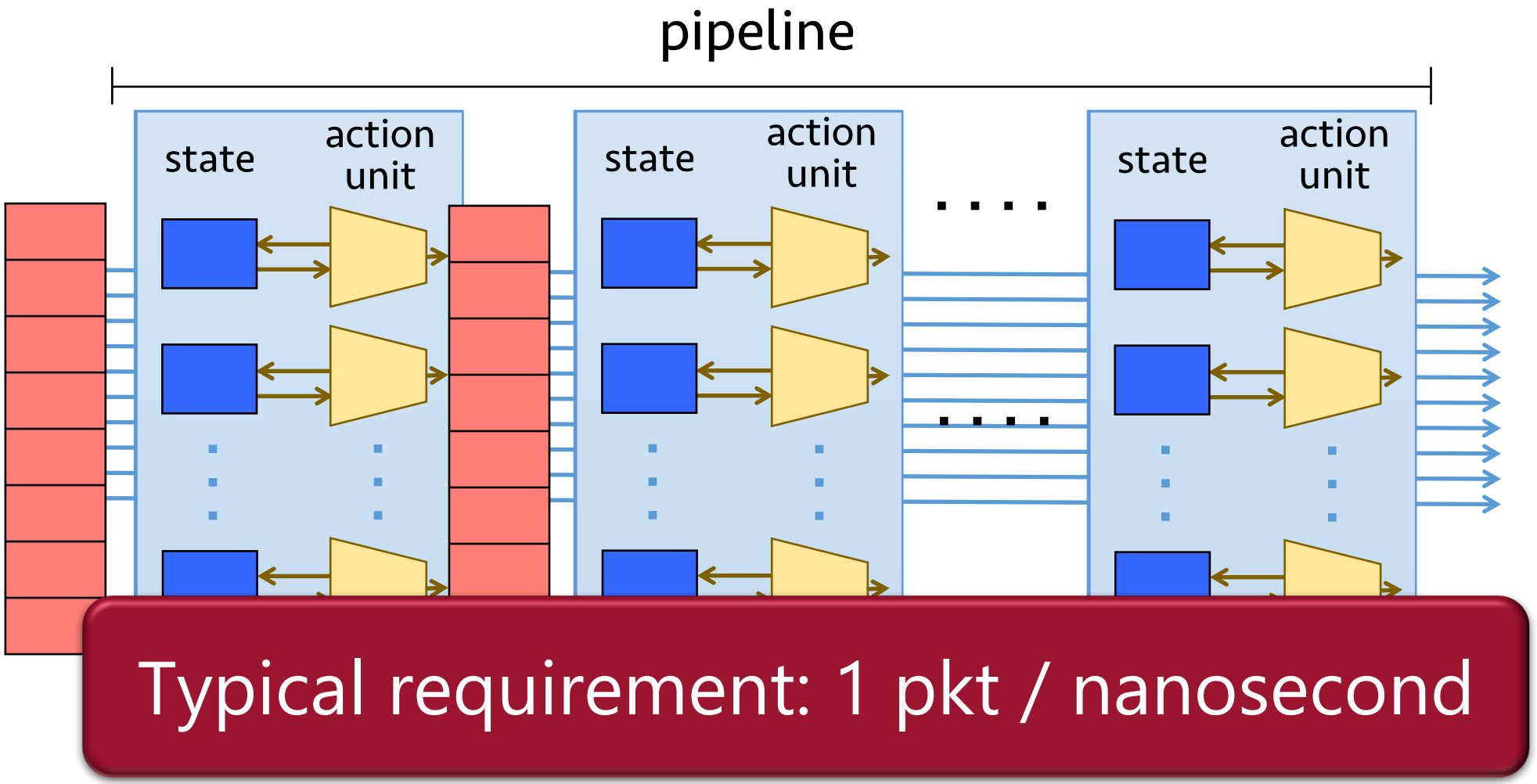
A machine model for line-rate switches



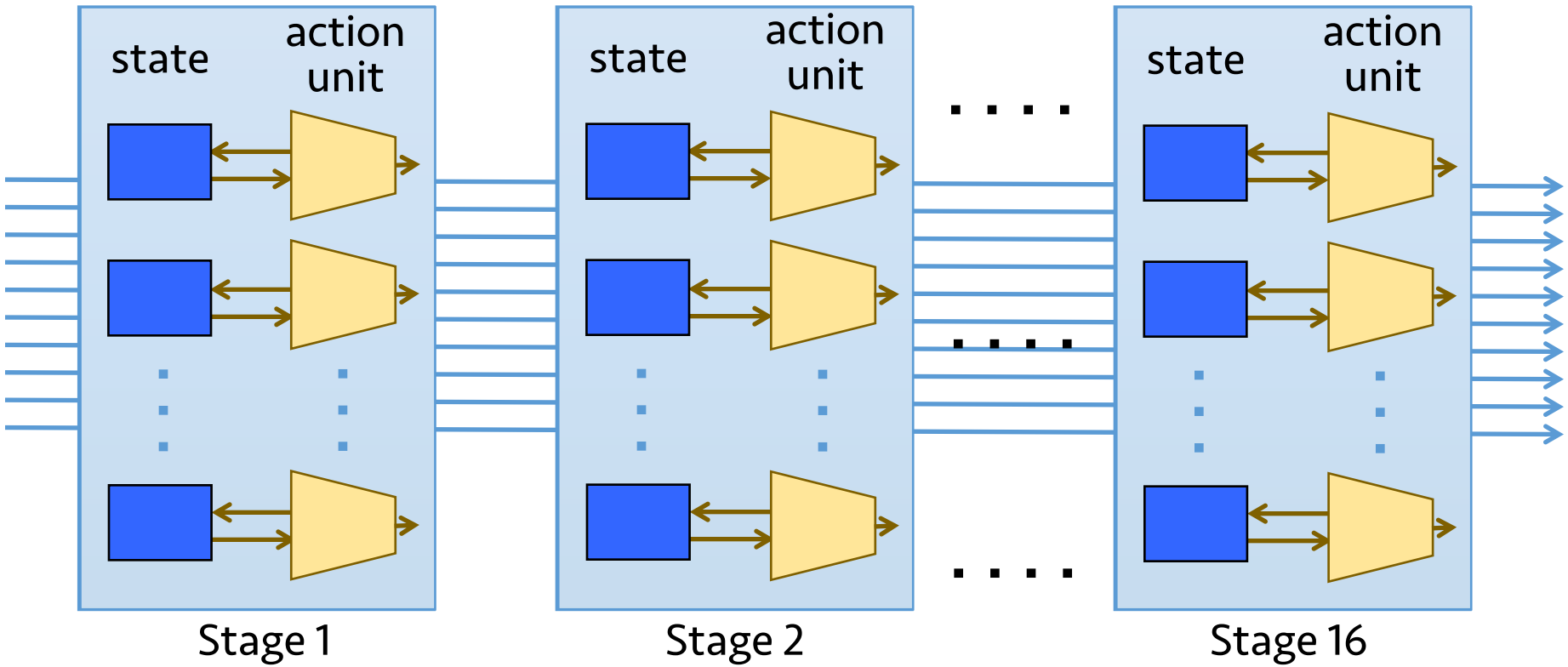
A machine model for line-rate switches



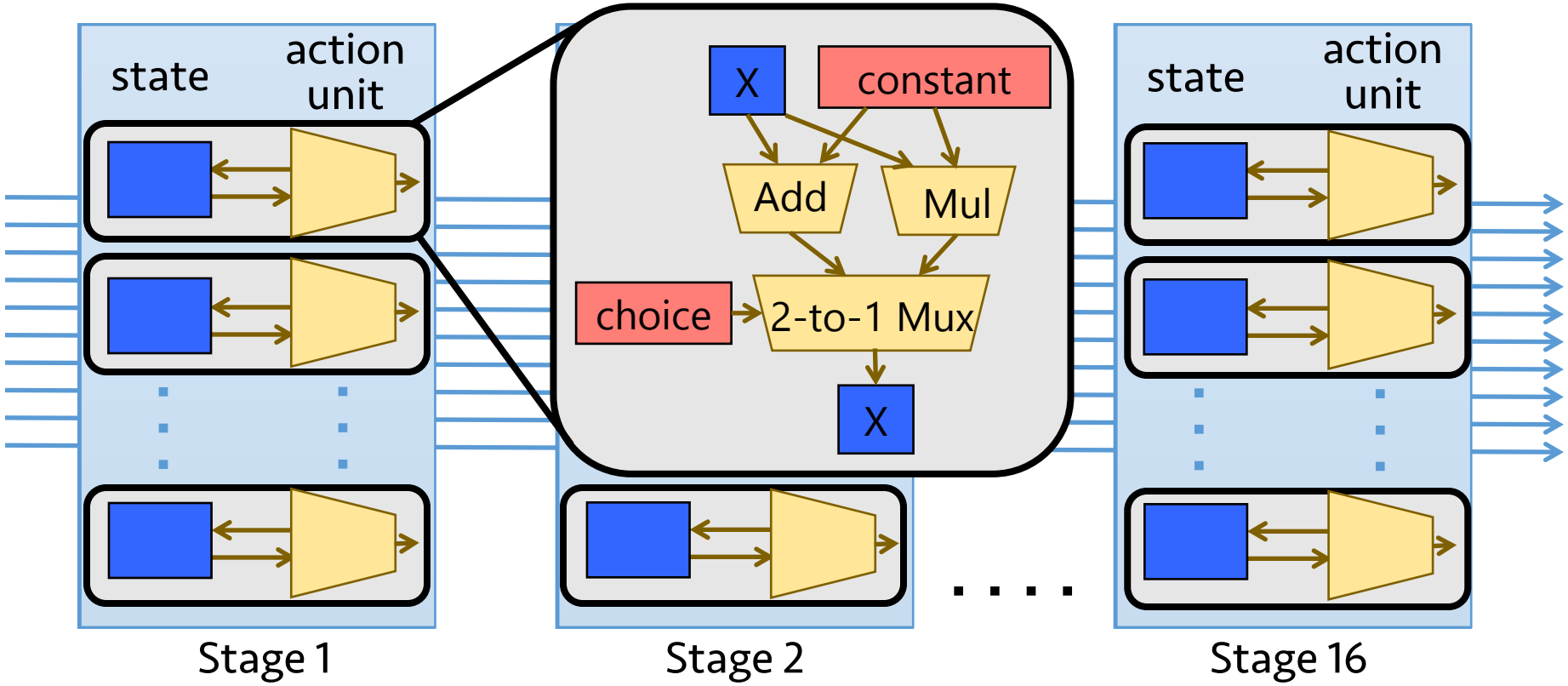
A machine model for line-rate switches



A machine model for line-rate switches



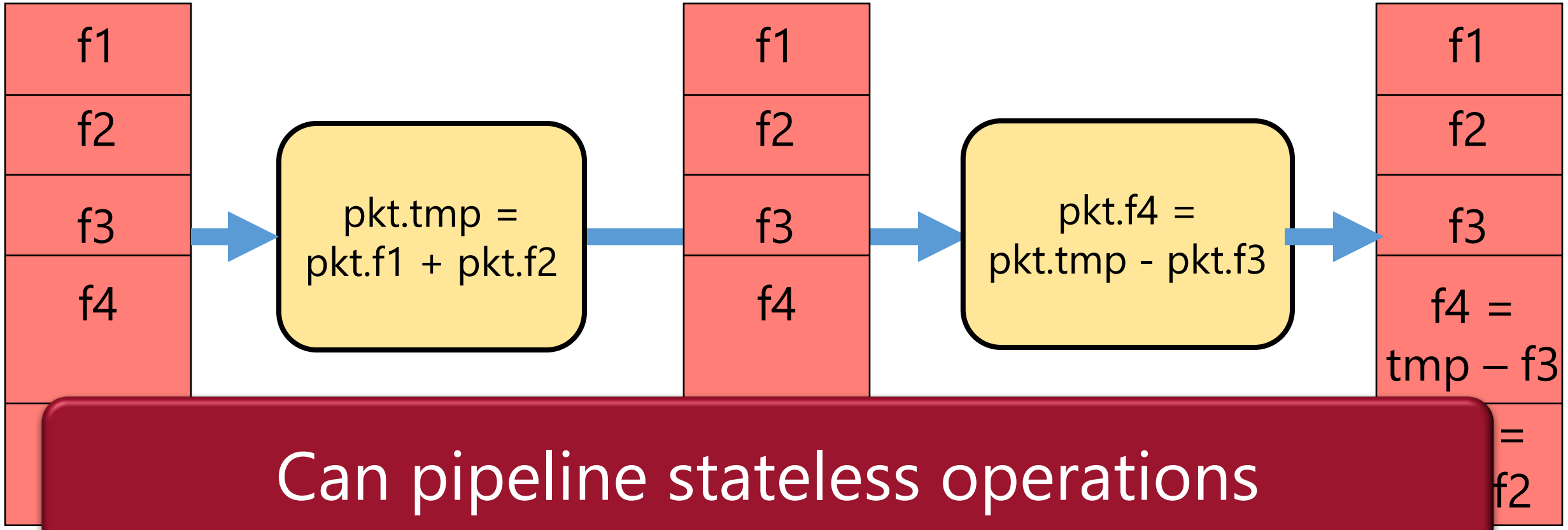
A machine model for line-rate switches



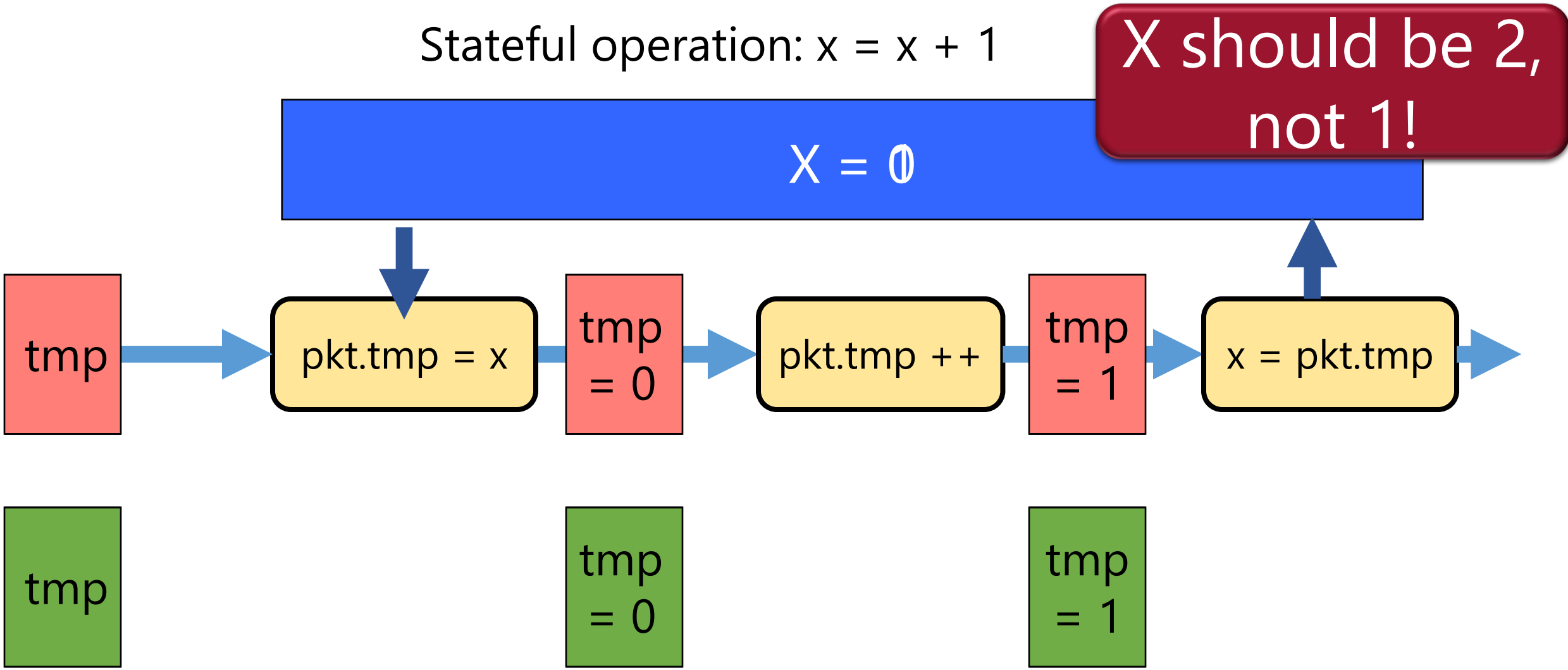
A switch's atoms constitute its instruction set

