Programmable Packet Scheduling at Line Rate

Sachin Katti

Anirudh Sivaraman, Stanford, MIT, Cisco, Barefoot

(to appear in ACM SIGCOMM 2016)

Programmable switching chips

Why is programmable scheduling hard?

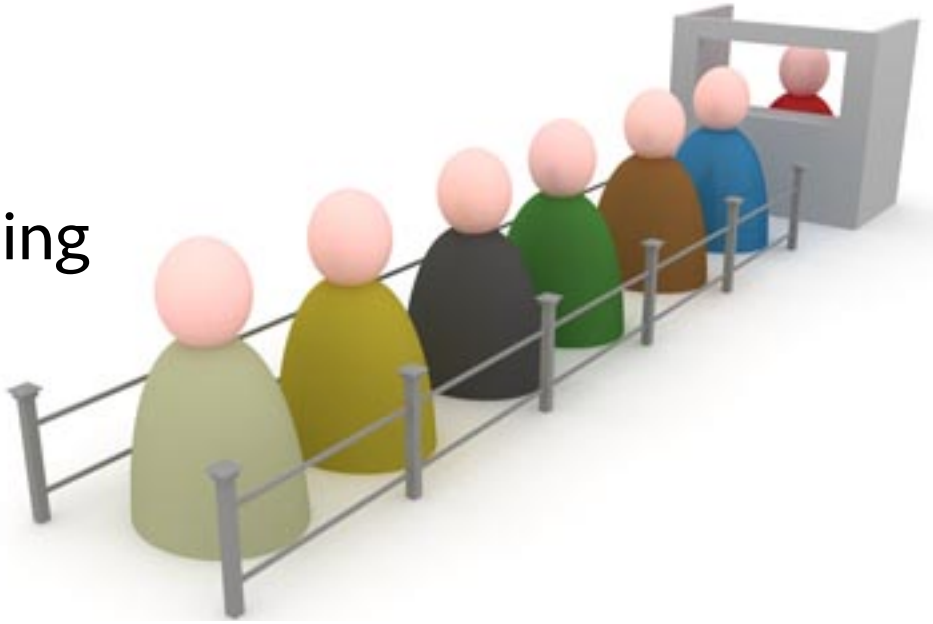
- Plenty of scheduling algorithms, but no consensus on the right abstractions for scheduling; in contrast to
  - Parse graphs for parsing
  - Match-action tables for forwarding
- The scheduler has very tight timing requirements
  - One decision per clock cycle is typical

Need expressive abstraction that can be implemented 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

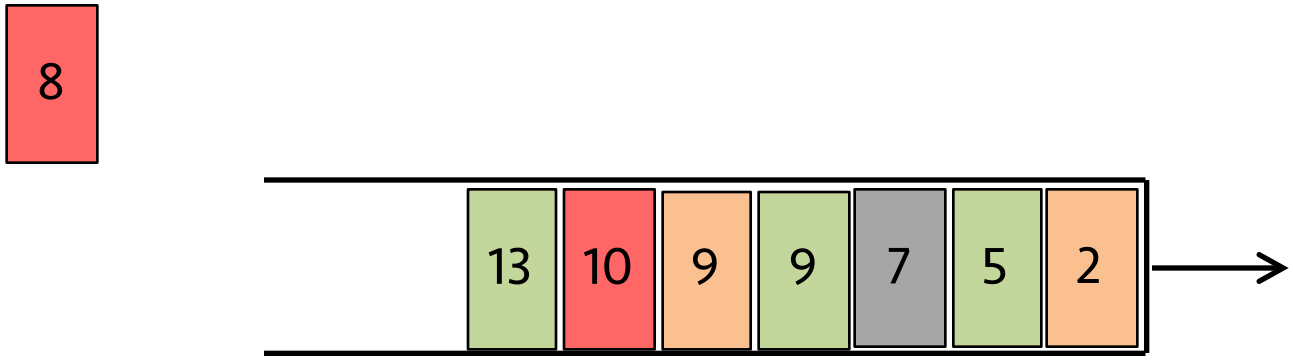


Key observation

- In many algorithms, the scheduling order/time does not change with future arrivals
- i.e., we can determine scheduling order before enqueue

The Push-In First-Out Queue

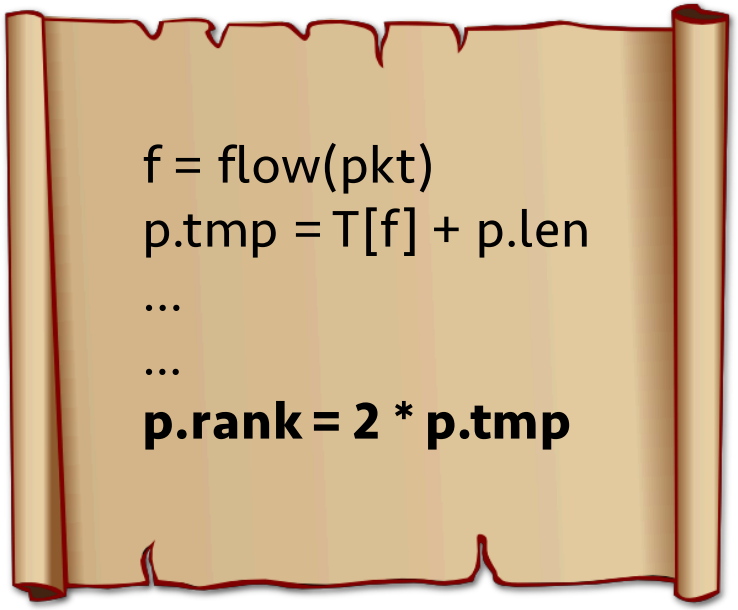
- Packets are pushed into an arbitrary location based on a **rank** number, and dequeued from the head
  - First used as a proof construct by Chuang et. al. in the 90s
  - Also a powerful construct for programmable scheduling



A programmable scheduler

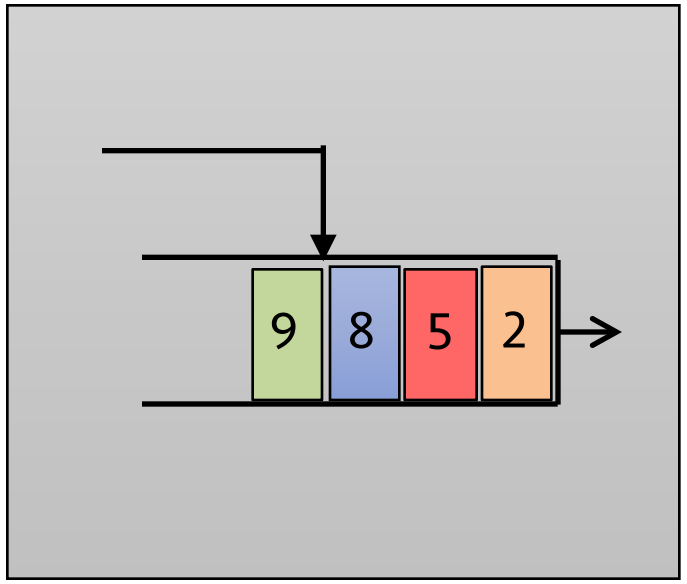
To program the scheduler, program the rank computation

Rank Computation



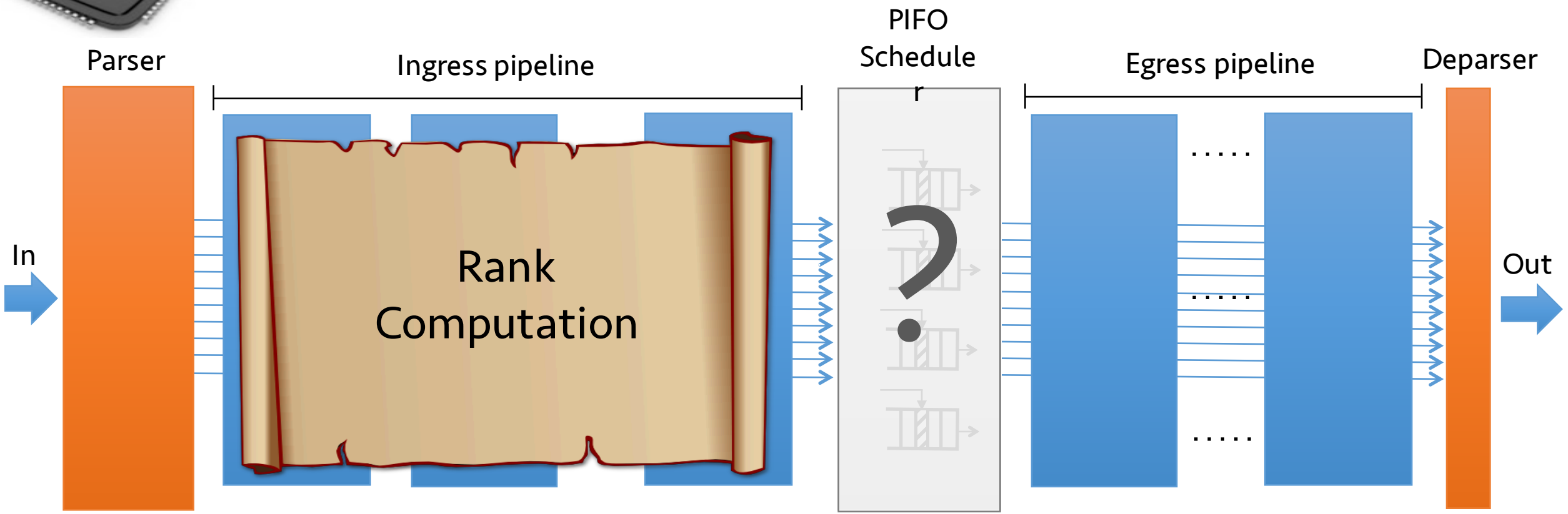
(programmable)