Stateless vs. stateful operations

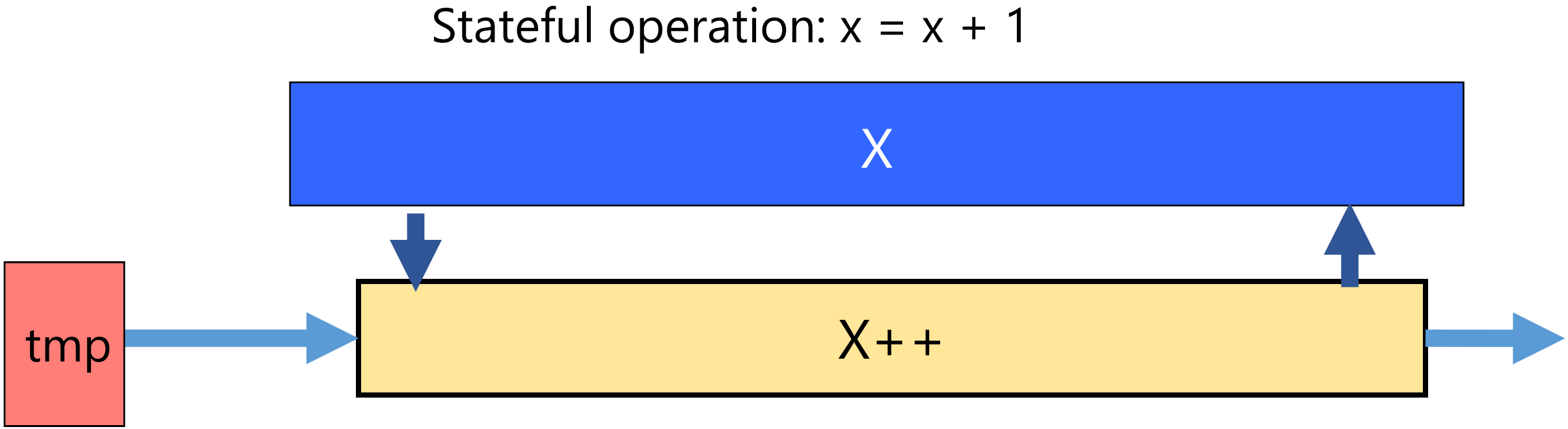
Stateless operation: pkt.f4 = pkt.f1 + pkt.f2 – pkt.f3



Stateless vs. stateful operations

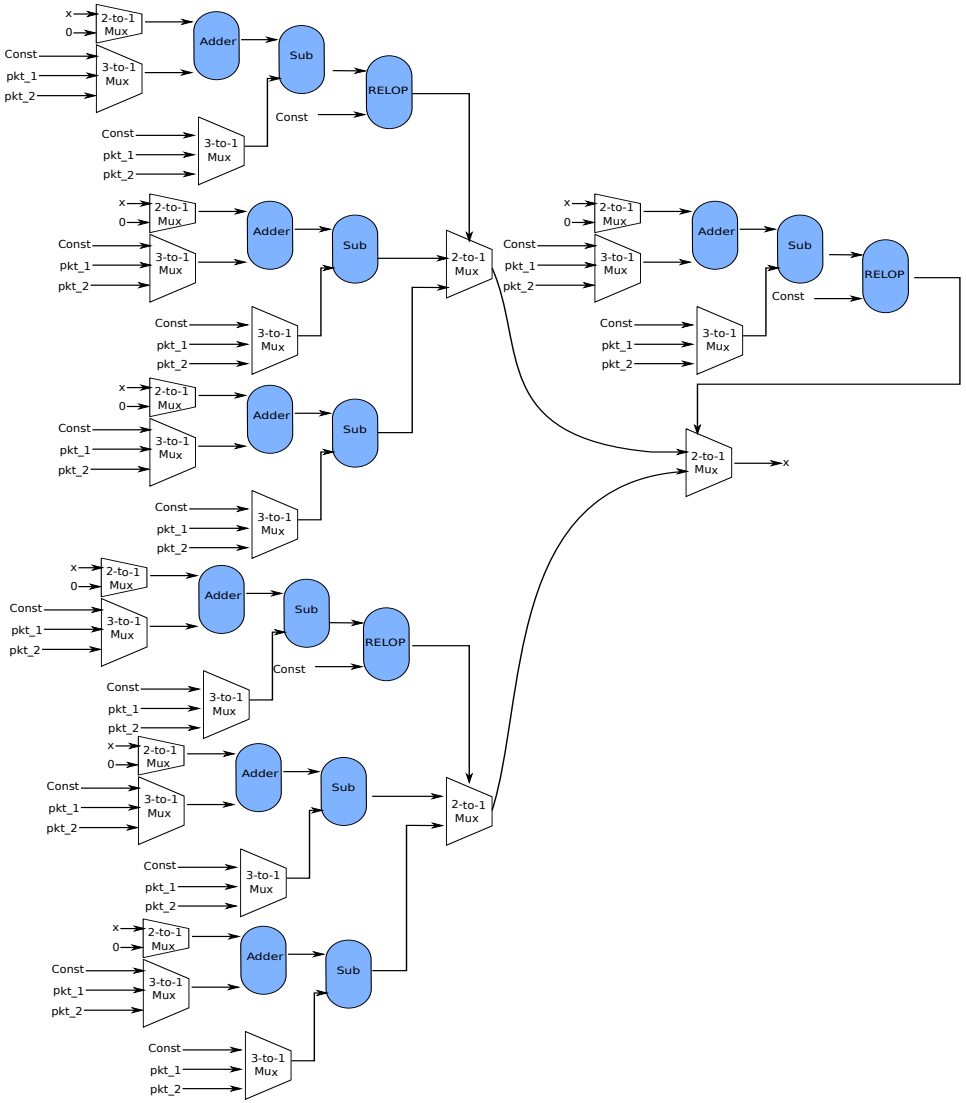


Stateless vs. stateful operations



Cannot pipeline, need atomic operation in h/w

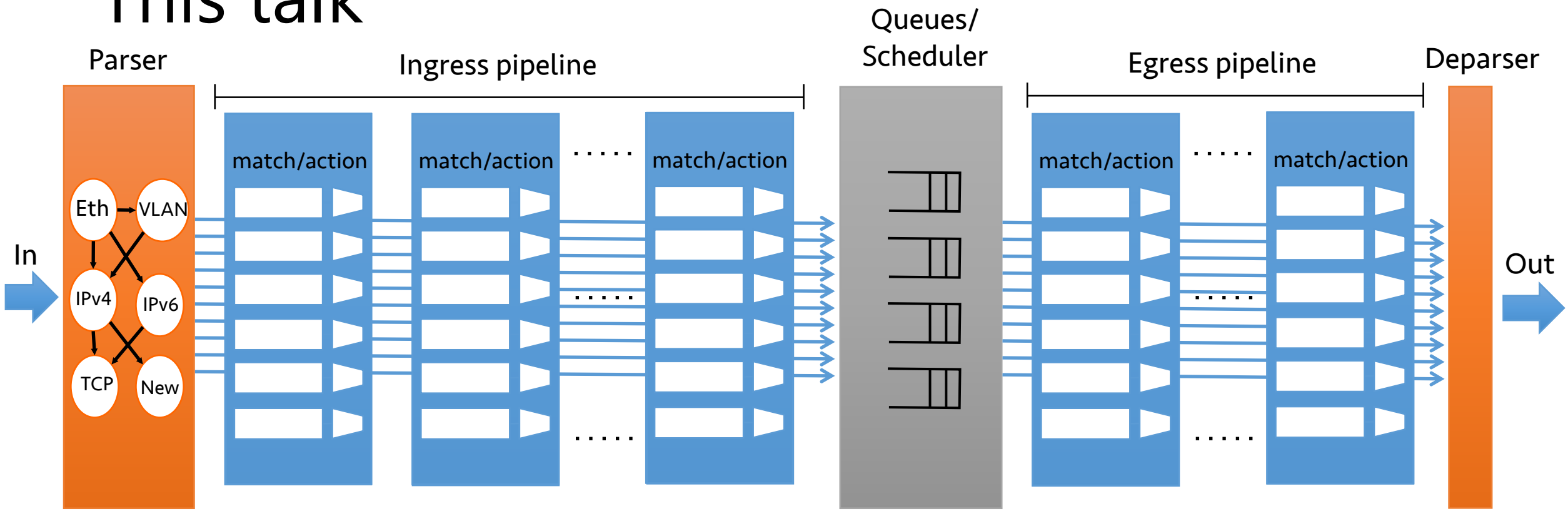
Stateful atoms can be fairly involved



Update state in one of four ways based on four predicates.

Each predicate can itself depend on the state.

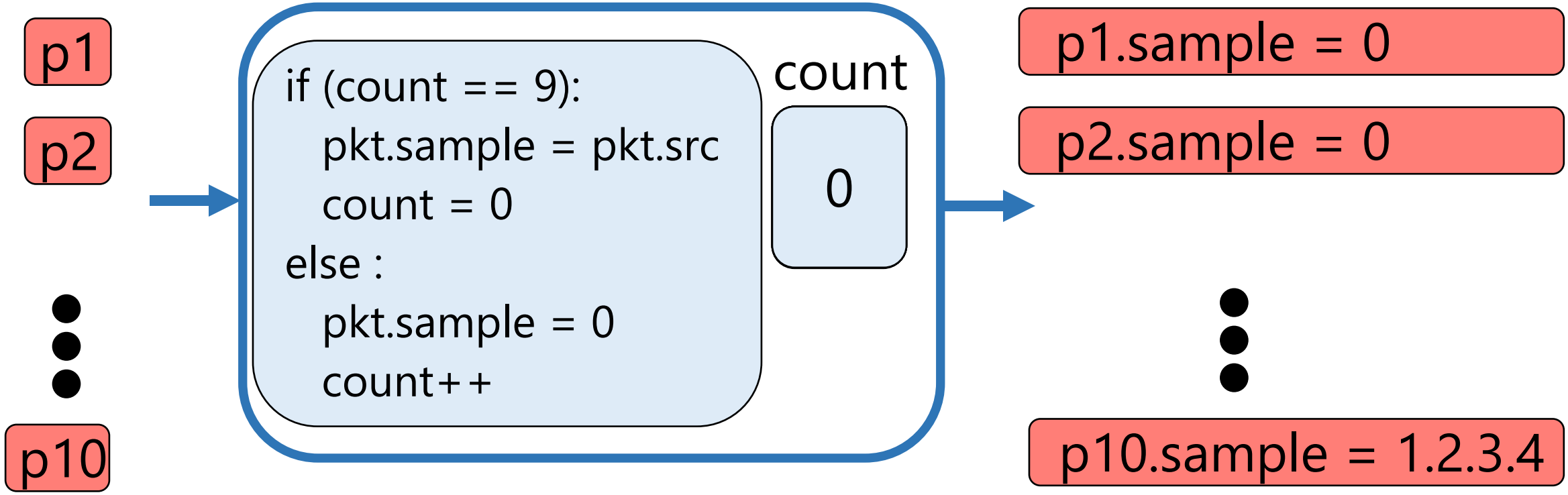
This talk



- The machine model: Formalizes the computational capabilities of line-rate switches
- Packet transactions: High-level programming for the switch pipeline
- Push-In First-Out Queues: Programming the scheduler

Packet transactions

- Packet transaction: block of imperative code
- Transaction runs to completion, one packet at a time, serially



Packet transactions are expressive

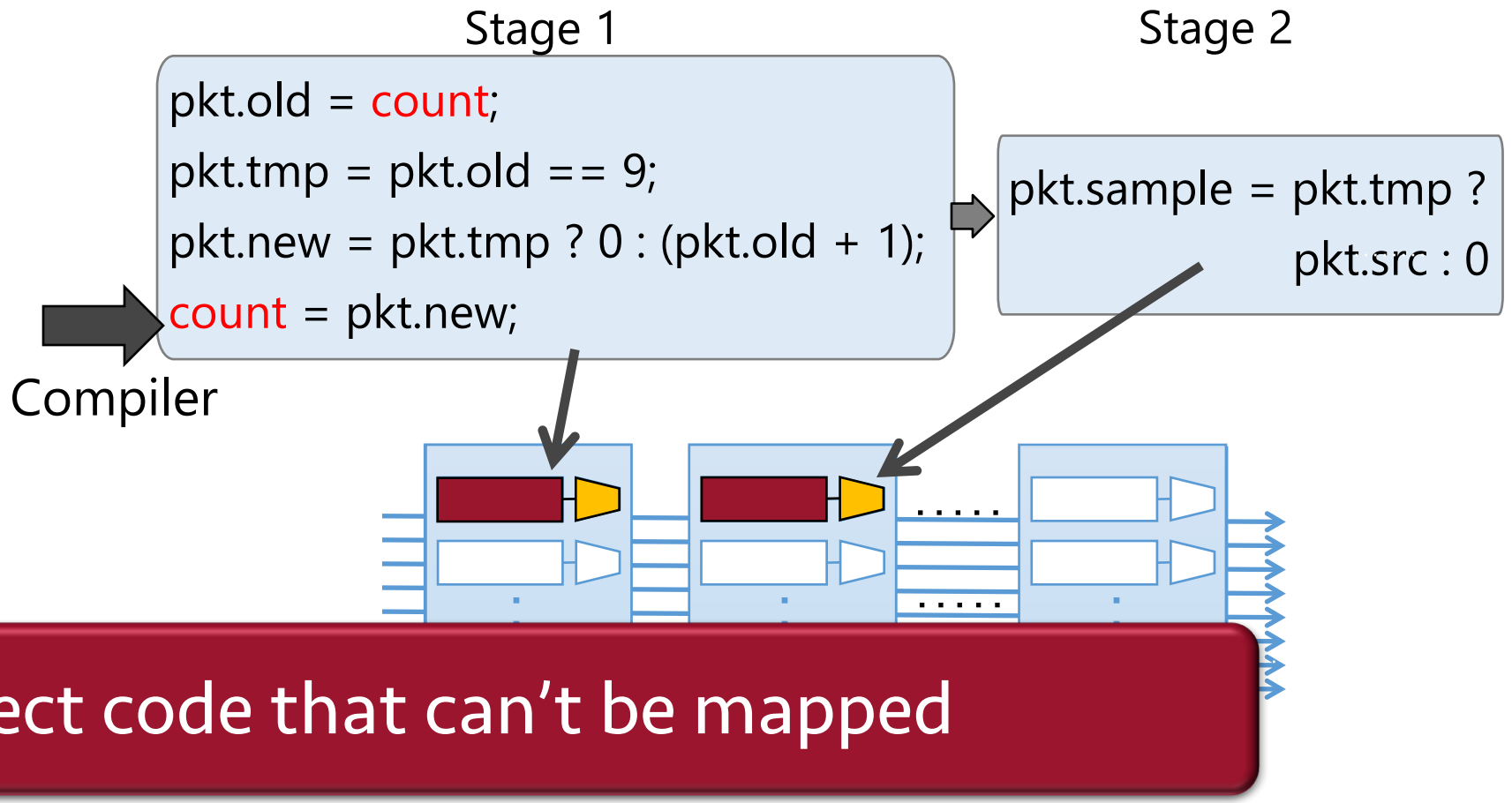
Algorithm	LOC
Bloom filter	29
Heavy hitter detection	35
Rate-Control Protocol	23
Flowlet switching	37
Sampled NetFlow	18
HULL	26
Adaptive Virtual Queue	36
CONGA	32
CoDel	57

Compiling packet transactions

Packet Sampling Algorithm

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

Packet Sampling Pipeline



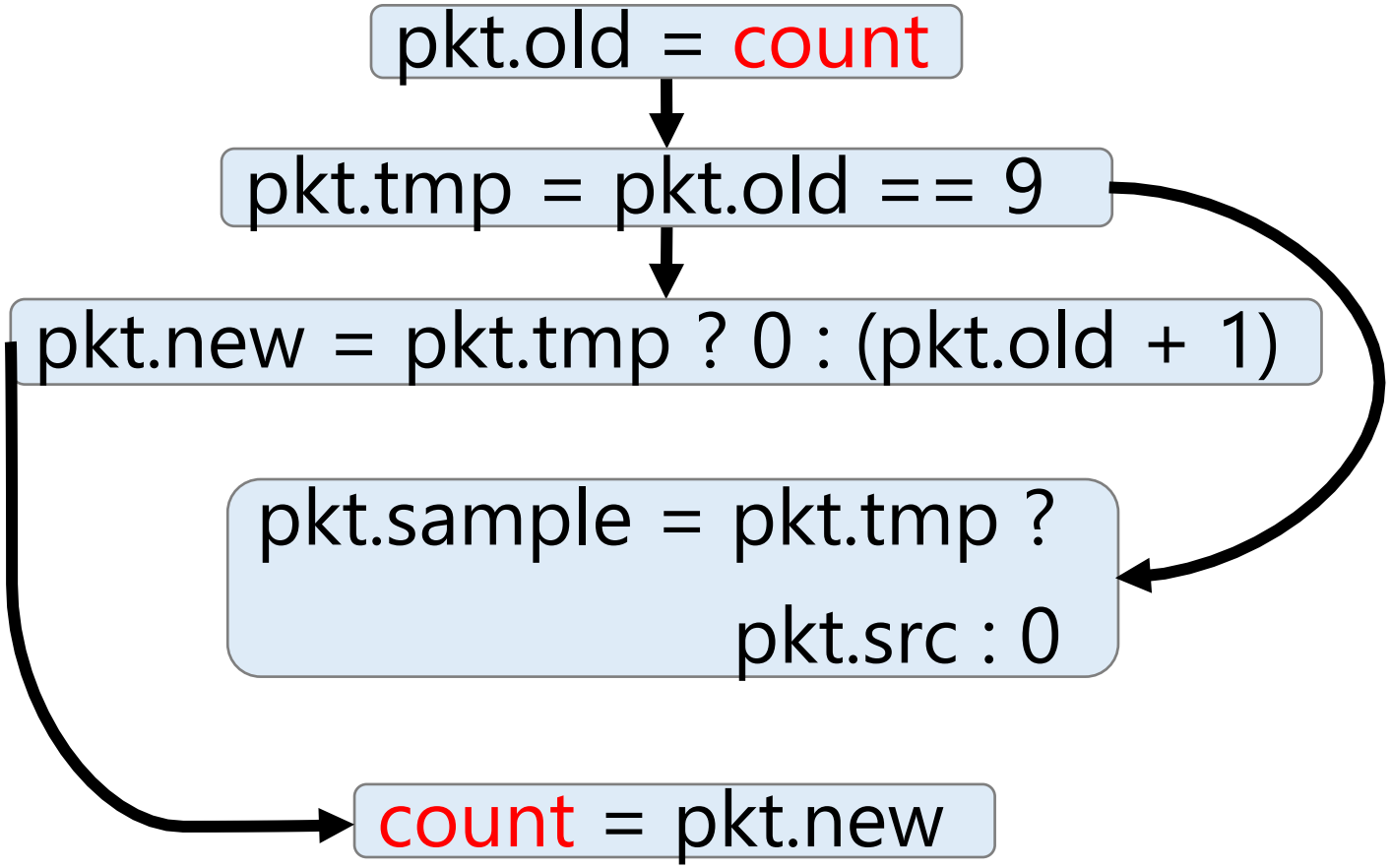
Reject code that can't be mapped

(1) Serial code to codelet pipeline

```
pkt.old = count

pkt.tmp = pkt.old == 9

pkt.new = pkt.tmp ? 0 : (pkt.old + 1)

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

count = pkt.new
```

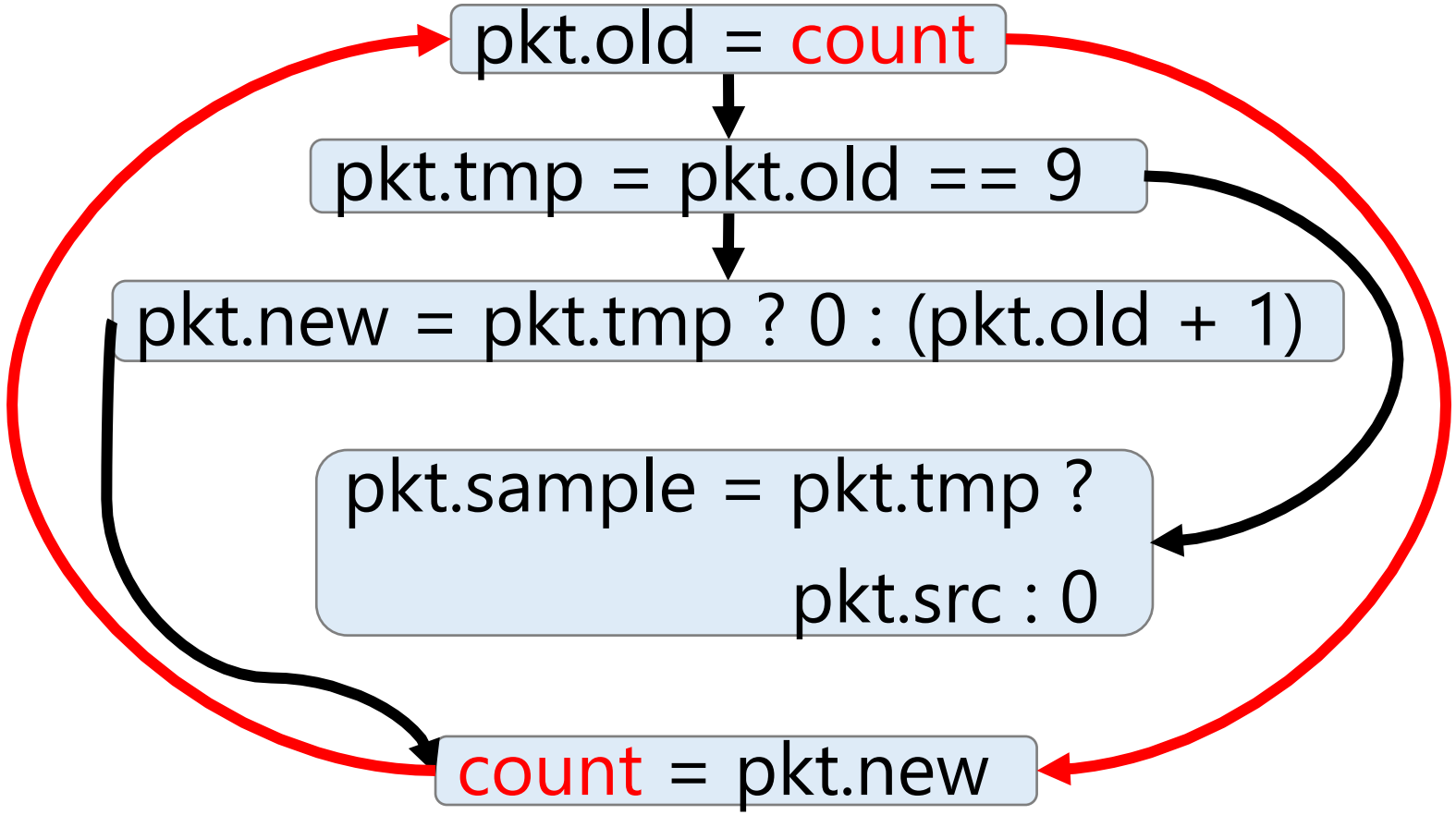
Create one node for each instruction

(1) Serial code to codelet pipeline



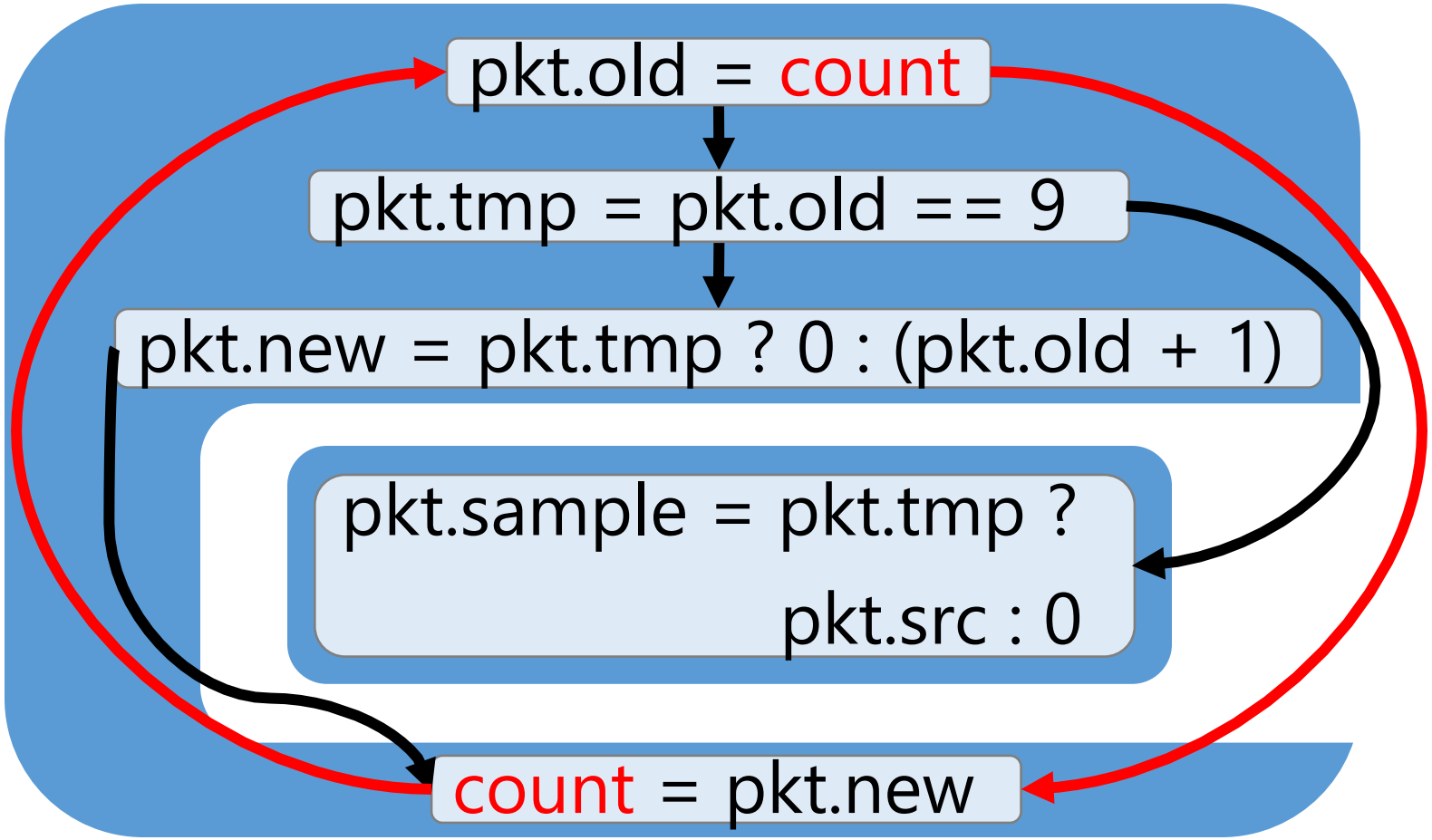
Packet field dependencies

(1) Serial code to codelet pipeline



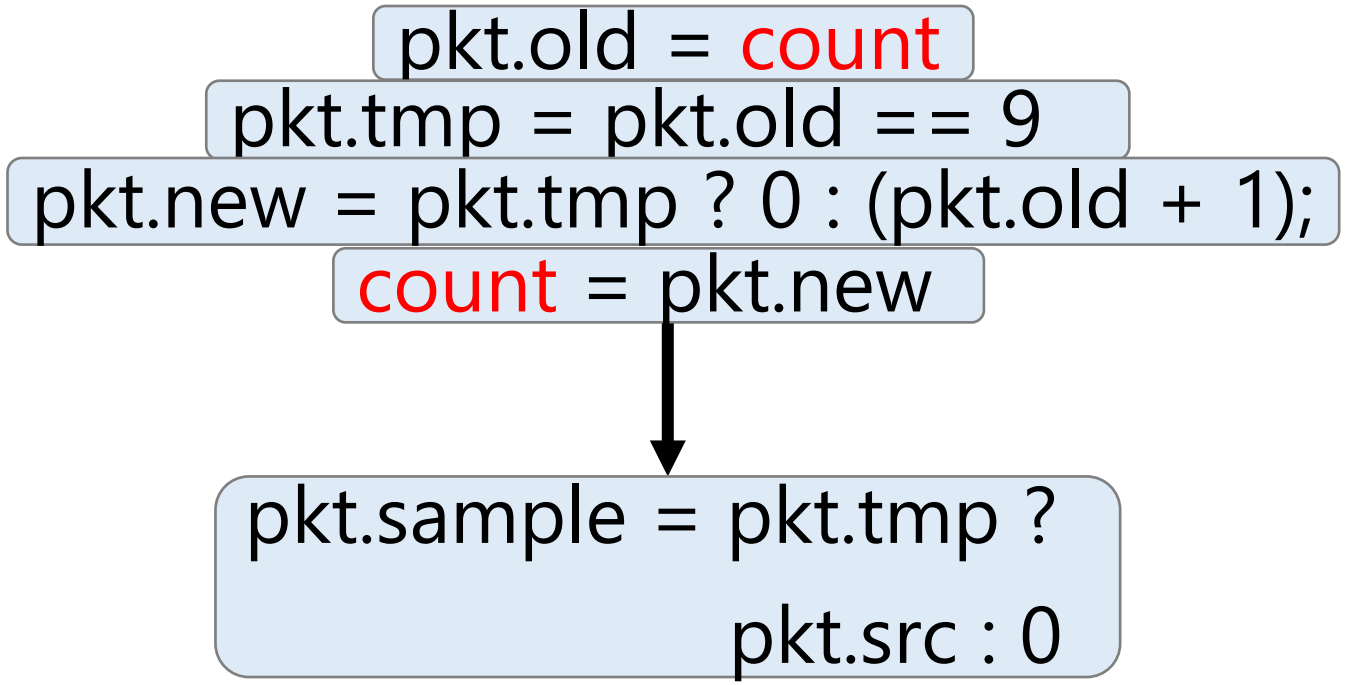
State dependencies

(1) Serial code to codelet pipeline



Strongly connected components

(1) Serial code to codelet pipeline



Condensed DAG

(1) Serial code to codelet pipeline

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
```

Code pipelining

(2) Codelets to atoms

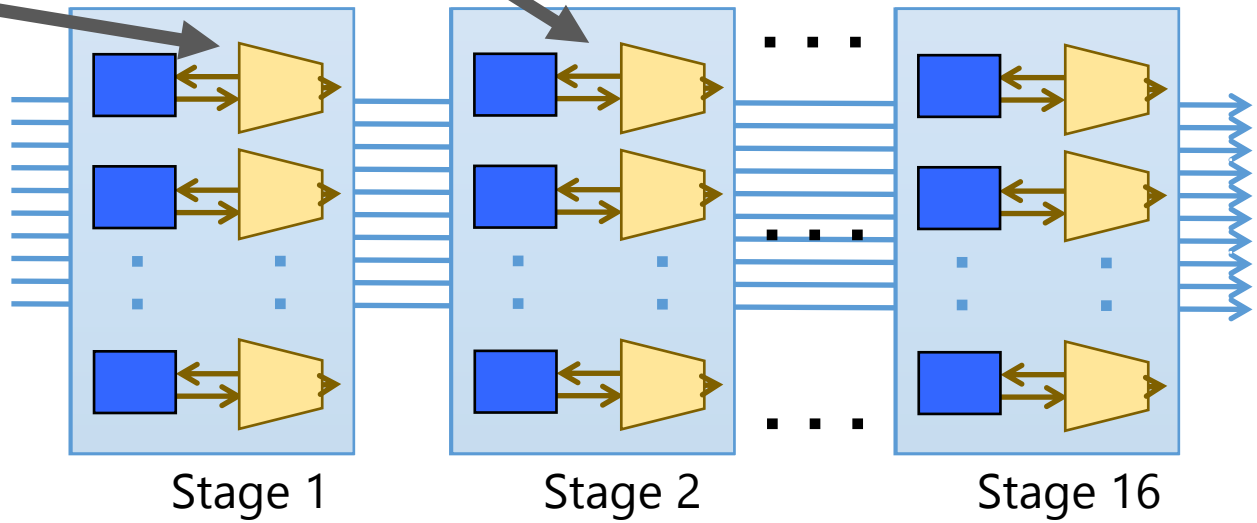
Stage 1