PIFO Scheduler

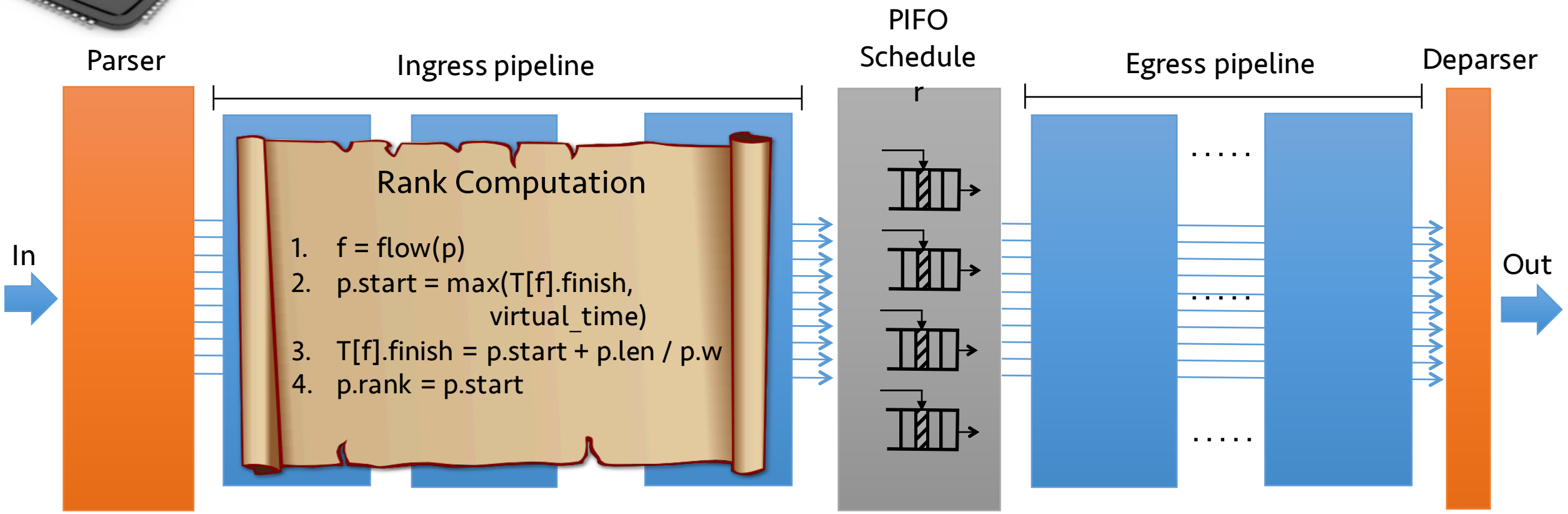


(fixed logic)

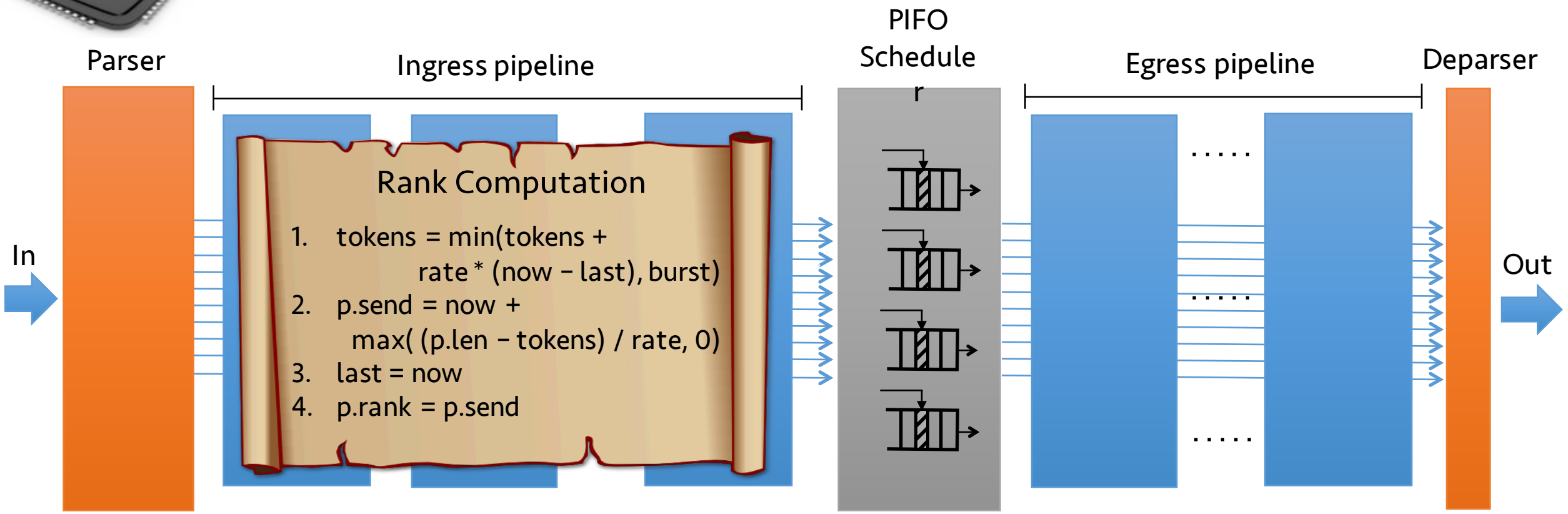
A programmable scheduler



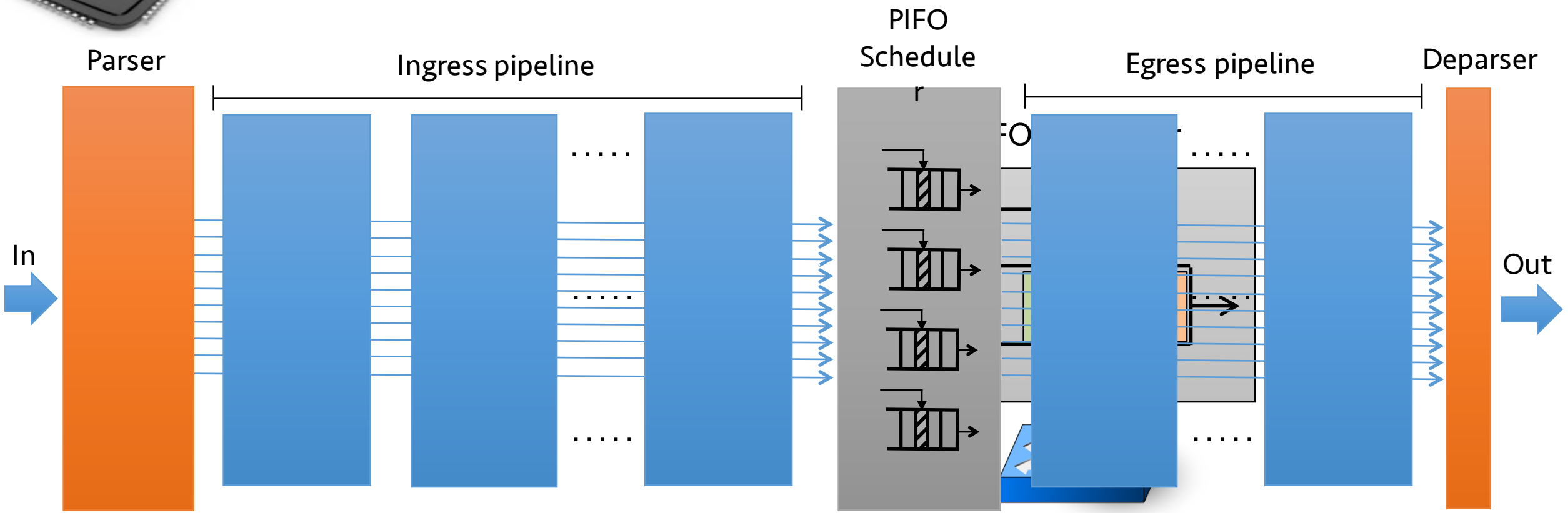
Weighted Fair Queuing



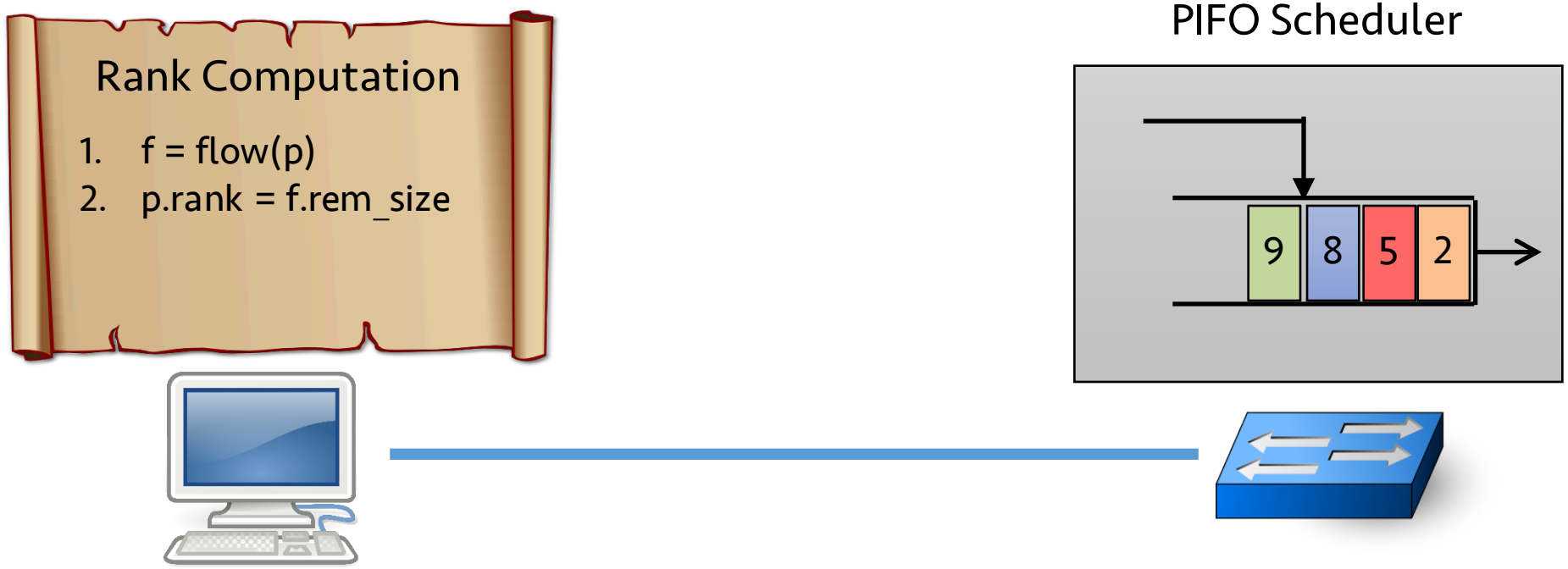
Traffic Shaping



pFabric (SRPT)

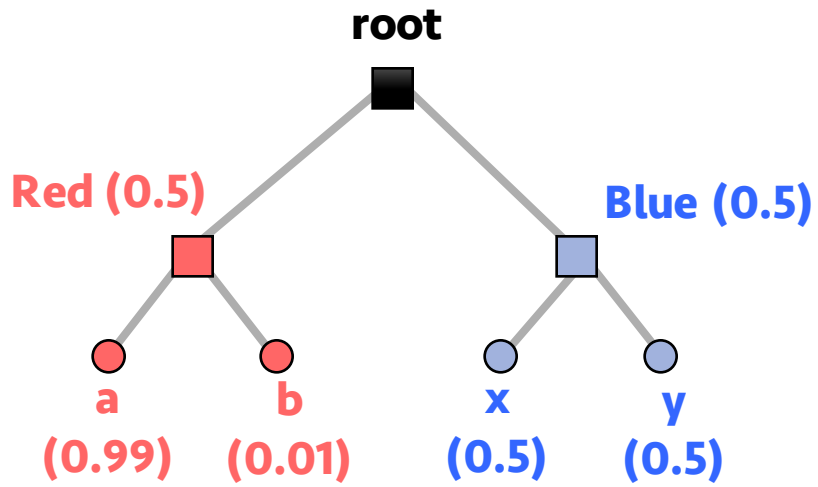


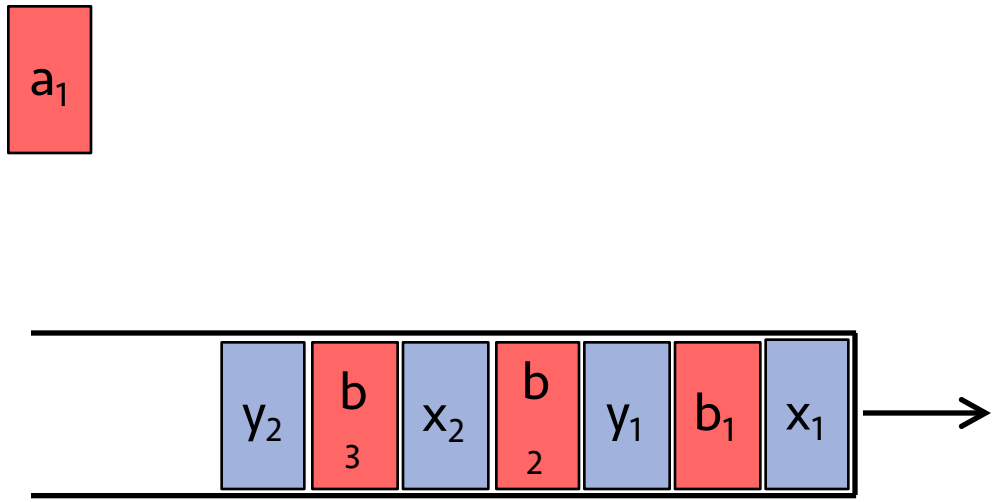
pFabric (SRPT)



Beyond a single PIFO

Hierarchical Packet Fair Queuing

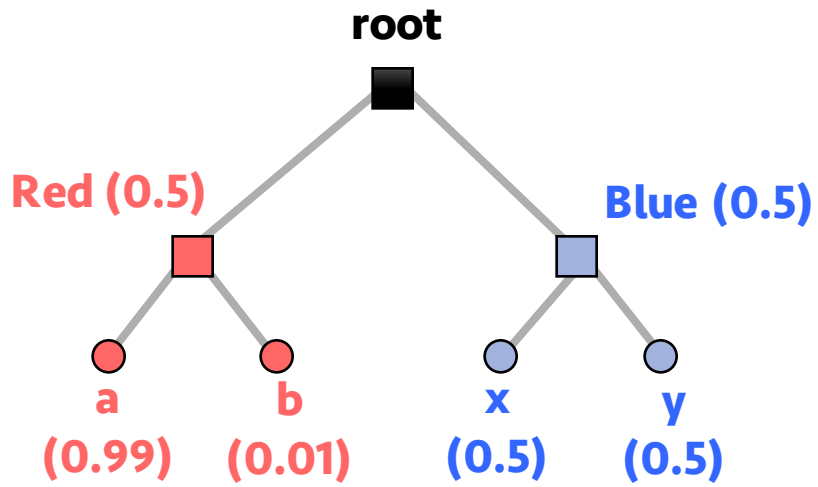


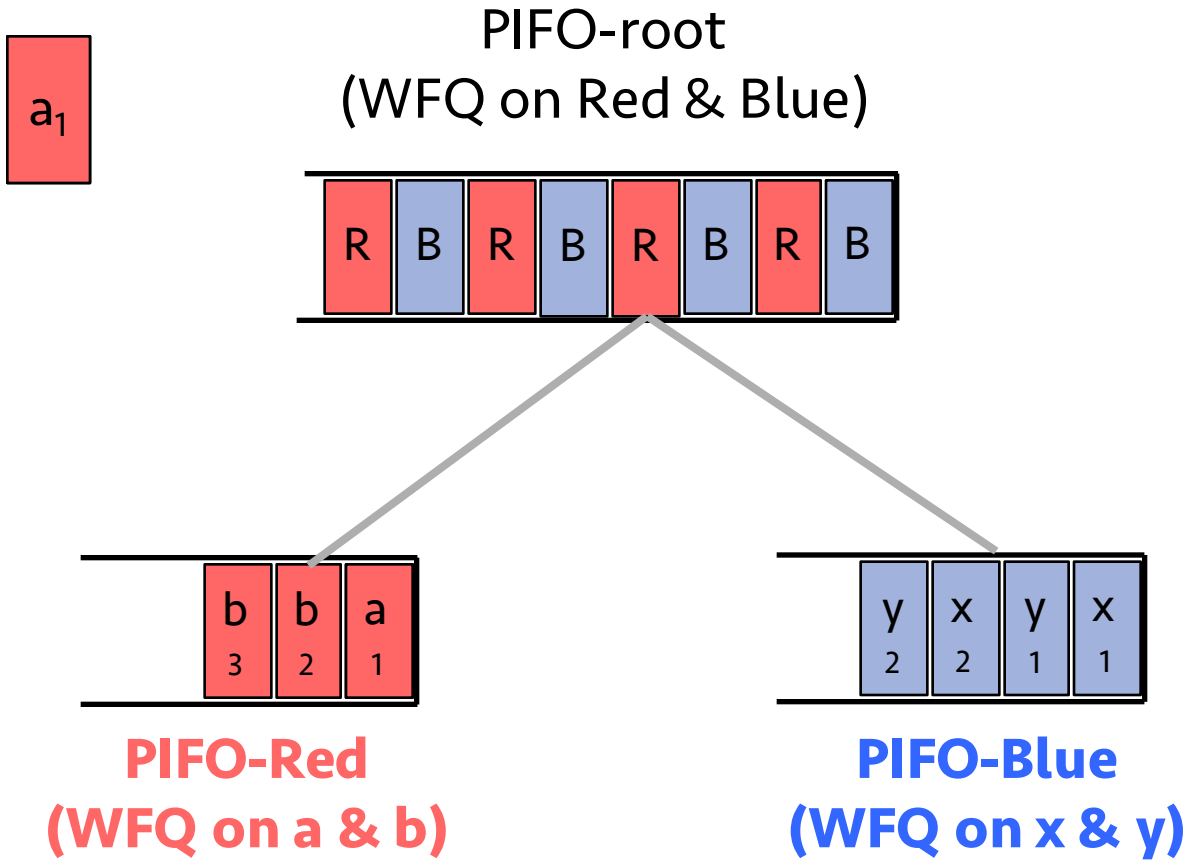


Hierarchical scheduling algorithms need hierarchy of PIFOs

Tree of PIFOs

Hierarchical Packet Fair Queuing





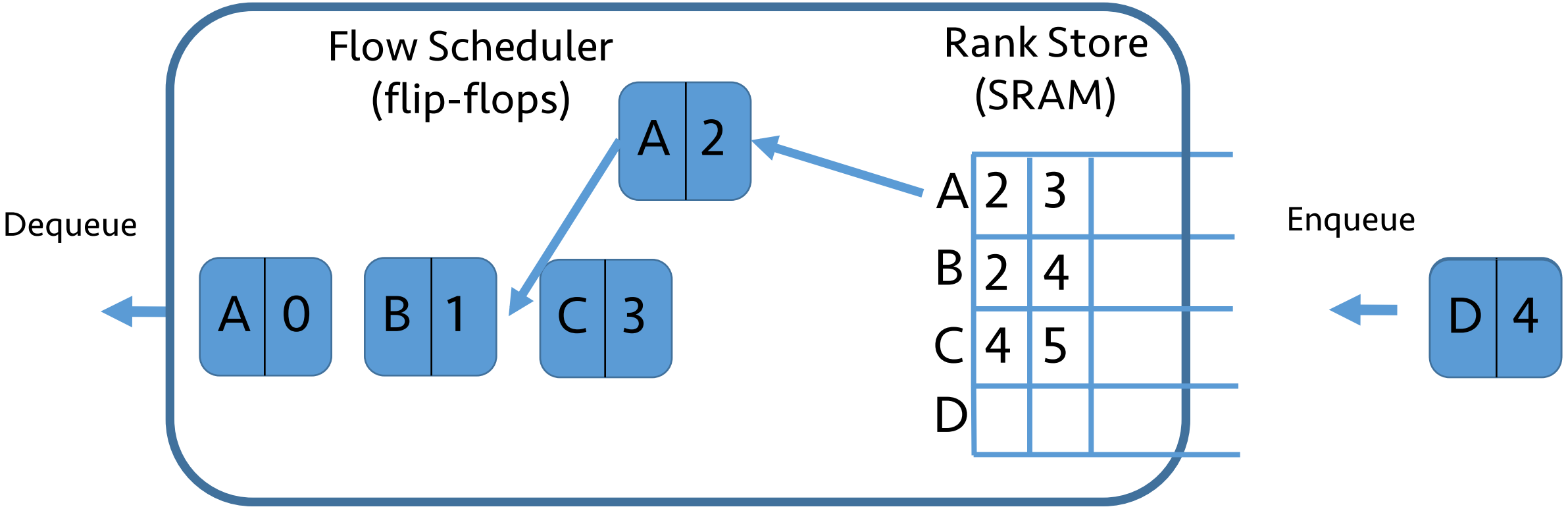
Expressiveness of PIFOs

- Fine-grained priorities: shortest-flow first, earliest deadline first
- 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