```
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
```

Assign each codelet to one atom. Reject if you run out of atoms.

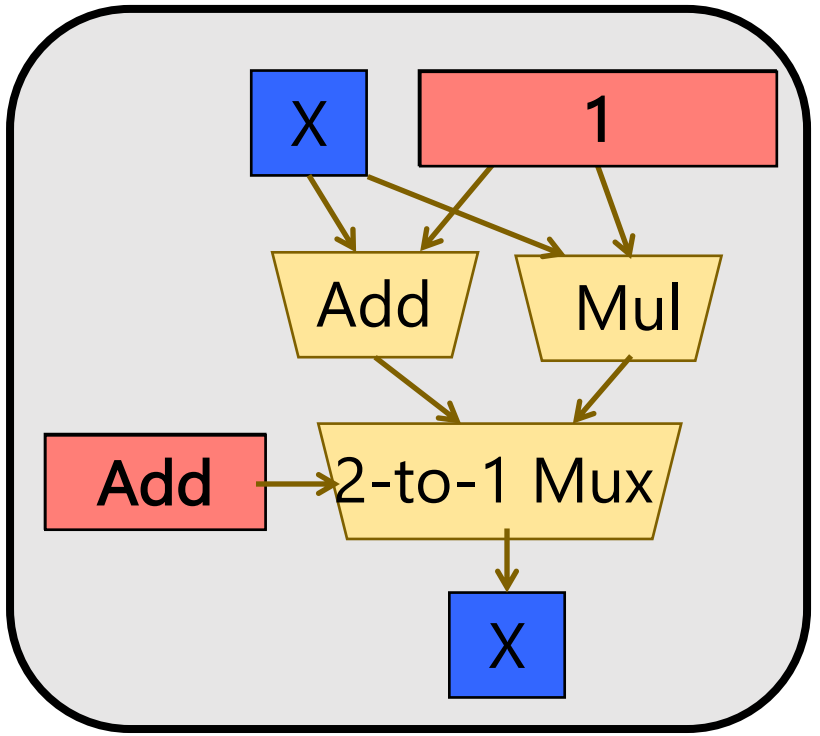


(2) Codelets to atoms

x = x + 1 maps to this atom

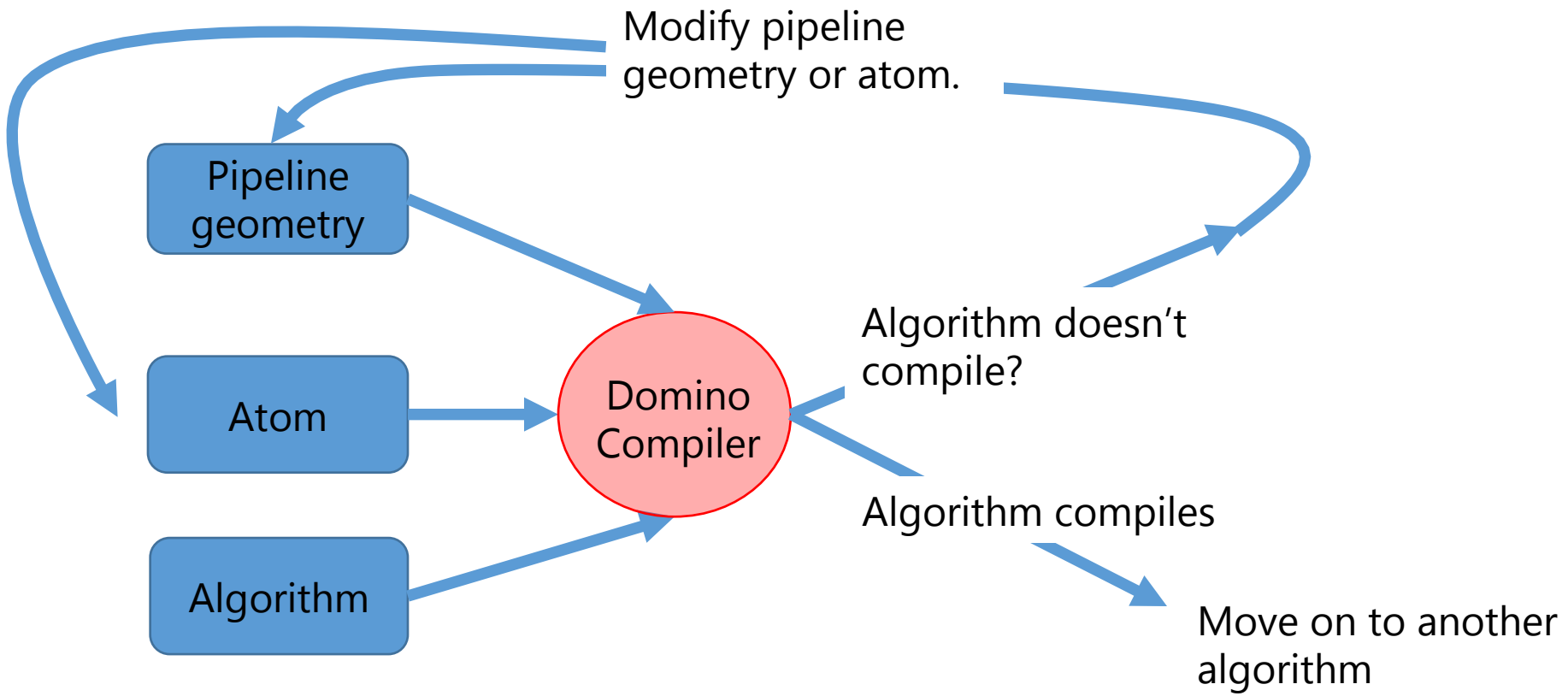
x = x * x doesn't map, reject code

Use program synthesis for mapping problem.



Determines if algorithm can run at line rate

The compiler as a tool for switch design



Demo

Stateful atoms for programmable switches

Atom	Description
R/W	Read or write state
RAW	Read, add, and write back
PRAW	Predicated version of RAW
IfElseRAW	2 RAWs, one each when a predicate is true or false
Sub	IfElseRAW with a stateful subtraction capability
Nested	4-way predication (nests 2 IfElseRAWs)
Pairs	Update a pair of state variables

Least Expressive

Most Expressive

Compiling packet transactions to atoms

Algorithm
Bloom filter
Heavy hitter detection
Rate-Control Protocol
Flowlet switching
Sampled NetFlow
HULL
Adaptive Virtual Queue
CONGA
CoDel

Compiling packet transactions to atoms

Algorithm	Most expressive stateful atom required
Bloom filter	R/W
Heavy hitter detection	RAW
Rate-Control Protocol	PRAW
Flowlet switching	PRAW
Sampled NetFlow	IfElseRAW
HULL	Sub
Adaptive Virtual Queue	Nested
CONGA	Pairs
CoDel	Doesn’t map

Compiling packet transactions to atoms

Algorithm	Most expressive stateful atom required	Pipeline Depth	Pipeline Width
Bloom filter	R/W	4	3
Heavy hitter detection	RAW	10	9
Rate-Control Protocol	PRAW	6	2
Flowlet switching	PRAW	3	3
Sampled NetFlow	IfElseRAW	4	2
HULL	Sub	7	1
Adaptive Virtual Queue	Nested	7	3
CONGA	Pairs	4	2
CoDel	Doesn't map	15	3

~100 atom instances are sufficient

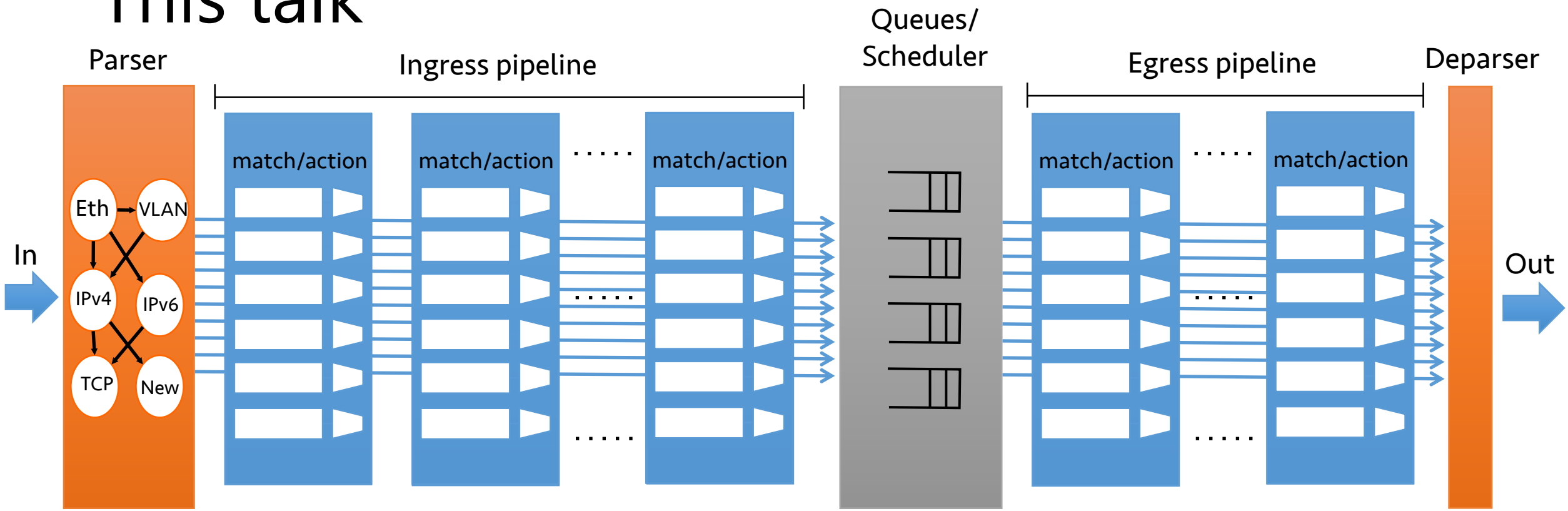
Programmability adds modest cost

- All atoms meet timing at 1 GHz in a 32-nm library.
- They occupy modest additional area relative to a switching chip.

Atom	Description	Atom area (micro m^2)	Area for 100 atoms relative to 200 mm^2 chip
R/W	Read or write state	250	0.0125%
RAW	Read, add, and write back	431	0.022%
PRAW	Predicated version of RAW	791	0.039%
IfElseRAW	2 RAWs, one each when a predicate is true or false	985	0.049%
Sub	IfElseRAW with a stateful subtraction capability	1522	0.076%
Nested	4-way predication (nests 2	3597	0.179%

<1 % additional area for 100 atom instances

This talk



- The machine model: Formalizes the computational capabilities of line-rate switches
- Packet transactions: High-level programming for the switch pipeline
- Push-In First-Out Queues: Programming the scheduler

Why is programmable scheduling hard?

- Many algorithms, yet no consensus on abstractions, cf.
  - Parse graphs for parsing
  - Match-action tables for forwarding
  - Packet transactions for data-plane algorithms
- Scheduler has tight timing requirements
  - Can’t simply use an FPGA/CPU

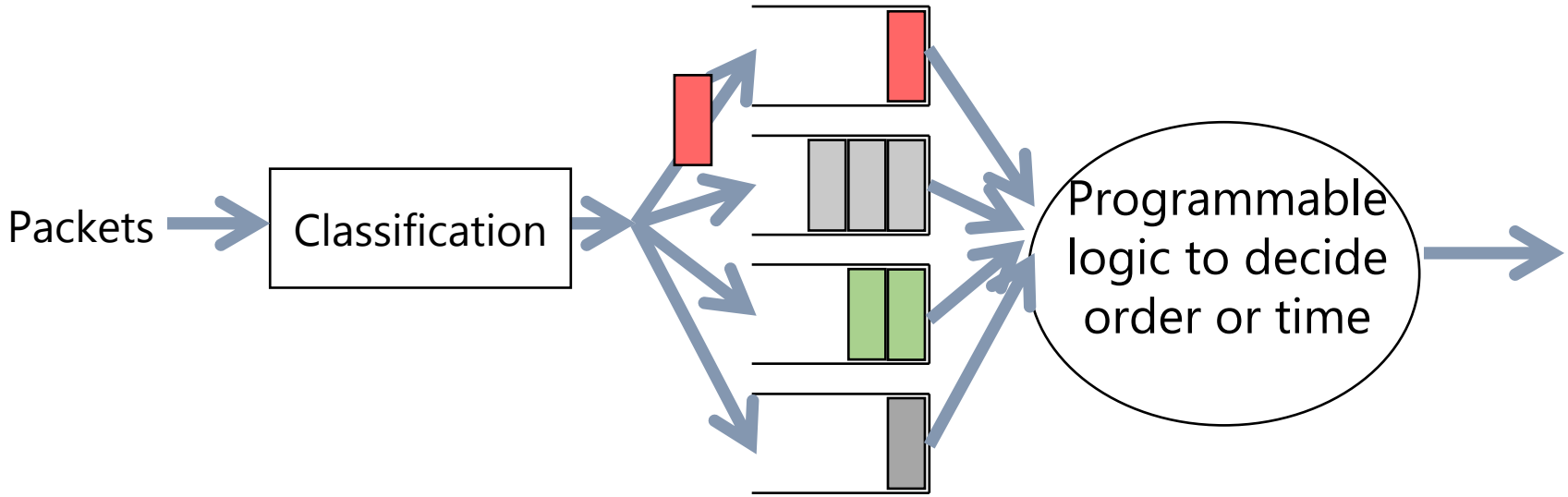
Need expressive abstraction that can run at line rate

What does the scheduler do?

It decides

- In what **order** are packets sent
  - e.g., FCFS, priorities, weighted fair queueing
- At what **time** are packets sent
  - e.g., Token bucket shaping

A strawman programmable scheduler



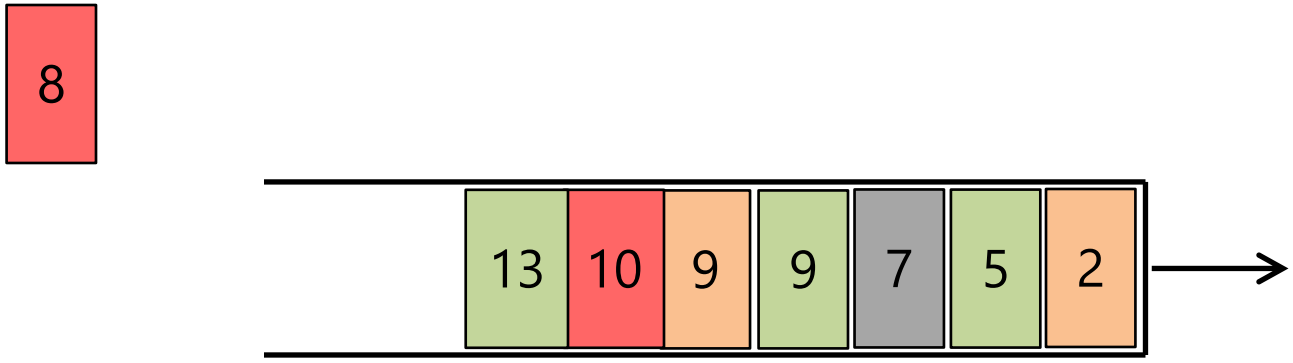
- Very little time on the dequeue side => limited programmability
- Can we move programmability to the enqueue side instead?

The Push-In First-Out Queue

Key observation

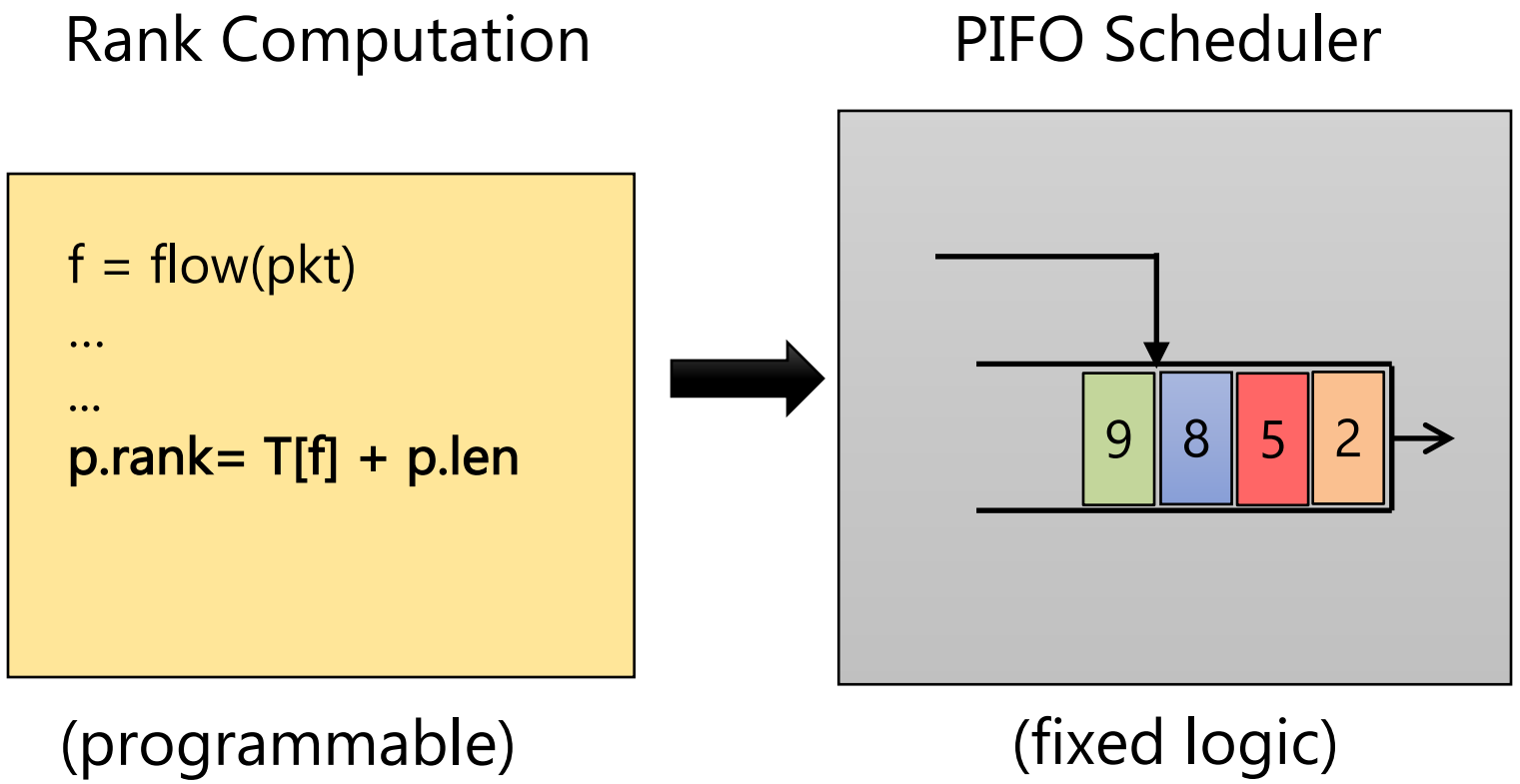
- In many schedulers, relative order of buffered packets does not change
- i.e., a packet's place in the scheduling order is known at enqueue

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

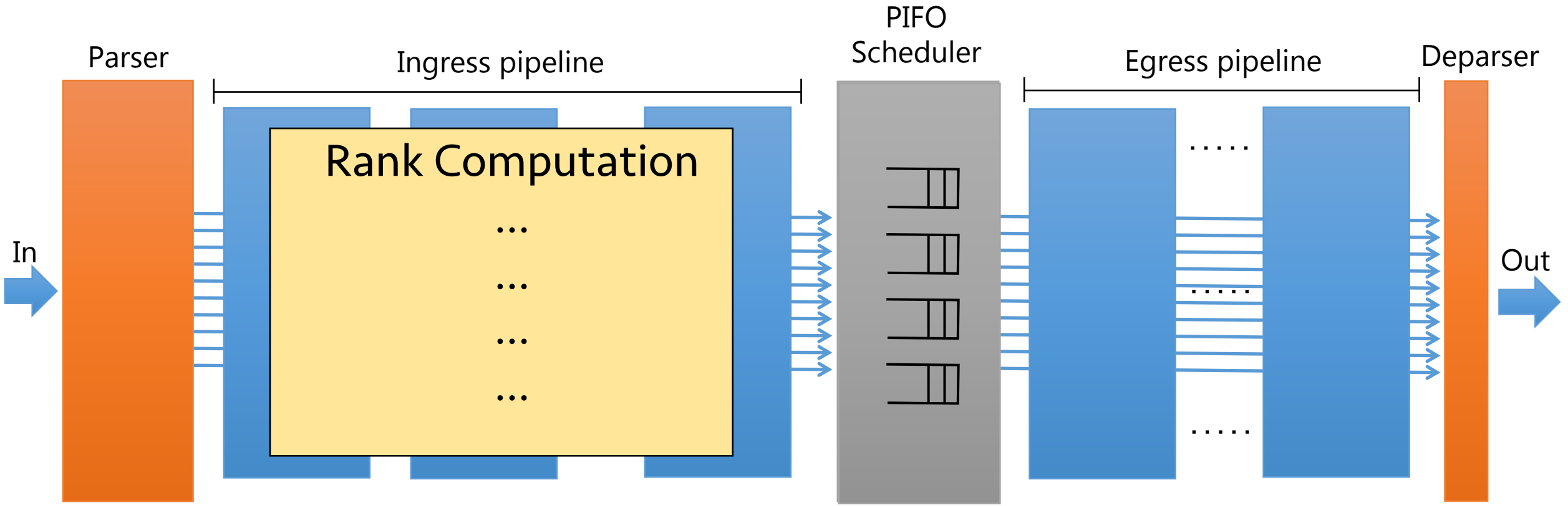


A programmable scheduler

To program the scheduler, program the rank computation

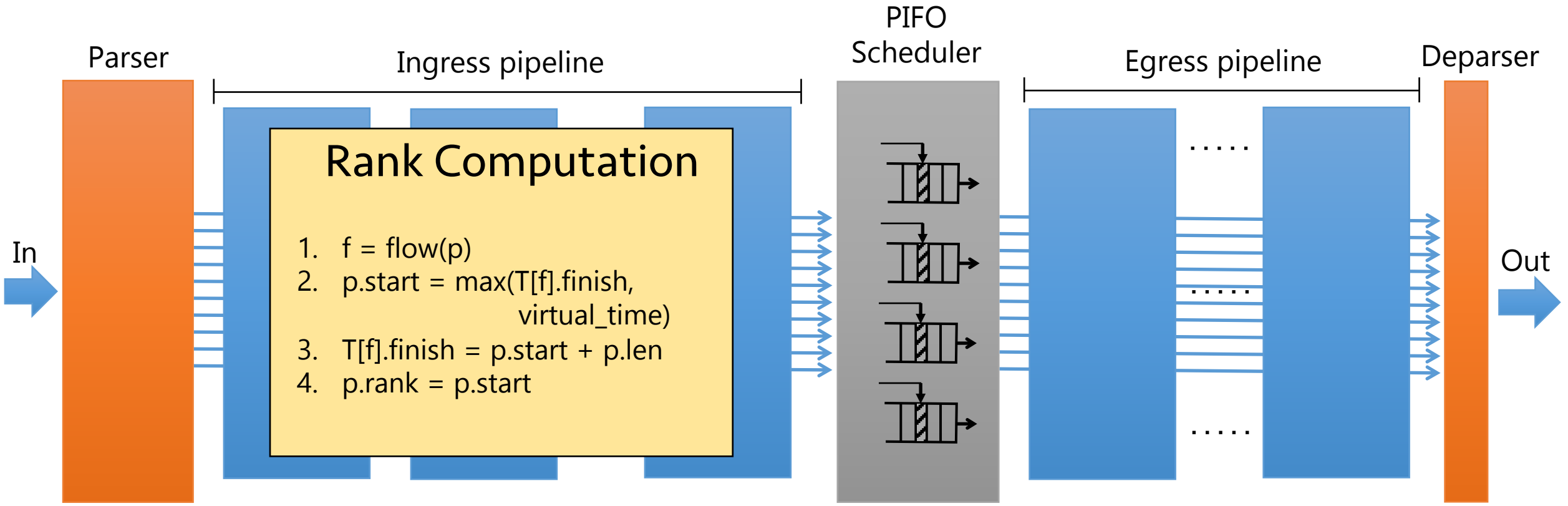


A programmable scheduler

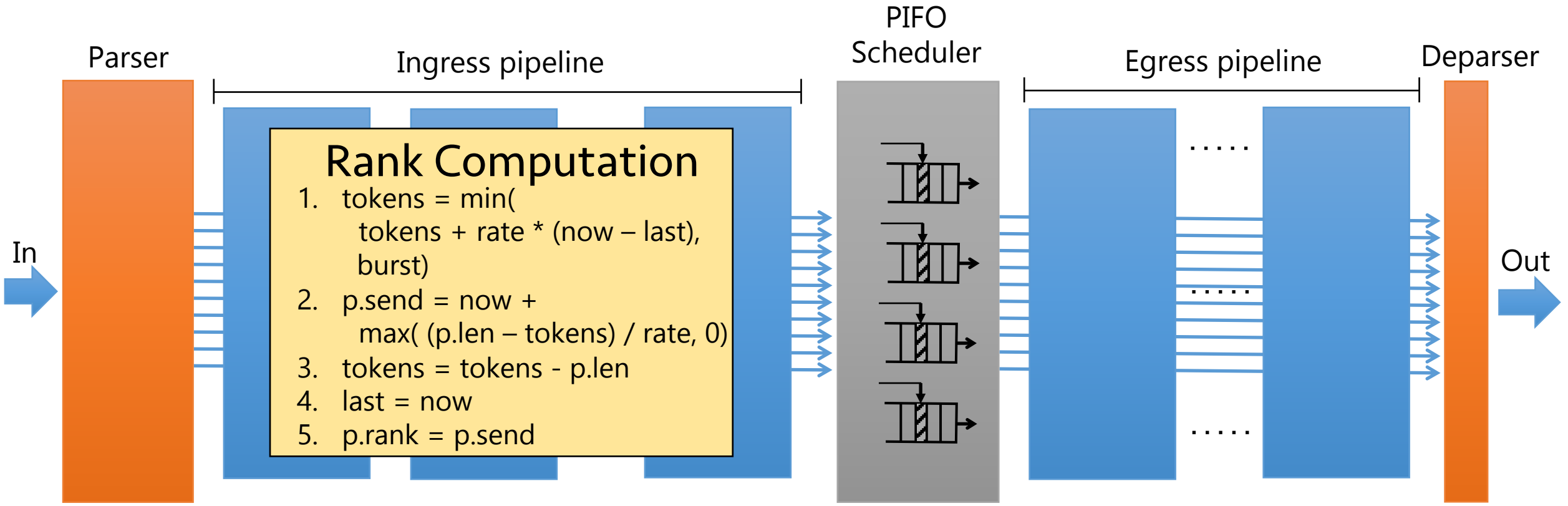


Rank computation is a packet transaction

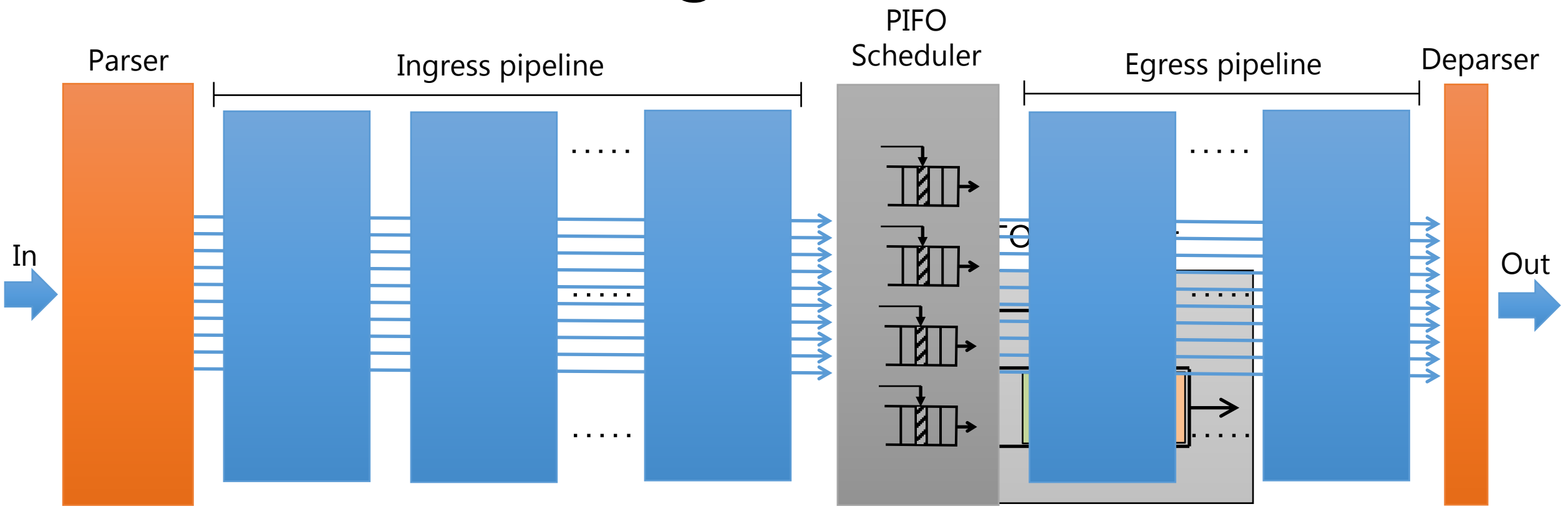
Fair queuing



Token bucket shaping



Shortest remaining flow size

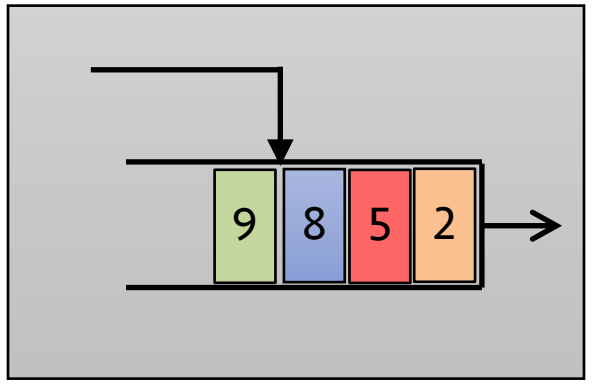


Shortest remaining flow size

Rank Computation

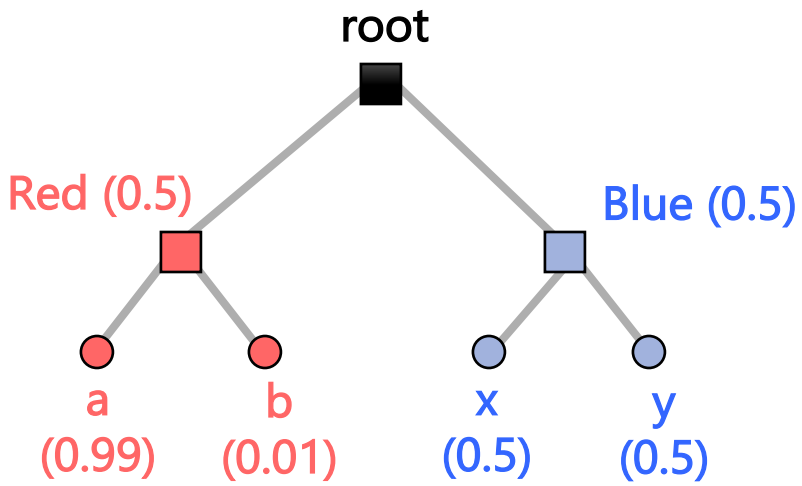
1. f = flow(p)
2. p.rank = f.rem_size

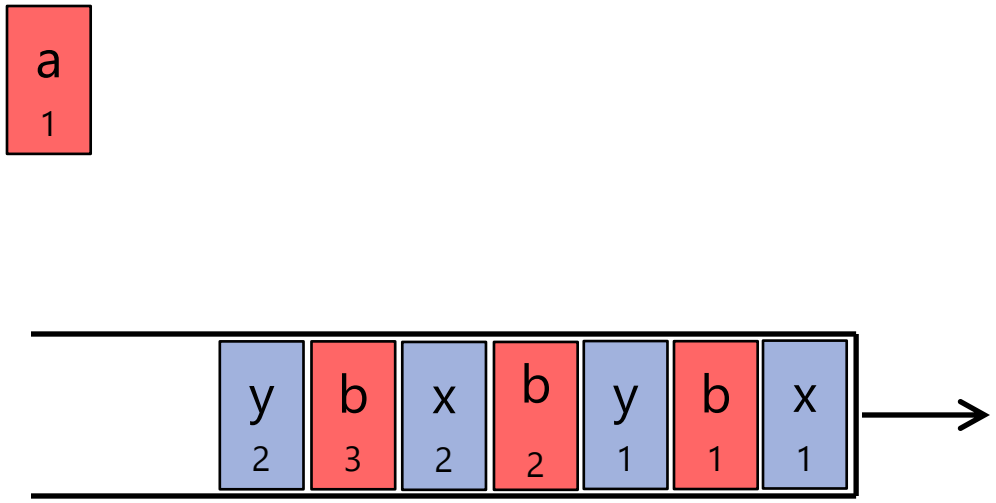
PIFO Scheduler



Beyond a single PIFO

Hierarchical
Packet Fair Queuing (HPFQ)

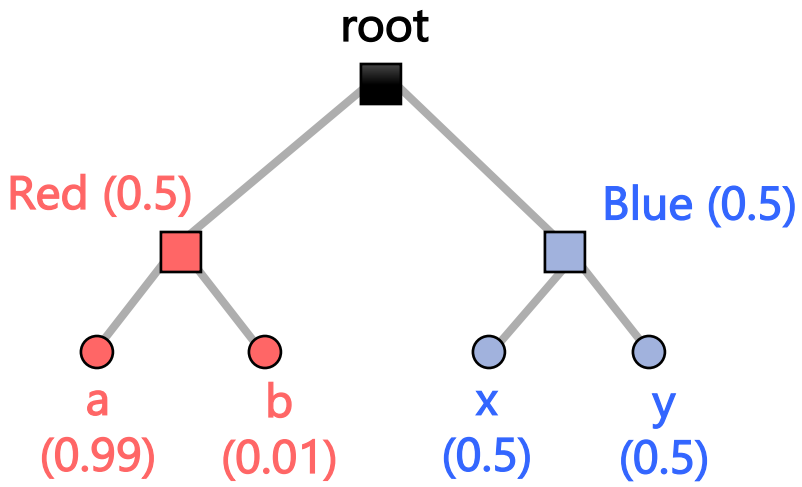




Hierarchical scheduling algorithms need hierarchy of PIFOs

Beyond a single PIFO

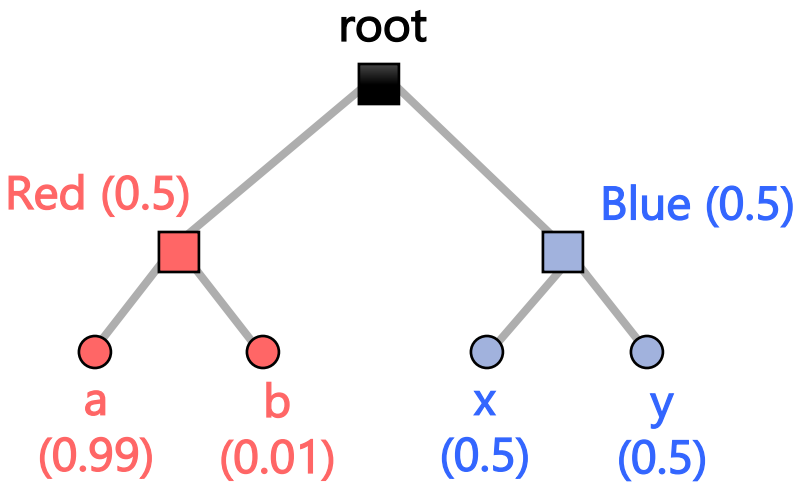
Hierarchical
Packet Fair Queuing (HPFQ)

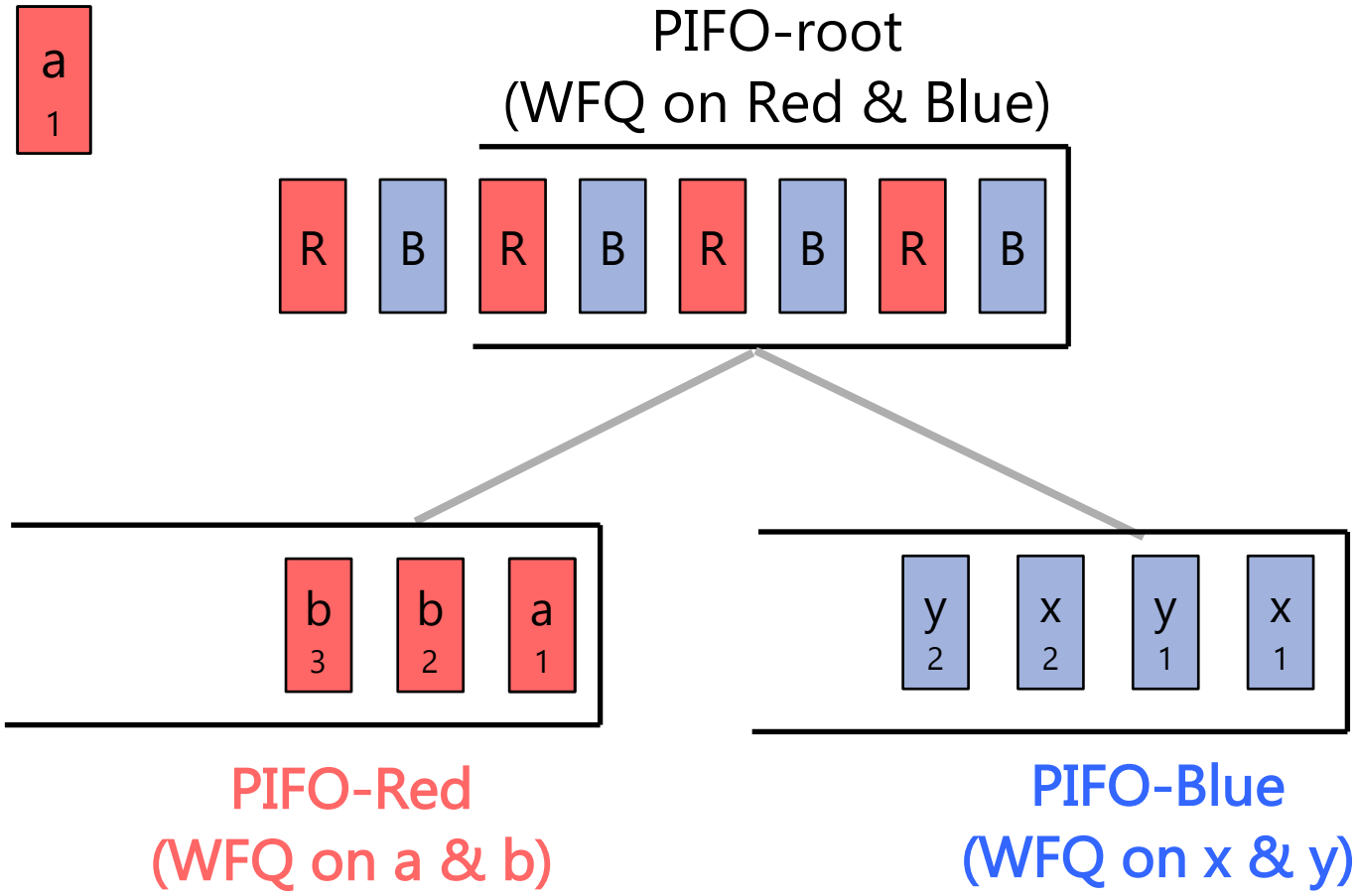


Hierarchical scheduling algorithms need hierarchy of PIFOs

Tree of PIFOs

Hierarchical
Packet Fair Queuing (HPFQ)





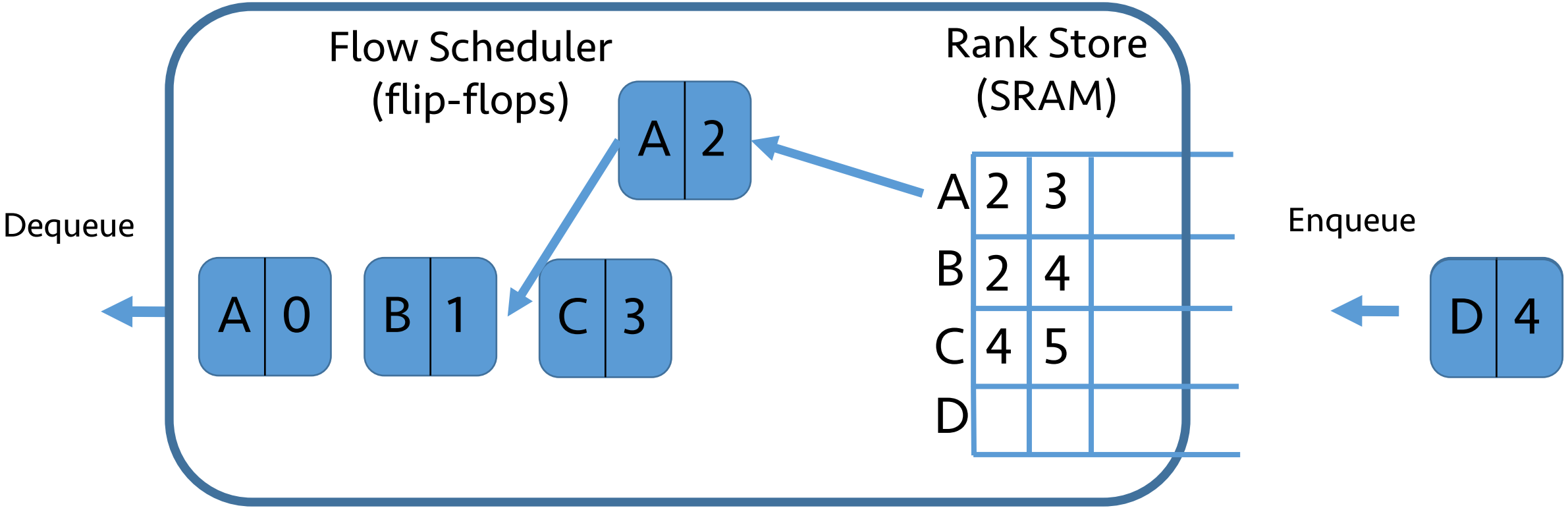
Expressiveness of PIFOs

- Fine-grained priorities: shortest-flow first, earliest deadline first, service-curve EDF
- Hierarchical scheduling: HPFQ, Class-Based Queuing
- Non-work-conserving algorithms: Token buckets, Stop-And-Go, Rate Controlled Service Disciplines
- Least Slack Time First
- Service Curve Earliest Deadline First
- Minimum and maximum rate limits on a flow
- **Cannot express some scheduling algorithms, e.g., output shaping.**

PIFO in hardware

- Performance targets for a shared-memory switch
  - 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 is infeasible
- Exploit observation that ranks increase within a flow

A single PIFO block



Hardware feasibility

- The rank store is a bank of FIFOs, used commonly to buffer data
- Flow scheduler for 1K flows meets timing at 1 GHz on a 16-nm transistor library
  - Continues to meet timing until 2048 flows, fails timing at 4096
- 7 mm^2 area for 5-level programmable hierarchical scheduler
  - < 4% relative to a 200 mm^2 baseline chip

A blueprint for programmable switches

- High-performance networking needs specialized hardware
- Tension between specialization and programmability
- Tailor abstractions to **restricted classes** of switch functions
  - Stateful header processing without loops: Packet transactions, atoms
  - Scheduling: PIFOs
  - Network diagnostics/measurement: Performance queries (HotNets 2016)
- Software and papers:
  - http://web.mit.edu/domino (Packet transactions)
  - http://web.mit.edu/pifo (PIFOs)

Backup slides

FAQ

- How is this different from P4?
  - When we started this work a year ago, P4 was much closer to the hardware. Over time, it's gotten more high-level, thanks in some part to this work (sequential semantics, ternary operators).
  - Even now, however, P4 doesn't provide transactional or atomic semantics.
  - We do have a P4 backend.
- Why a pipeline?
  - NPUs have a shared-memory architecture, but sharing memory is hard and slows down the switch.

FAQ

- What's in the compiler?
  - Strongly Connected Components to extract atomic portions.
  - Code generation using program synthesis.
- Do the atoms generalize?
  - We don't know for sure. We designed the atoms and were able to tweak them a little bit to serve more algorithms. But this is something we don't yet have a handle on.
- Is someone implementing it?
  - We are tabling a proposal on @atomic for P4.
  - There's industry interest in PIFO, but no one I know actively working on it.

The SKETCH algorithm

- We have an automated search procedure that configures the atoms appropriately to match the specification, using a SAT solver to verify equivalence.
- This procedure uses 2 SAT solvers:

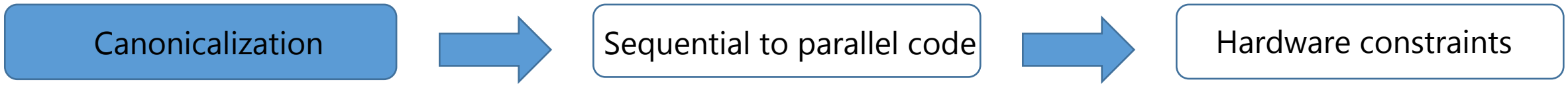
1. Generate random input x.
2. Does there exist configuration such that spec and impl. agree on random input?
3. Can we use the same configuration for all x?
4. If not, add the x to set of counter examples and go back to step 1.

Relationship to prior compiler techniques

Technique	Prior work	Differences
If Conversion	Kennedy et al. 1983	No breaks, continue, gotos, loops
Static Single-Assignment	Ferrante et al. 1988	No branches
Strongly Connected Components	Lam et al. 1989 (Software Pipelining)	Scheduling in space instead of time
Synthesis for instruction mapping	Technology mapping	Map to 1 hardware primitive, not multiple
	Superoptimization	Counter-example-guided, not brute force

Hardware feasibility of PIFOs

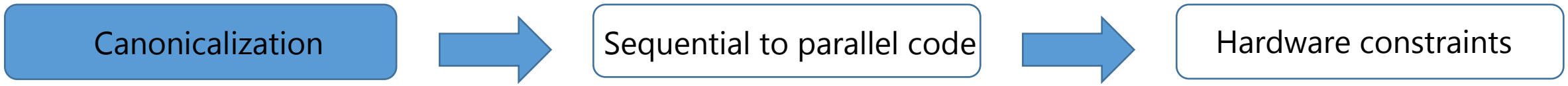
- Number of flows handled by a PIFO affects timing.
- Number of logical PIFOs within a PIFO, priority and metadata width, and number of PIFO blocks only increases area.



Static Single-Assignment

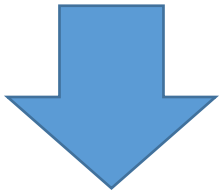
```
pkt.id = hash2(pkt.sport, pkt.dport) % NUM_FLOWLETS;
pkt.last_time = last_time[pkt.id];
...
pkt.last_time = pkt.arrival;
last_time[pkt.id] = pkt.last_time ;
```