PIFO in hardware

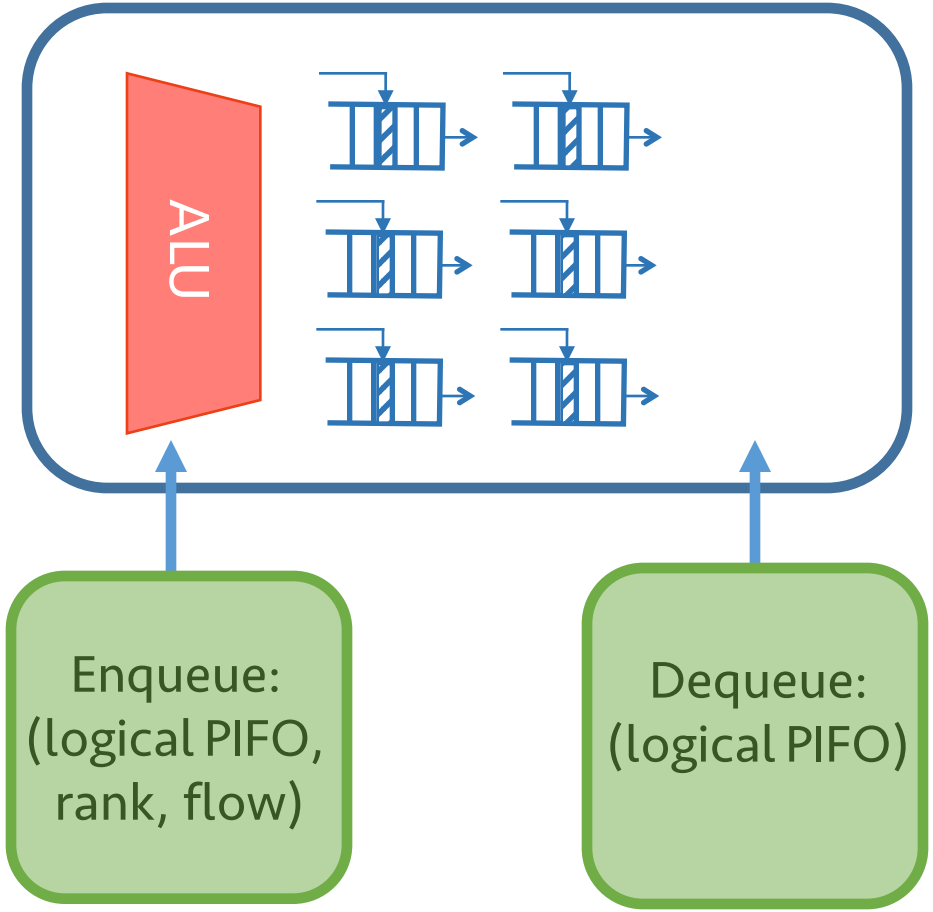
- Performance requirements, based on standard single-chip shared-memory switch (e.g., Broadcom Trident)
  - 1 GHz pipeline
  - 1K flows/physical queues
  - 60K packets (12 MB packet buffer, 200 byte cell)
- Naive solution: flat, sorted array, doesn't scale
- Scalable solution: use fact that ranks increase within a flow

A PIFO block

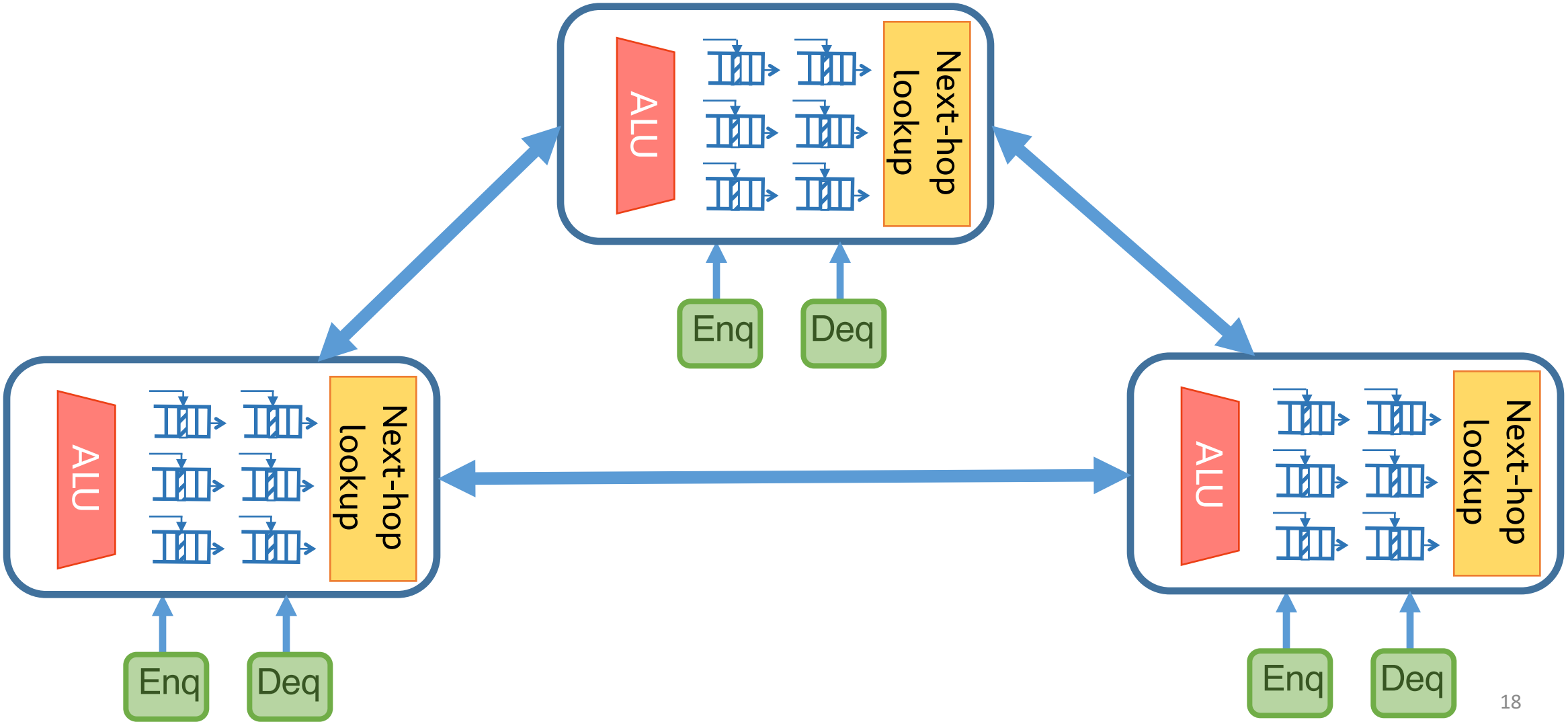


- 1 enqueue + 1 dequeue per clock cycle
- Can be shared among multiple logical PIFOs

A PIFO block



A PIFO mesh


ALU
Next-hop lookup
Enq
Deq
ALU
Next-hop lookup
Enq
Deq
ALU
Next-hop lookup
Enq
Deq

Hardware feasibility

- The Rank store is just a bank of FIFOs (stable hardware IP)
- Flow scheduler for 60K packets, 1K flows meets timing at 1GHz on a 16-nm transistor library
  - Continues to meet timing until 2048 flows, fails timing at 4096.
- E.g., 4% area overhead to program 5-level hierarchies (5-block PIFO mesh)

Summary

- PIFO is a promising abstraction for packet scheduling
  - Can express a wide range of algorithms
  - Can be implemented at line rate with modest overhead
- Next steps
  - PIFO-capable switching chips
  - Language support
- Would love feedback

Proposal: scheduling in P4

- Currently not modeled at all, blackbox left to vendor
- Only part of the switch that isn't programmable
- PIFOs present a candidate
- Concurrent work on Universal Packet Scheduling also requires a priority queue that is identical to a PIFO

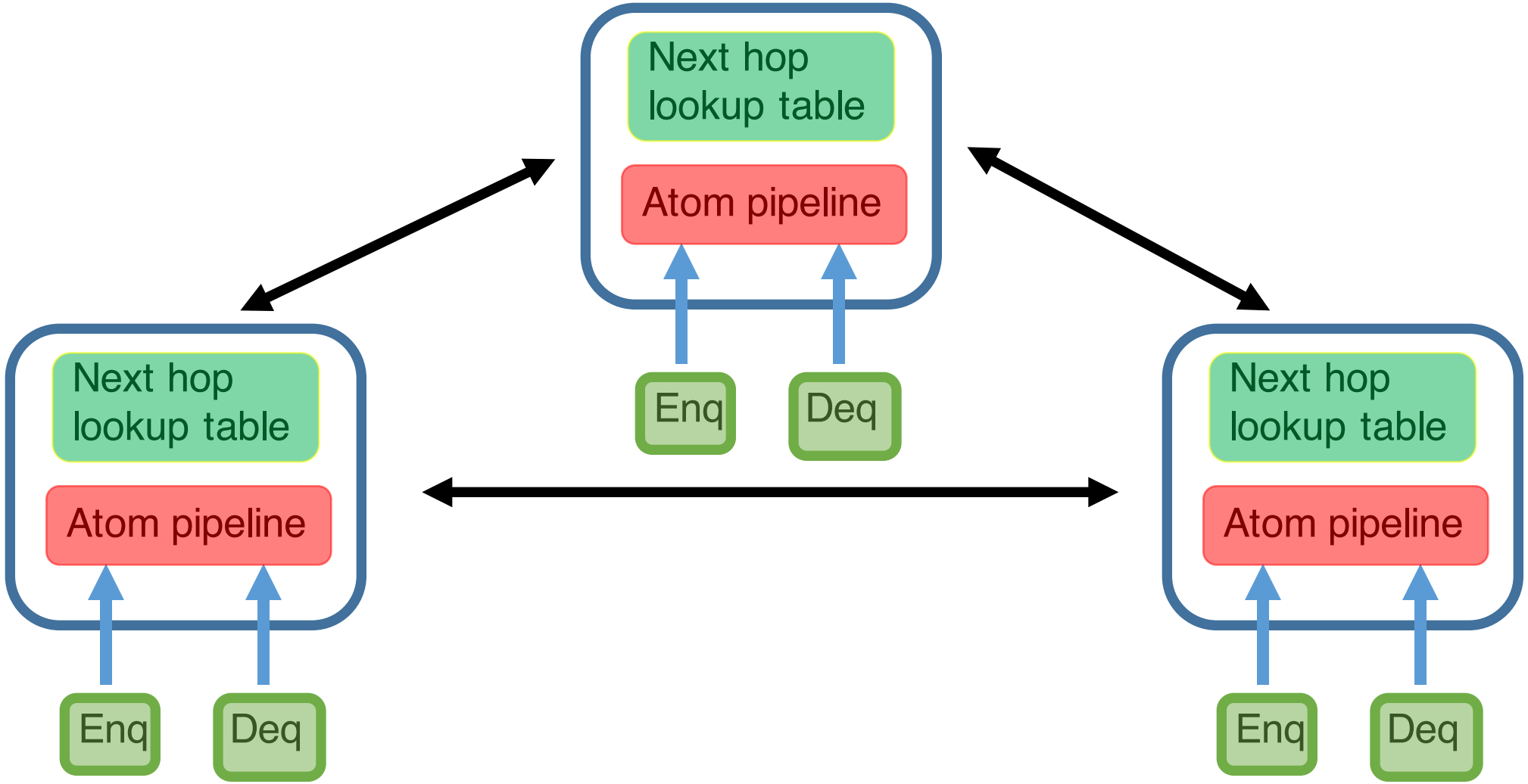
Proposal: scheduling in P4

- Need to model a PIFO (or priority queue) in P4

- Requires an extern instance to model a PIFO
  - Can start by including it in a target-specific library
  - Later migrate to standard library if there's sufficient interest
  - Section 16 of P4v1.1