```
pkt.id0 = hash2(pkt.sport, pkt.dport) % NUM_FLOWLETS;
pkt.last_time0 = last_time[pkt.id0];
...
pkt.last_time1 = pkt.arrival;
...
last_time [pkt.id0] = pkt.last_time1 ;
```



Expression Flattening

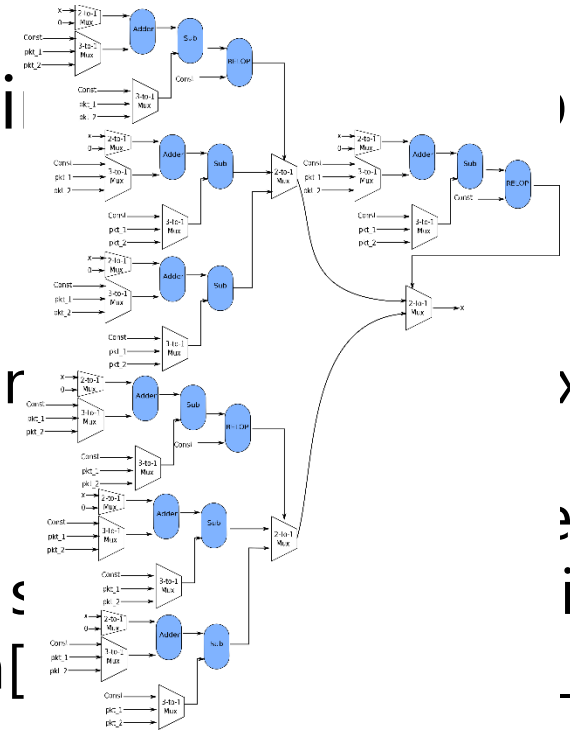
```
pkt.tmp = pkt.arrival - last_time[pkt.id] > THRESHOLD;
saved_hop [ pkt . id ] = pkt.tmp
                         ? pkt . new_hop
                         : saved_hop [ pkt . id ];
```