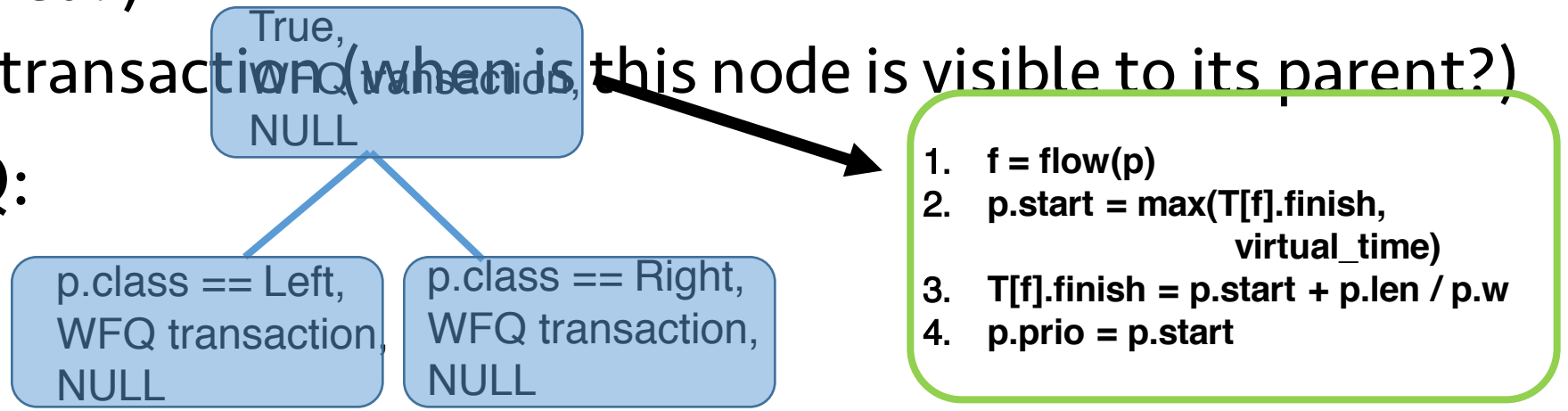
- Transactions themselves can be compiled down to P4 code using the Domino DSL for stateful algorithms.

A PIFO mesh


Next hop lookup table
Atom pipeline
Enq
Deq
Next hop lookup table
Atom pipeline
Enq
Deq
Next hop lookup table
Atom pipeline
Enq
Deq

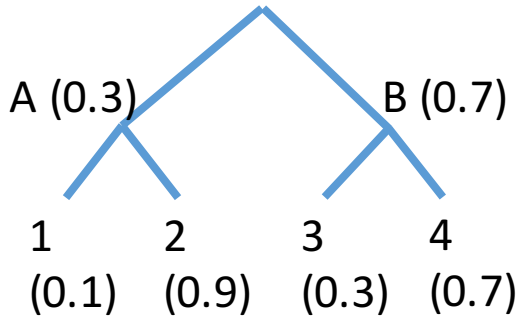
Language for programmable scheduling

- Tree of scheduling nodes
- Each node has:
  - Predicate (which packets belong to this node?)
  - Scheduling transaction (how are packet/class priorities determined?)
  - Shaping transaction (when is this node is visible to its parent?)
- E.g., HPFQ:

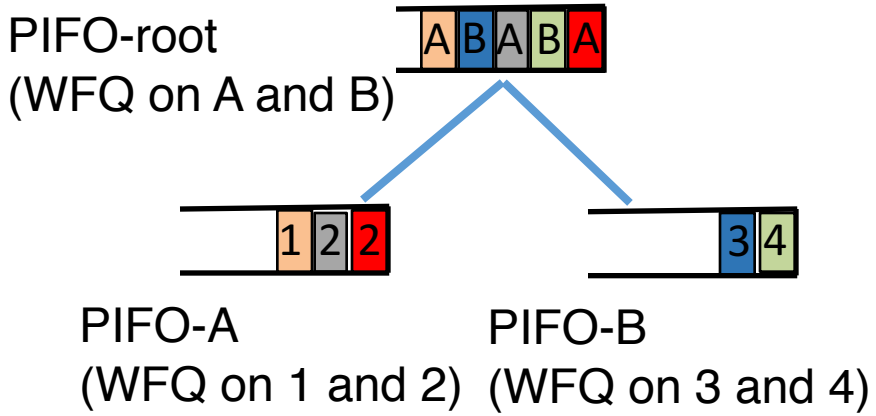


Composing PIFOs

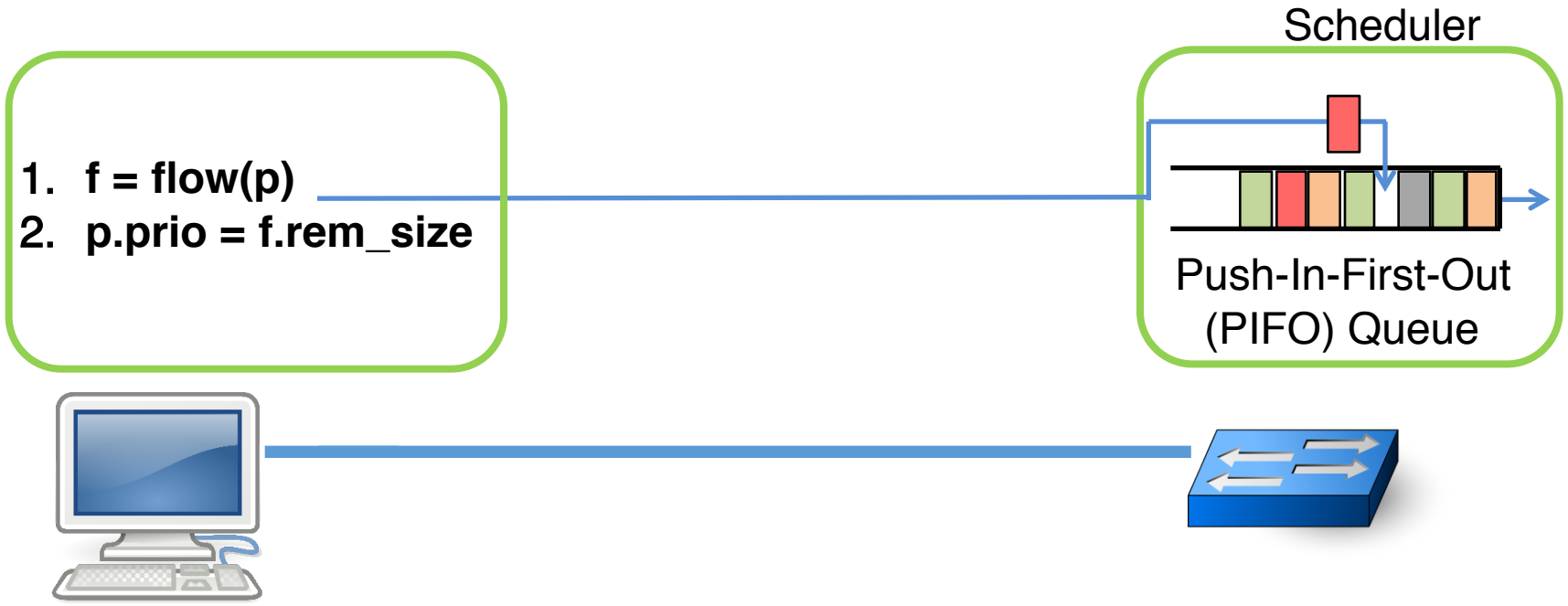
Hierarchical packet-fair queueing (HPFQ)