```
pkt.tmp    = pkt.arrival - last_time[pkt.id];
pkt.tmp2 = pkt.tmp > THRESHOLD;
saved_hop [ pkt . id ] = pkt.tmp2
                         ? pkt . new_hop
                         : saved_hop [ pkt . id ];
```

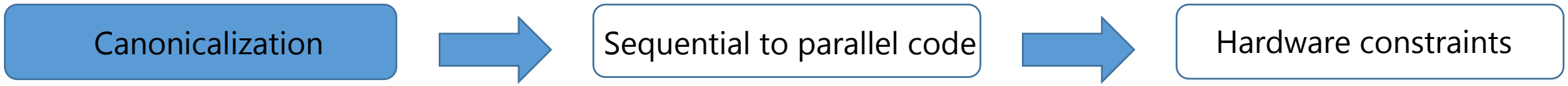
Instruction mapping: results

- Generic method to handle fairly complex templates
- Templates determi rogram can run at line rate.
- Example results:
  - Flowlet switching r xecution to save next hop information:
    saved_hop[pkt.id] ew_hop : saved_hop[pkt.id]
  - Simple increment s itter detection
    count_min_sketch[_sketch[hash] + 1



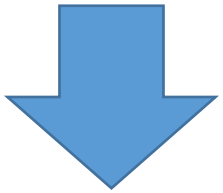
Generating P4 code

- Required changes to P4
  - Sequential execution semantics (required for read from, modify, and write back to state)
  - Expression support
  - Both available in v1.1
- Encapsulate every codelet in a table’s default action
- Chain together tables as P4 control program

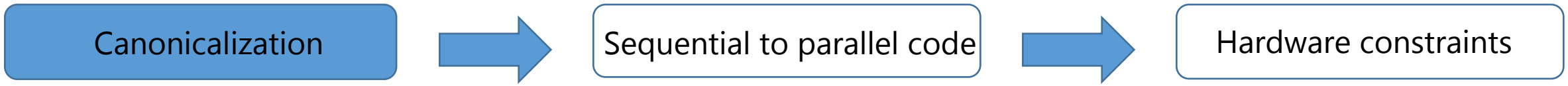


Branch Removal

```
if (pkt.arrival - last_time[pkt.id] > THRESHOLD) {
    saved_hop [ pkt . id ] = pkt . new_hop ;
 }
```

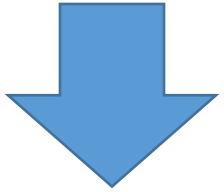


```
pkt.tmp = pkt.arrival - last_time[pkt.id] > THRESHOLD;
saved_hop [ pkt . id ] = pkt.tmp
                         ? pkt . new_hop
                         : saved_hop [ pkt . id ];
```

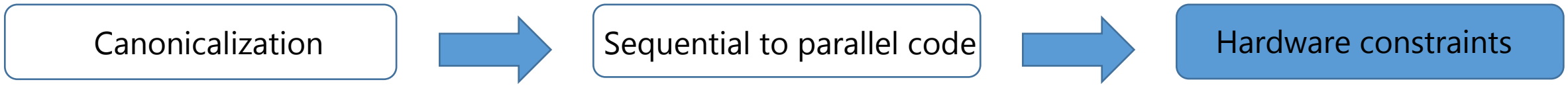


Handling State Variables

```
pkt.id = hash2(pkt.sport, pkt.dport) % NUM_FLOWLETS;
...
last_time[pkt.id] = pkt.arrival;
...
```



```
pkt.id = hash2(pkt.sport, pkt.dport) % NUM_FLOWLETS;
pkt.last_time = last_time[pkt.id]; // Read flank
...
pkt.last_time = pkt.arrival;
...
last_time[pkt.id] = pkt.last_time; // Write flank
```



Instruction mapping: the SKETCH algorithm

- Map each codelet to an atom template
- Convert codelet and template both to functions of bit vectors
- Q: Does there exist a template config s.t.

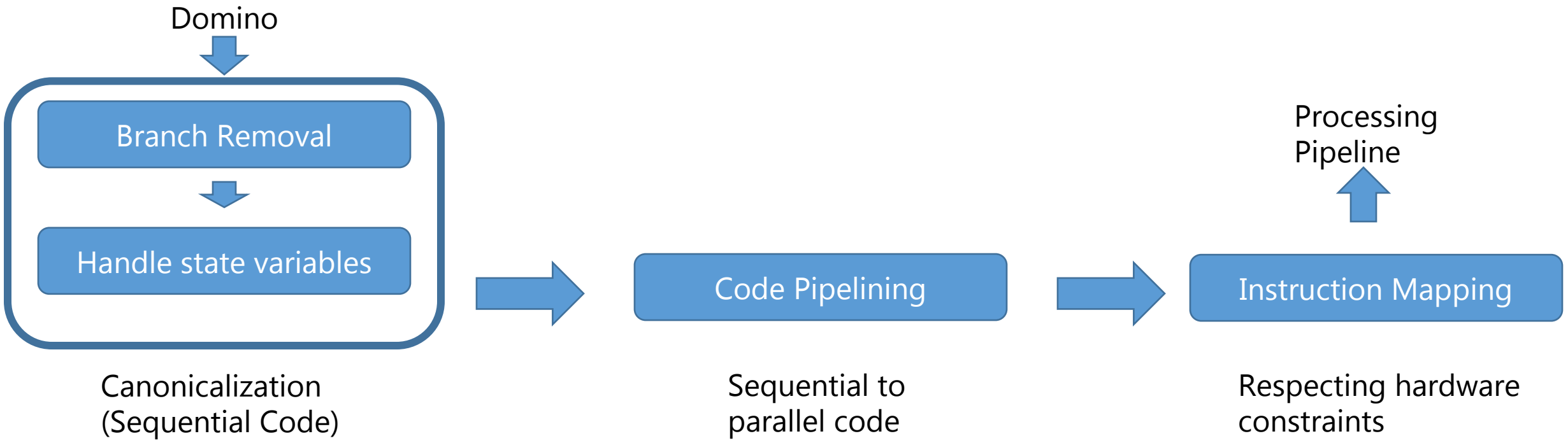
  for all inputs,

  codelet and template functions agree?
- Quantified boolean satisfiability (QBF) problem
- Use the SKETCH program synthesis tool to automate it

FAQ

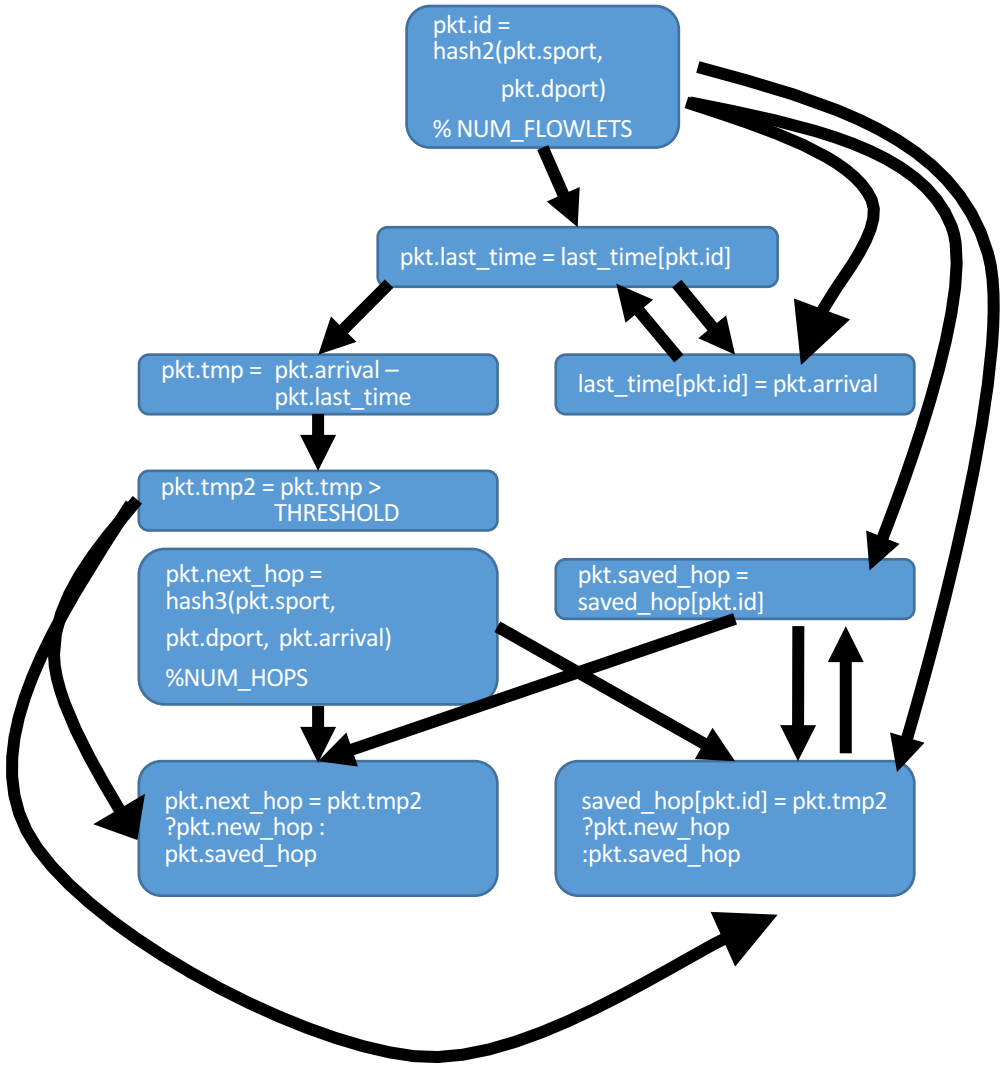
- Does predication require you to do twice the amount of work (for both the if and the else branch)?
  - Yes, but it's done in parallel, so it doesn't affect timing.
  - The additional area overhead is negligible.
- What do you do when code doesn't map?
  - We reject it and the programmer retries
- Why can't you give better diagnostics?
  - It's hard to say why a SAT solver says unsatisfiable, which is at the heart of these issues.
- Approximating square root.
  - Approximation is a good next step, especially for algorithms that are ok with sampling.
- How do you handle wrap arounds in the PIFO?
  - We don't right now.
- Is the compiler optimal?
  - No, it's only correct.

The Domino compiler



Canonicalization → Sequential to parallel code → Hardware constraints

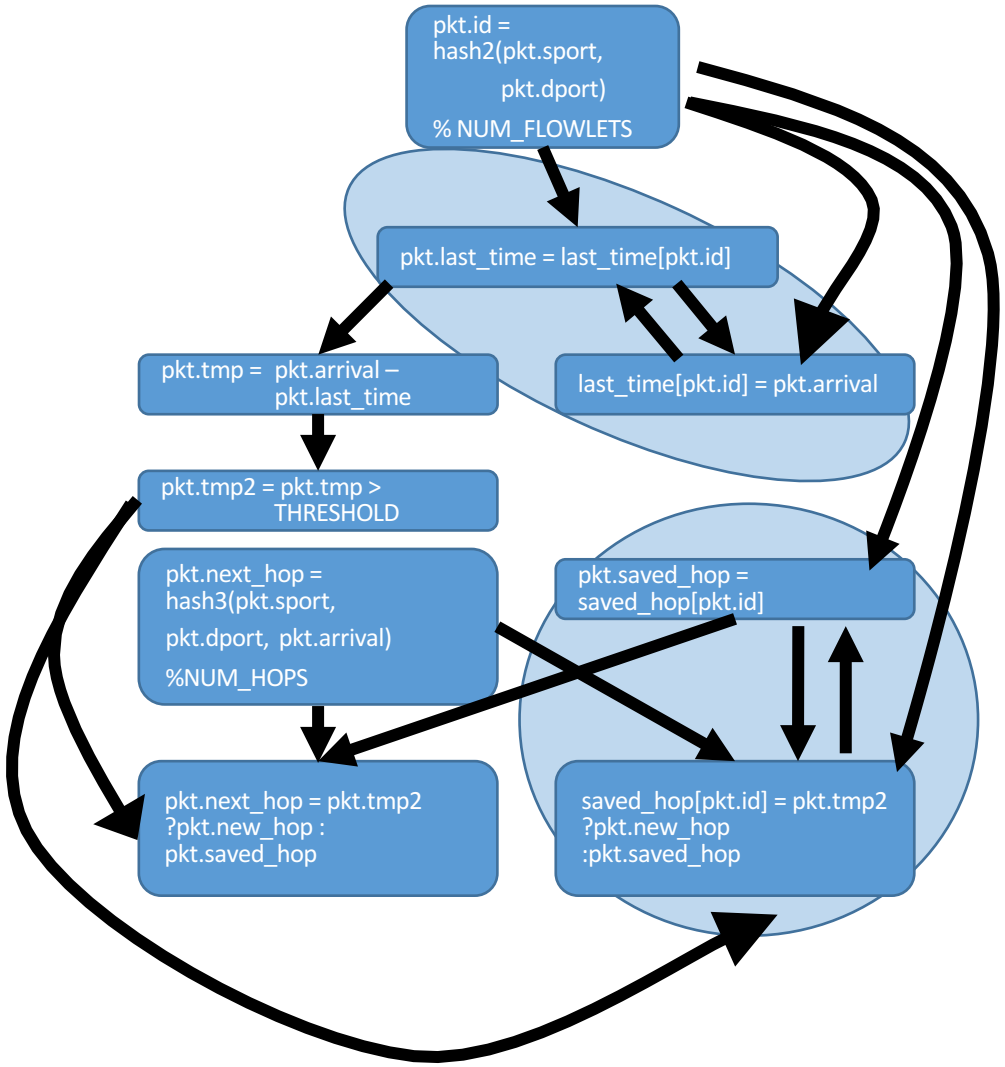
Code Pipelining



Pair up read/write flanks

Canonicalization → Sequential to parallel code → Hardware constraints

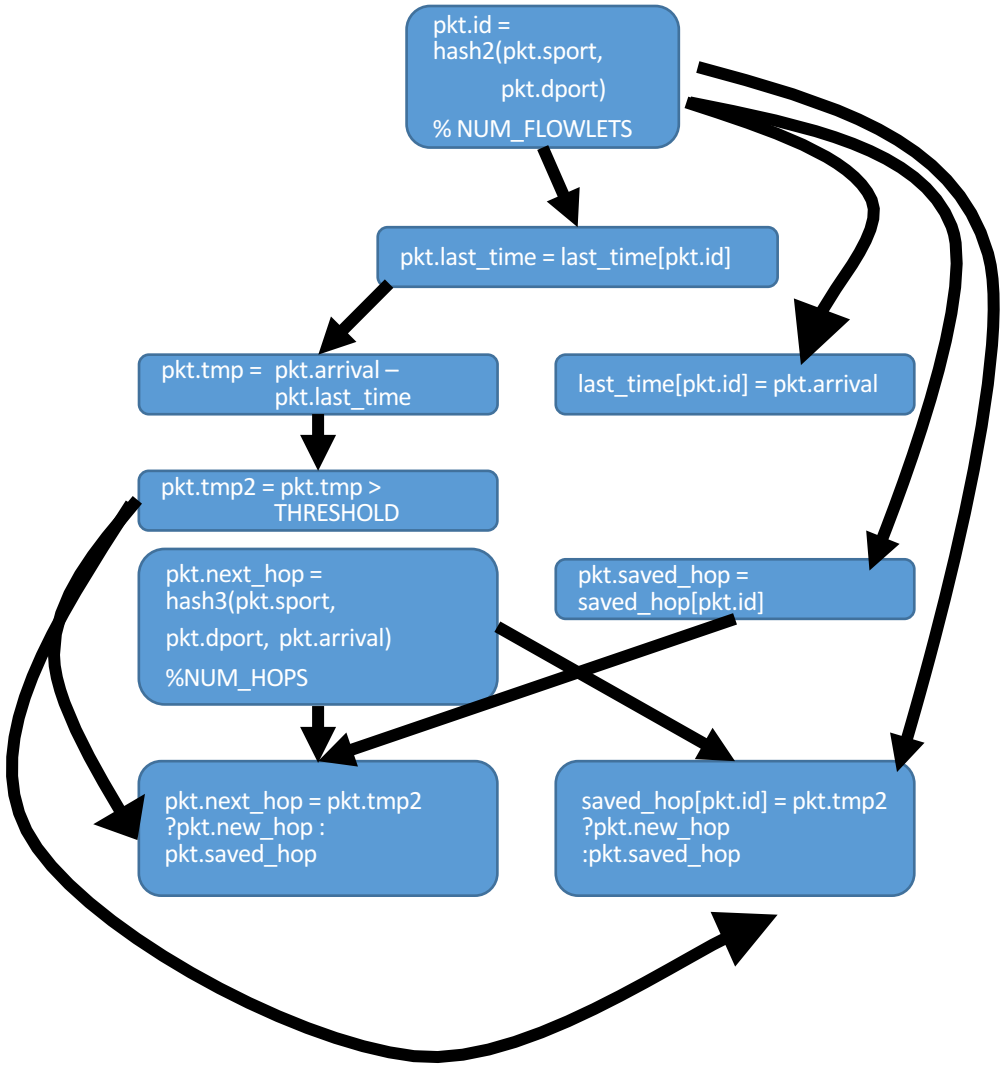
Code Pipelining



Condense strongly connected components into codelets

Canonicalization → Sequential to parallel code → Hardware constraints

Code Pipelining



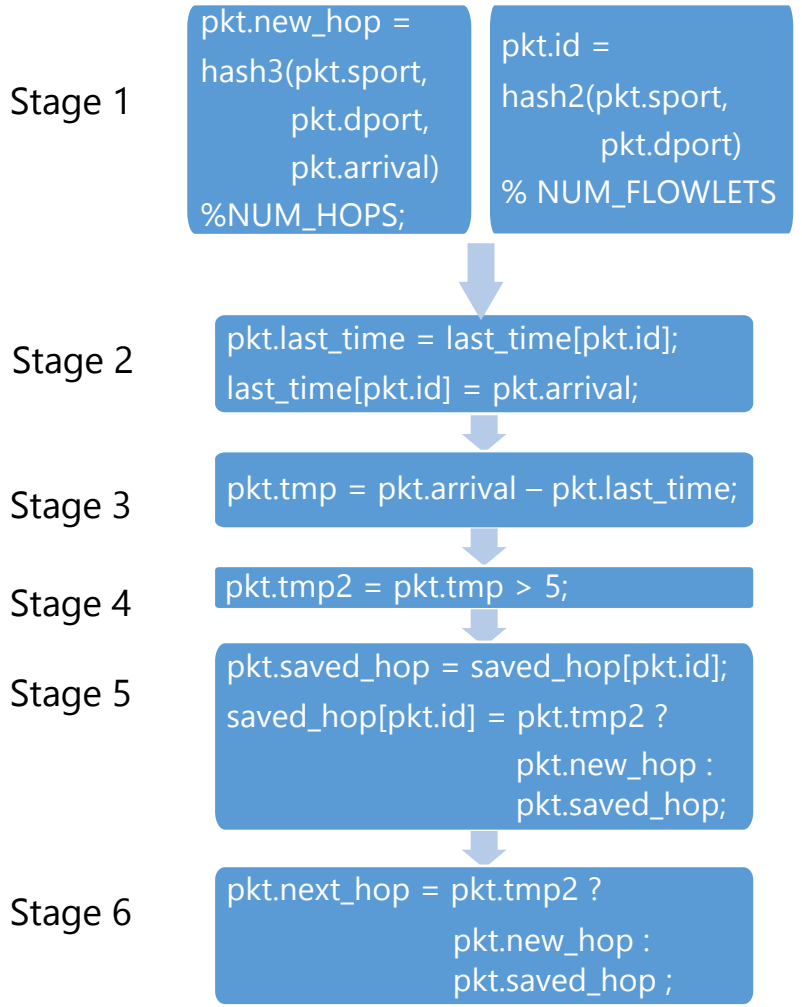
Add packet-field dependencies

Programming with Packet Transactions

Domino

```
#define NUM_FLOWLETS 8000
#define THRESHOLD    5
#define NUM_HOPS     10

struct Packet { int sport; int dport; ...};

int last_time [NUM_FLOWLETS] = {0};
int saved_hop [NUM_FLOWLETS] = {0};