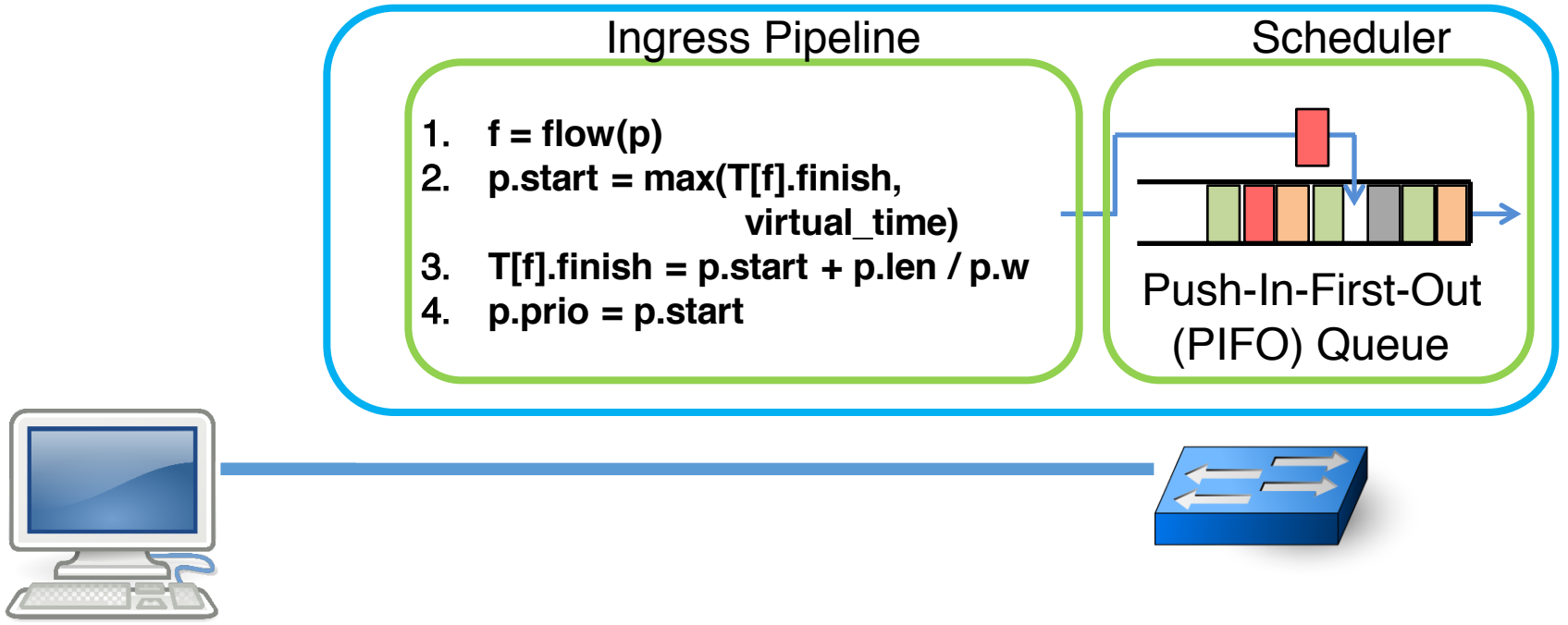
Composing PIFOs



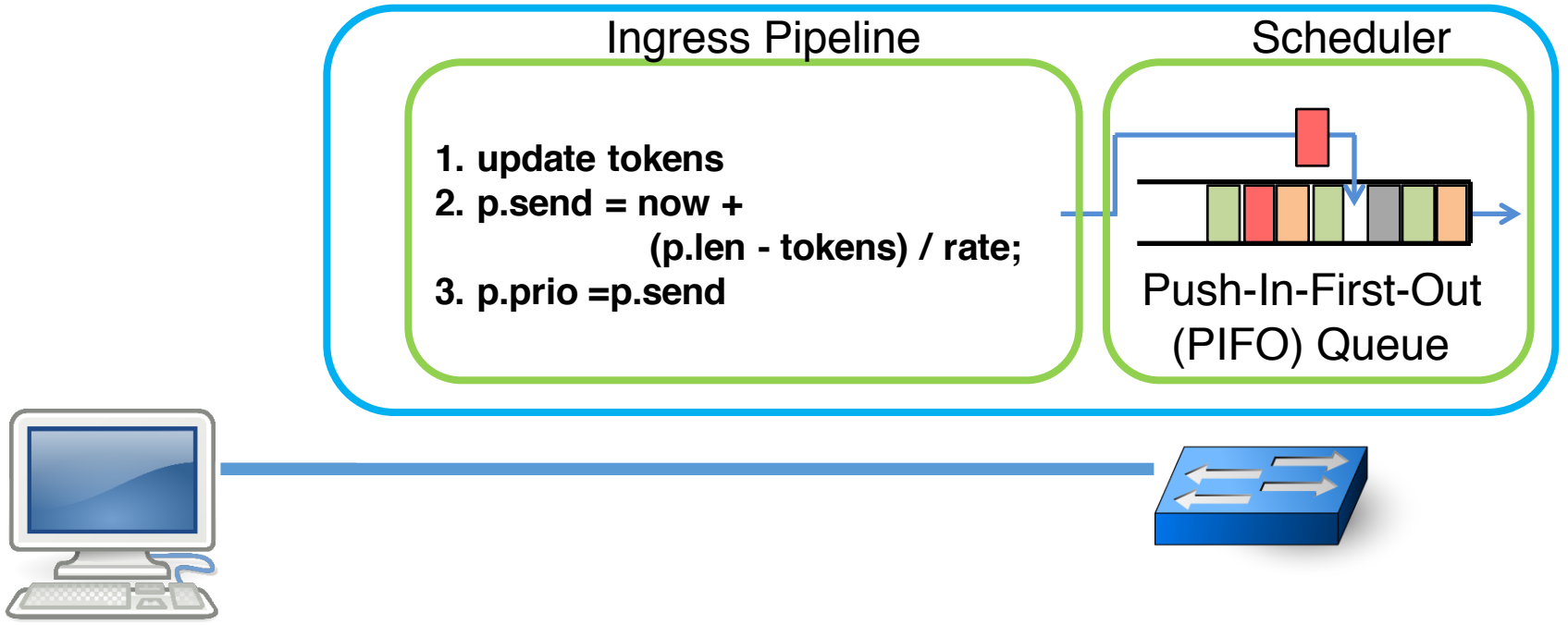
Shortest remaining processing time



Weighted fair queuing

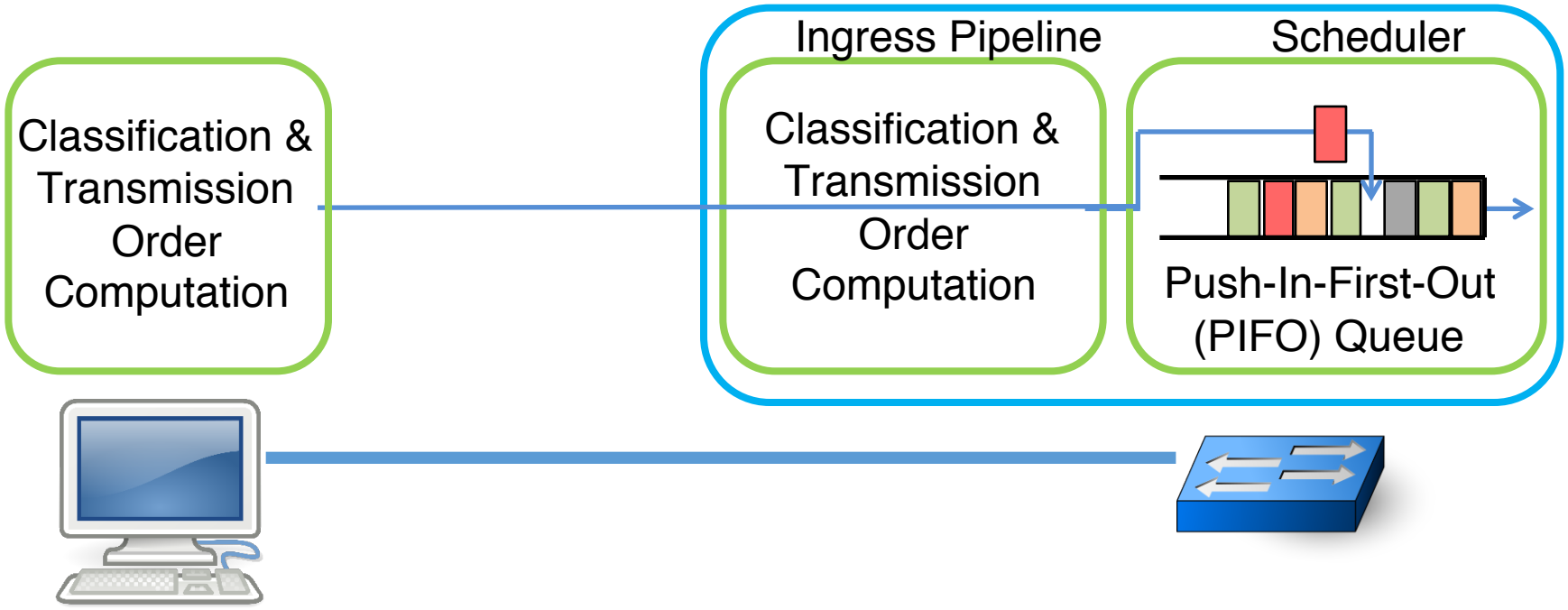


Traffic Shaping



A programmable scheduler

Key idea: separate priority computation from enforcement



The Push-In First-Out Queue

- In many algorithms, relative scheduling order of enqueued packets doesn't change with future arrivals
- i.e., we can determine scheduling order at packet arrival
- Examples:
  - SJF: Order determined by flow size
  - FCFS: Order determined by arrival time
- Push-in first-out queues (PIFO): packets are pushed into an arbitrary location based on a priority, and dequeued from the head
- First used as a proof construct by Chuang et. al

What does the scheduler do?

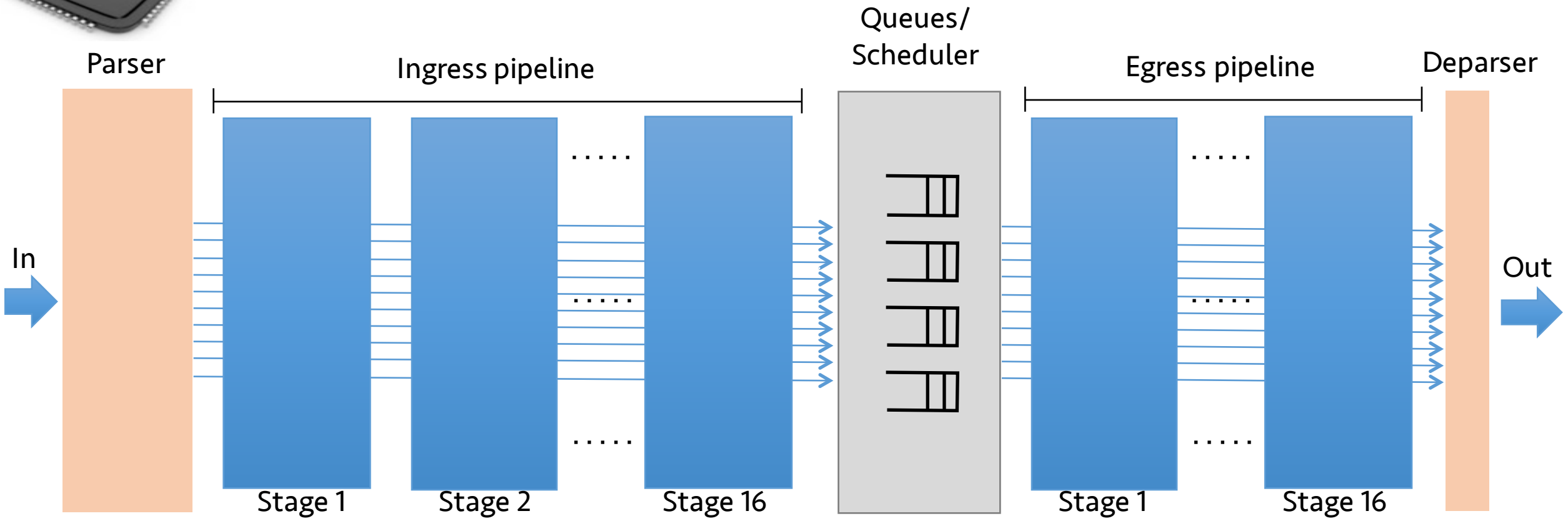
- It decides
  - In what **order** are packets sent
  - At what **time** are packets sent

- Key observation
  - In many algorithms, the packet scheduling order/time does not change with future arrivals
  - i.e., we can determine scheduling order before enqueue

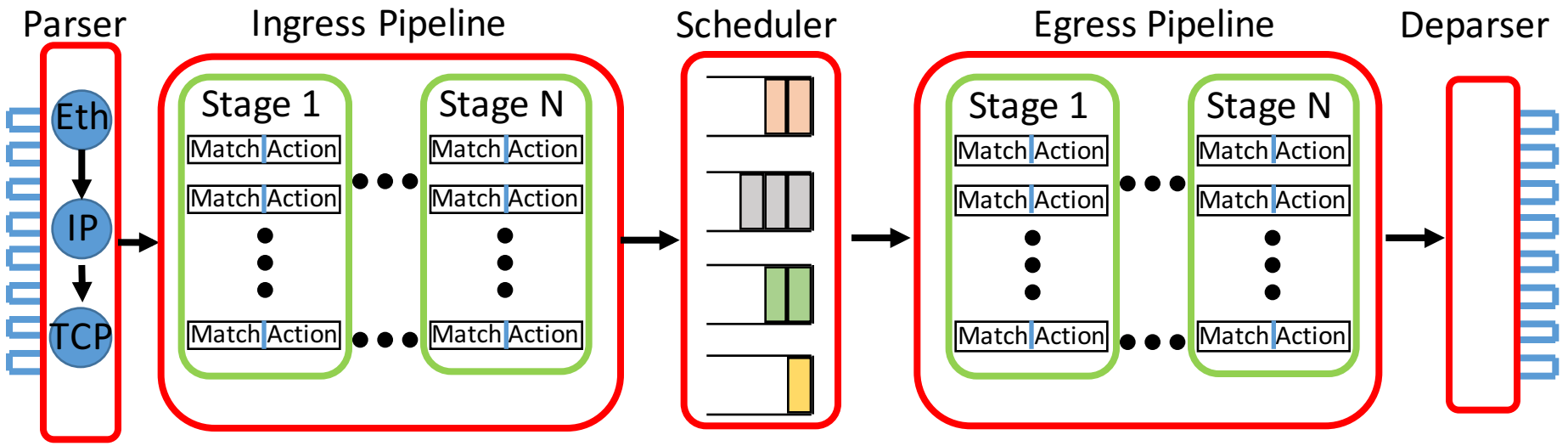
Why is programmable scheduling hard?

- Plenty of scheduling algorithms
- Yet, no consensus on the right abstractions for scheduling
- In contrast to
  - Parse graphs for parsing
  - Match-action tables for forwarding
- On the surface, packet transactions are insufficient

Programmable switching chips



Today: Packet scheduling is off-limits for programing



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

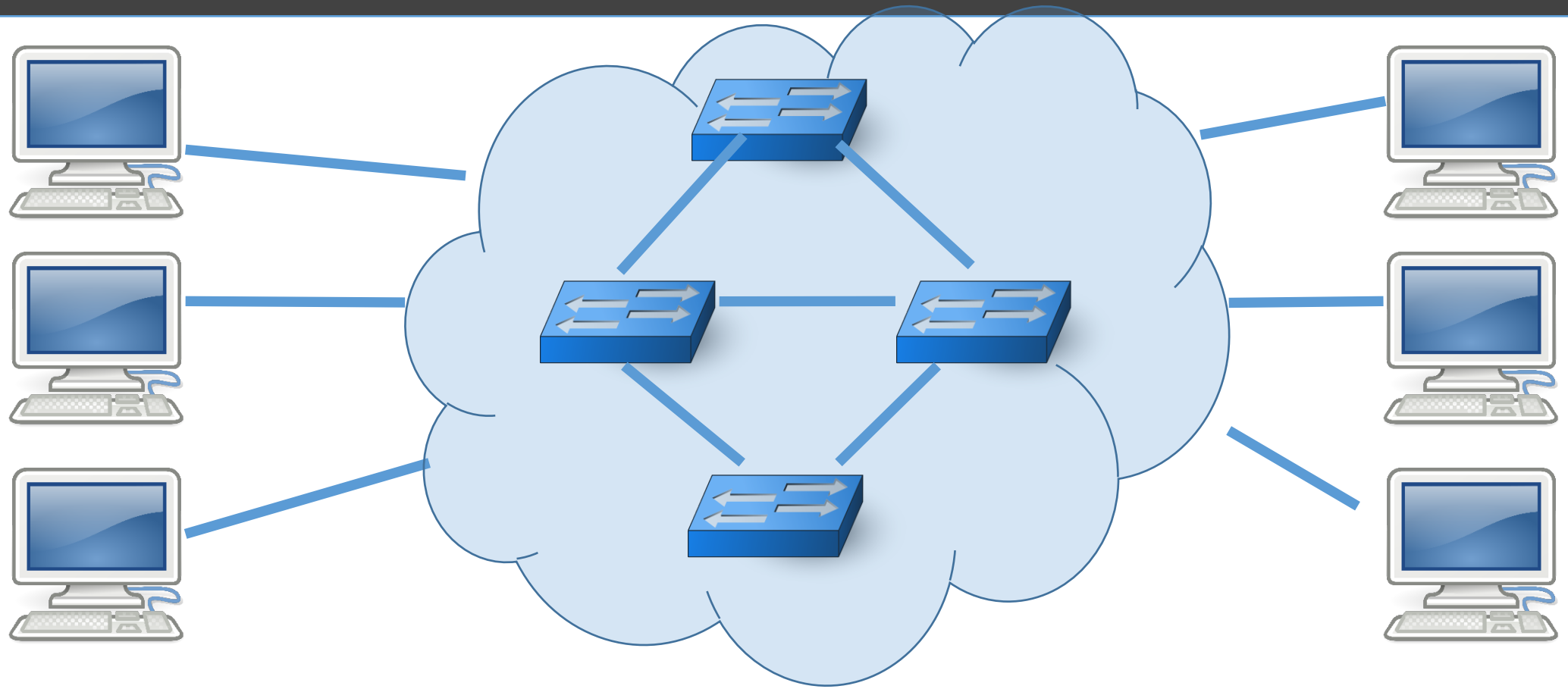
Looking forward

- The end of Moore's law => specialized hardware

- The solution (for networking hardware): high-performance abstractions for programming specific router functionality
  - Stateful algorithms: Packet transactions, atoms
  - Scheduling: PIFOs
  - Network diagnostics/measurement: ?

- Preprints of papers appearing at SIGCOMM 2016:
  - http://arxiv.org/abs/1512.05023 (Packet transactions)
  - http://arxiv.org/abs/1602.06045 (PIFOs)

Traditional networking



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

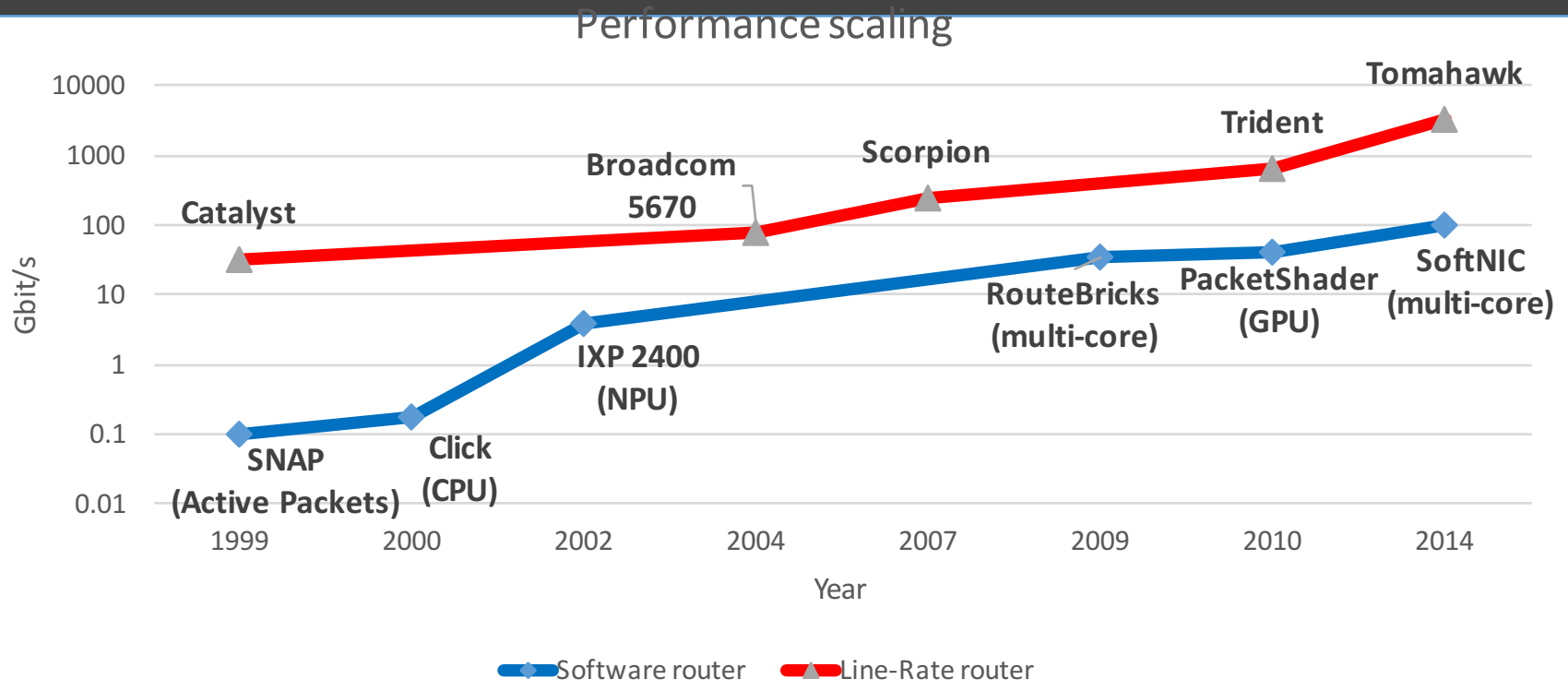
Why is the traditional view insufficient?