void flowlet(struct Packet pkt) {
  pkt.new_hop = hash3(pkt.sport, pkt.dport, pkt.arrival)
                          % NUM_HOPS;
  pkt.id  = hash2(pkt.sport, pkt.dport) % NUM_FLOWLETS;
  if (pkt.arrival - last_time[pkt.id] > THRESHOLD) {
    saved_hop[pkt.id] = pkt.new_hop;
  }
  last_time[pkt.id] = pkt.arrival;
  pkt.next_hop = saved_hop[pkt.id];
}
```

Pipeline



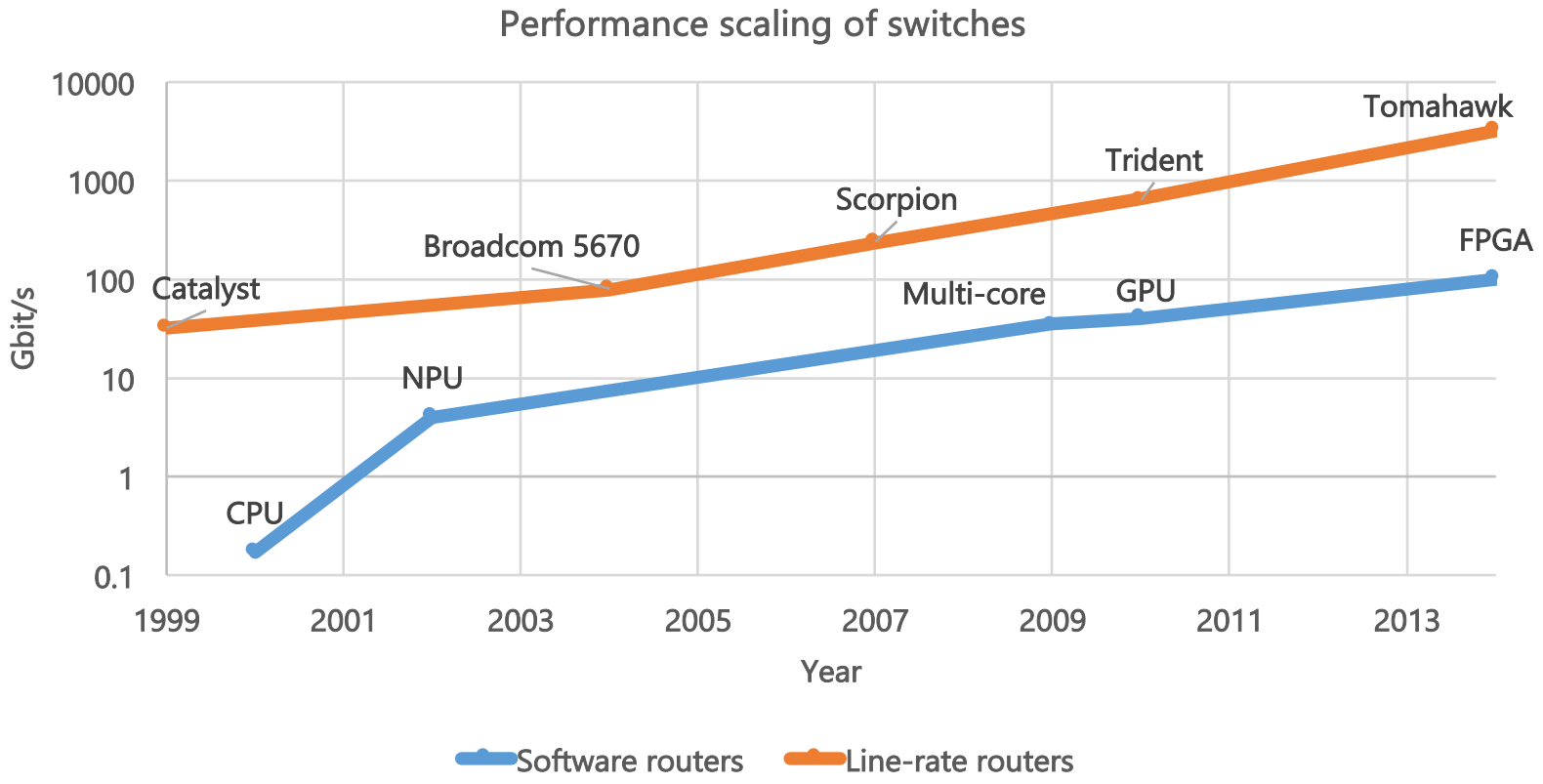
The quest for programmability

System	Year	Substrate	Performance
Click	2000	CPUs	170 Mbit/s
Intel IXP 2400	2002	NPUs	4 Gbit/s
RouteBricks	2009	Multi-core	35 Gbit/s
PacketShader	2010	GPUs	40 Gbit/s
NetFPGA SUME	2014	FPGA	100 Gbit/s

Switch	Year	Line-rate
Cisco Catalyst	1999	32 Gbit/s
Broadcom 5670	2004	80 Gbit/s
Broadcom Scorpion	2007	240 Gbit/s
Broadcom Trident	2010	640 Gbit/s
Broadcom Tomahawk	2014	3.2 Tbit/s

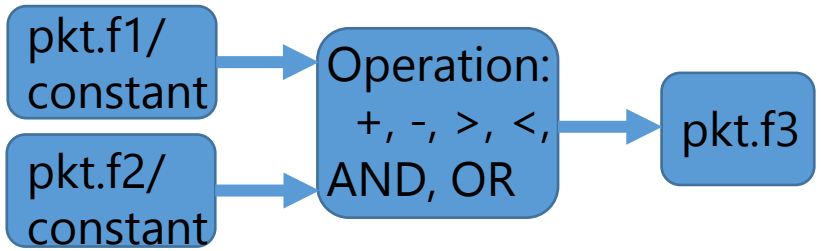
Programmability => 10--100x slower than line rate.

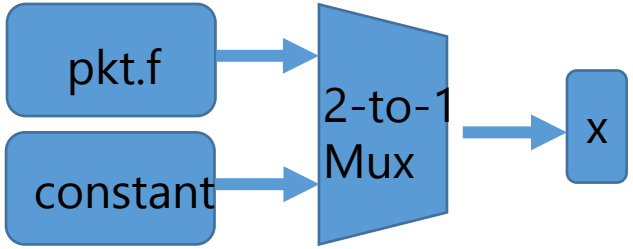
The quest for programmability

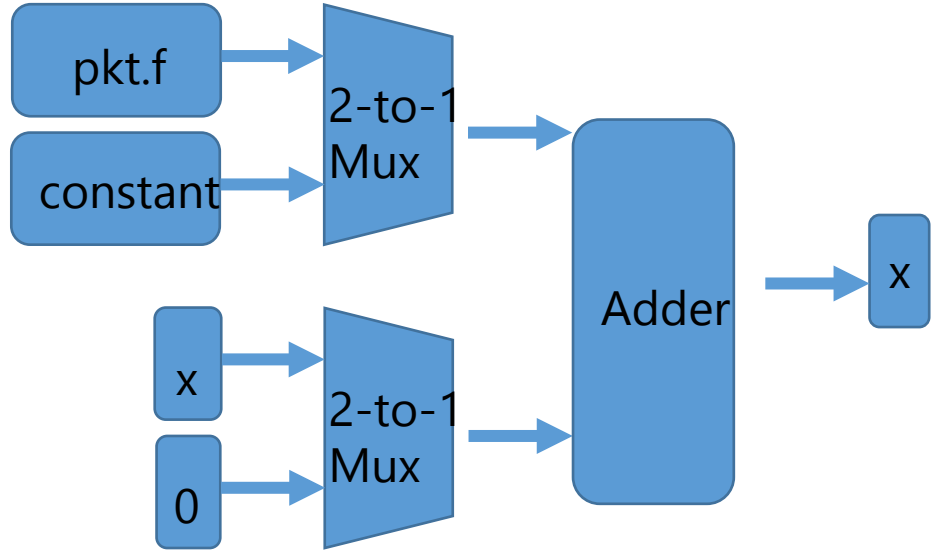


Programmability => 10--100x slower than line rate.

Compiler targets: diagram







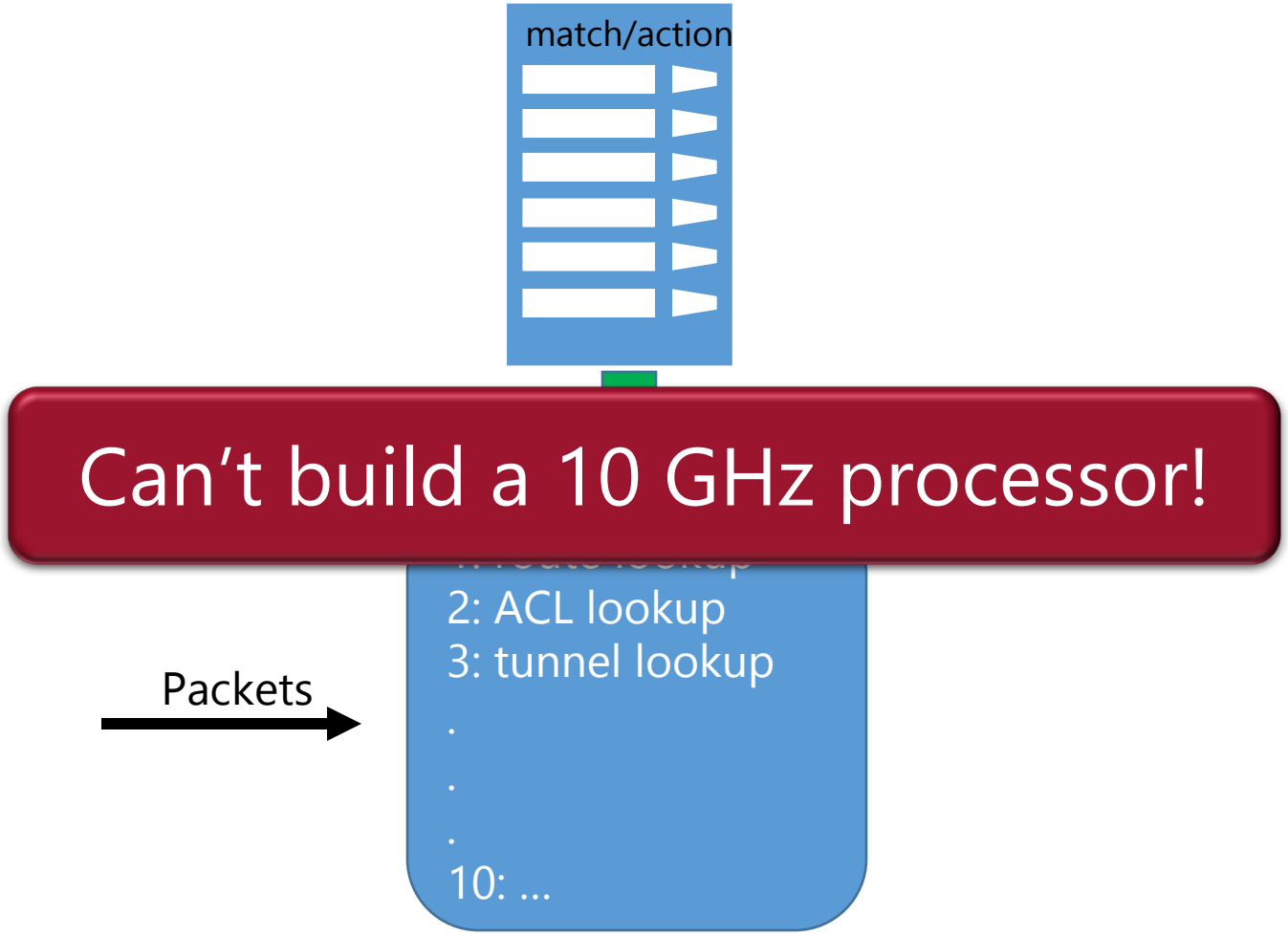
Why are switches pipelined?

Performance requirements at line rate

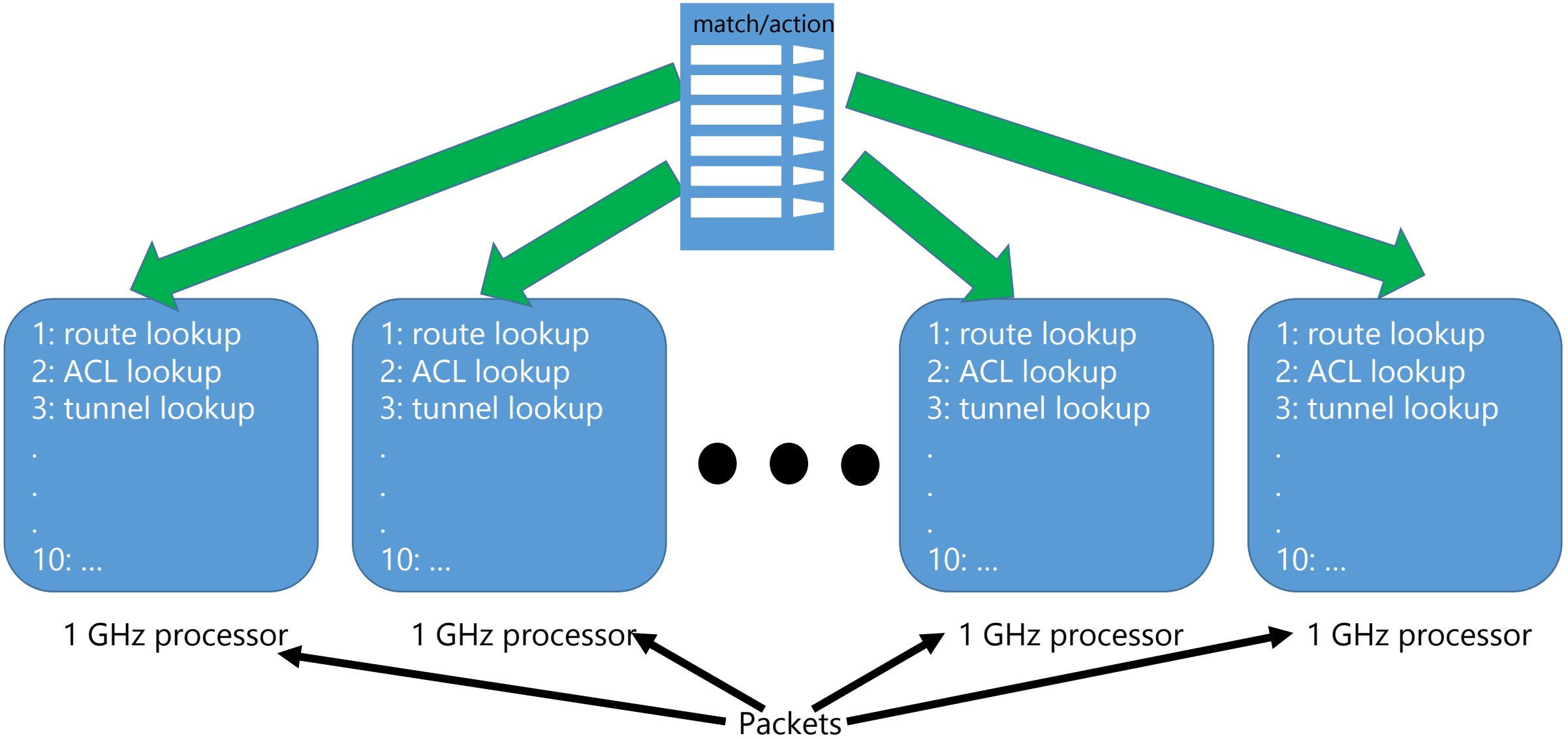
- Aggregate capacity ~ 1 Tbit/s
- Packet size ~ 1000 bits
- 10 operations per packet: routing, access control (ACL), tunnels, …

Need to process 1 billion pkts/s, 10 ops per packet

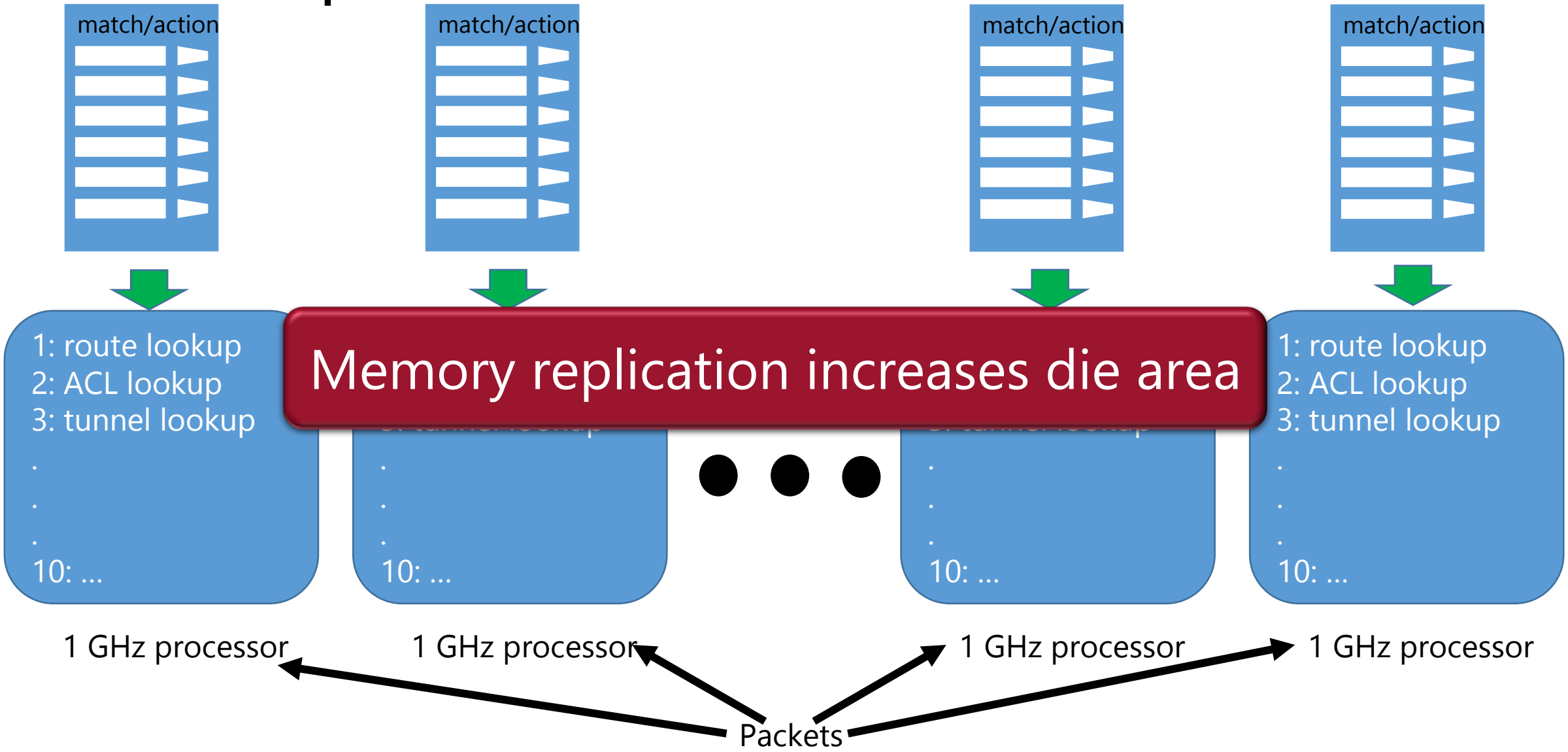
Single processor architecture



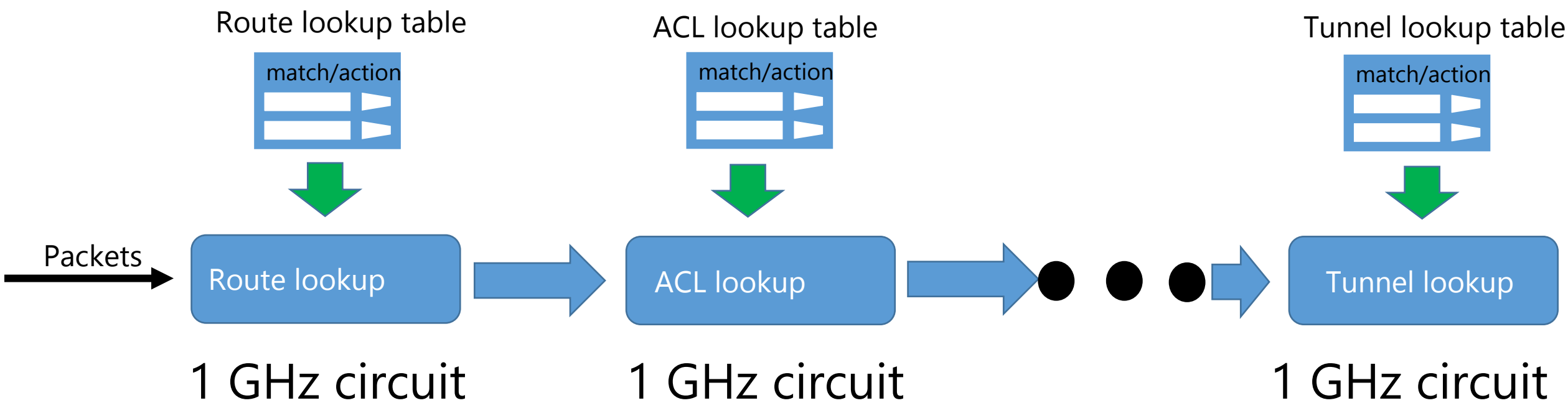
Packet-parallel architecture



Packet-parallel architecture



Function-parallel or pipelined architecture



- Factors out global state into per-stage local state
- Replaces full-blown processor with a circuit

P4 comparison

Programming with packet transactions

Algorithm	LOC	P4 LOC
Bloom filter	29	104
Heavy hitter detection	35	192
Rate-Control Protocol	23	75
Flowlet switching	37	107
Sampled NetFlow	18	70
HULL	26	95
Adaptive Virtual Queue	36	147
CONGA	32	89
CoDel	57	271