- Router features tied to ASIC design cycles (2-3 years)
  - Long lag time for new protocol formats (IPv6, VXLAN)
- Operators (especially in datacenters) need greater control
  - Access control, load balancing, bandwidth sharing, measurement
- Many proposals never make it to production
- Ideally, we would have a programmable router

The quest for programmable routers

- Early routers (late 60s to mid 90s) built out of commodity CPUs
  - IMPs (1969): Honeywell DDP-516
  - Fuzzball (1971): DEC LSI-11
  - Stanford multiprotocol router (1981): DEC PDP 11
  - Proteon / MIT C gateway (1980s): DEC MicroVAX II

The quest for programmable routers



- 10—100 x loss in performance relative to line-rate, fixed-function routers
- Unpredictable performance (e.g., cache contention)

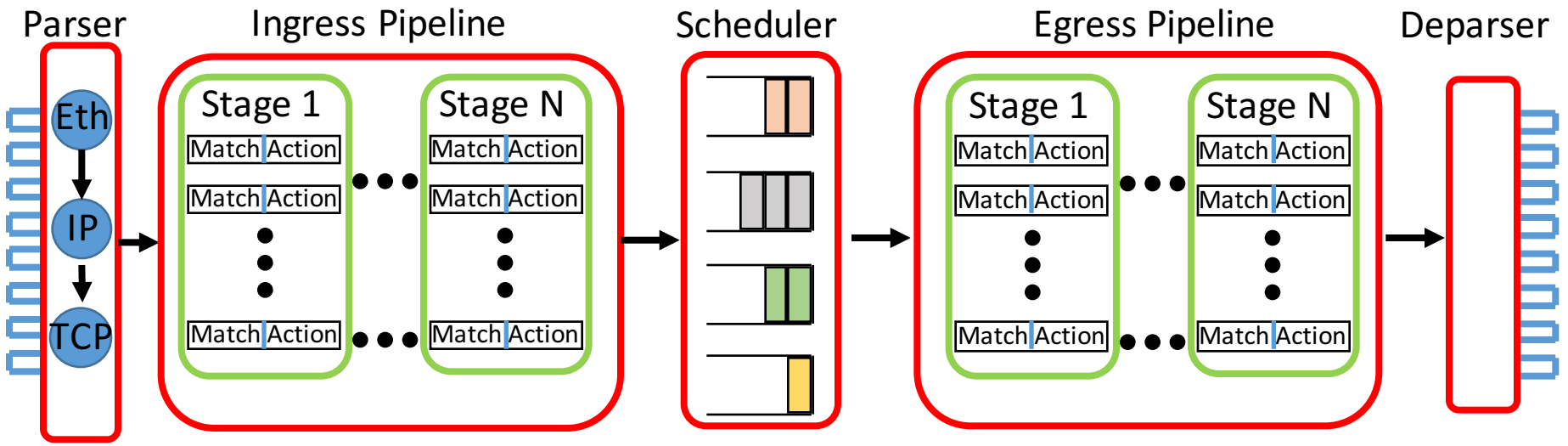
The vision: programmability at line-rate

- Performance and predictability of hardware, line-rate routers

- More programmable than fixed-function routers
  - Much more than the current OpenFlow/SDN APIs for routers
  - …, but less than software routers

- Chipsets emerging around this paradigm: RMT, FlexPipe, Xpliant
  - Moore's law has reduced area overhead for programmability

My work



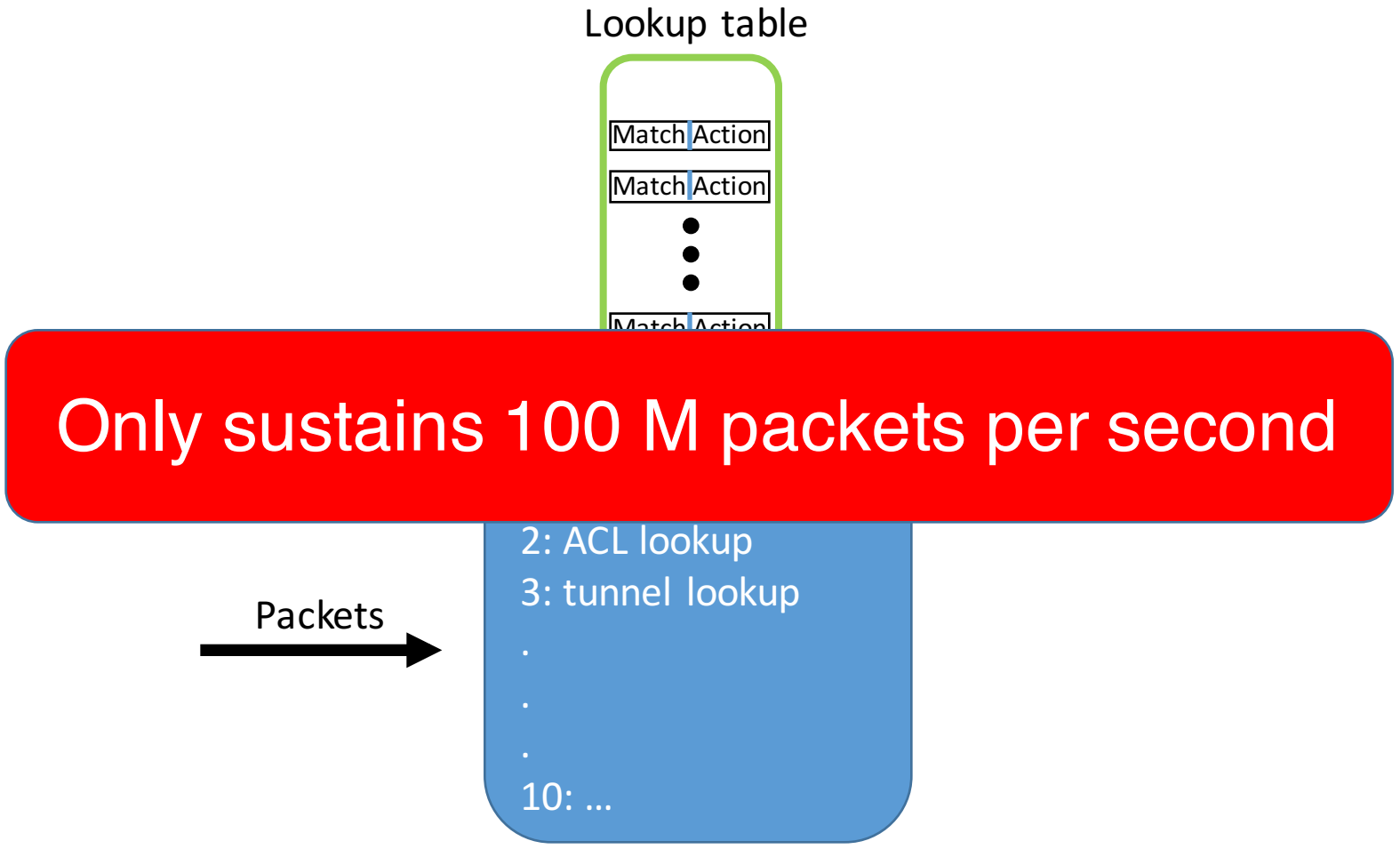
- The machine model: Formalizing the computational capabilities of line-rate routers
- Packet transactions: High-level programming for the router pipeline
- Push-In First-Out Queues: Programming the scheduler

Performance requirements at line-rate

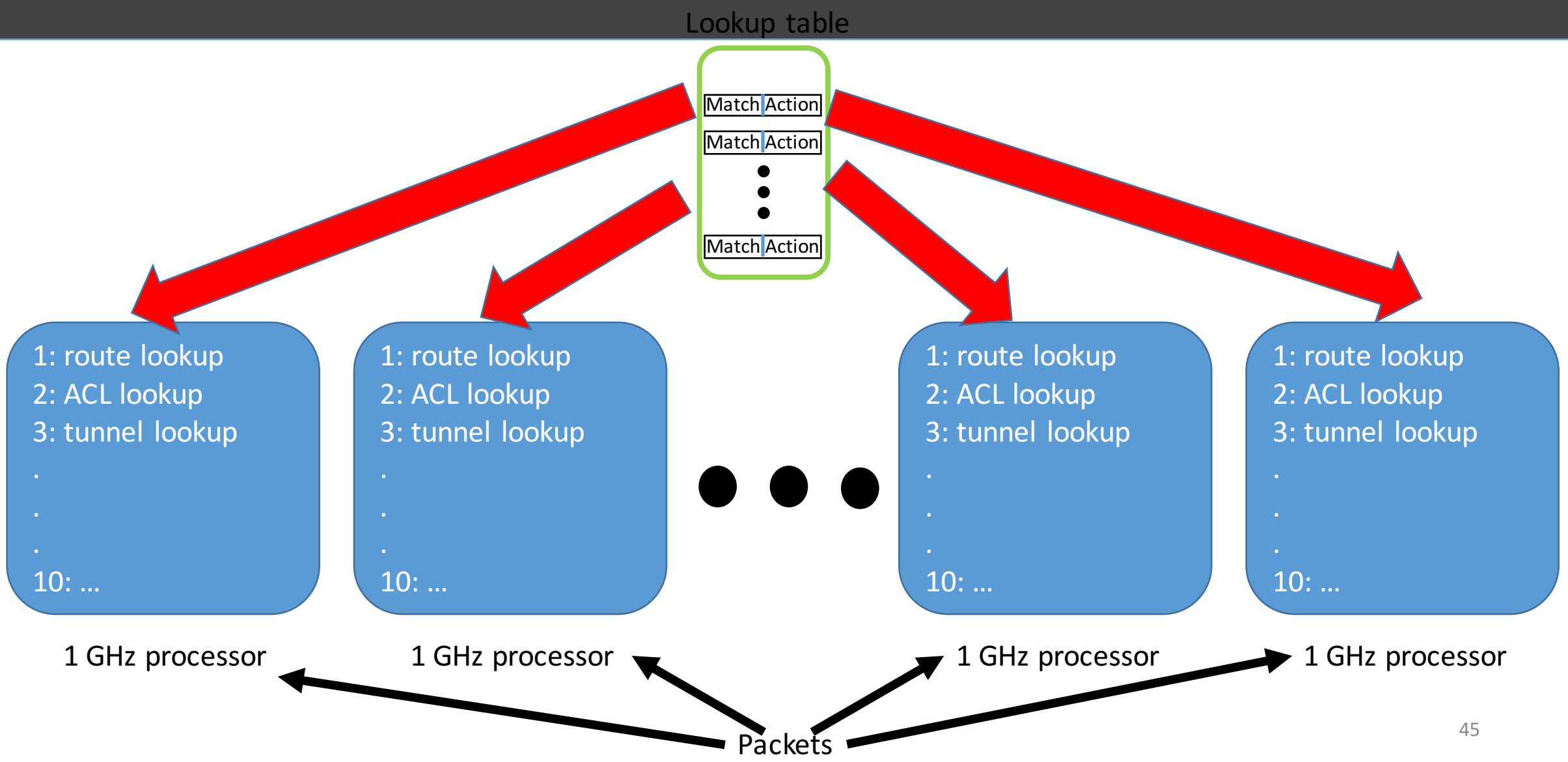
- Aggregate capacity ~ 1 Tbit/s
- Packet size ~ 1000 bits
- ~10 operations per packet (e.g., routing, ACL, tunnels)

Need to process 1 billion packets per second, 10 ops per packet

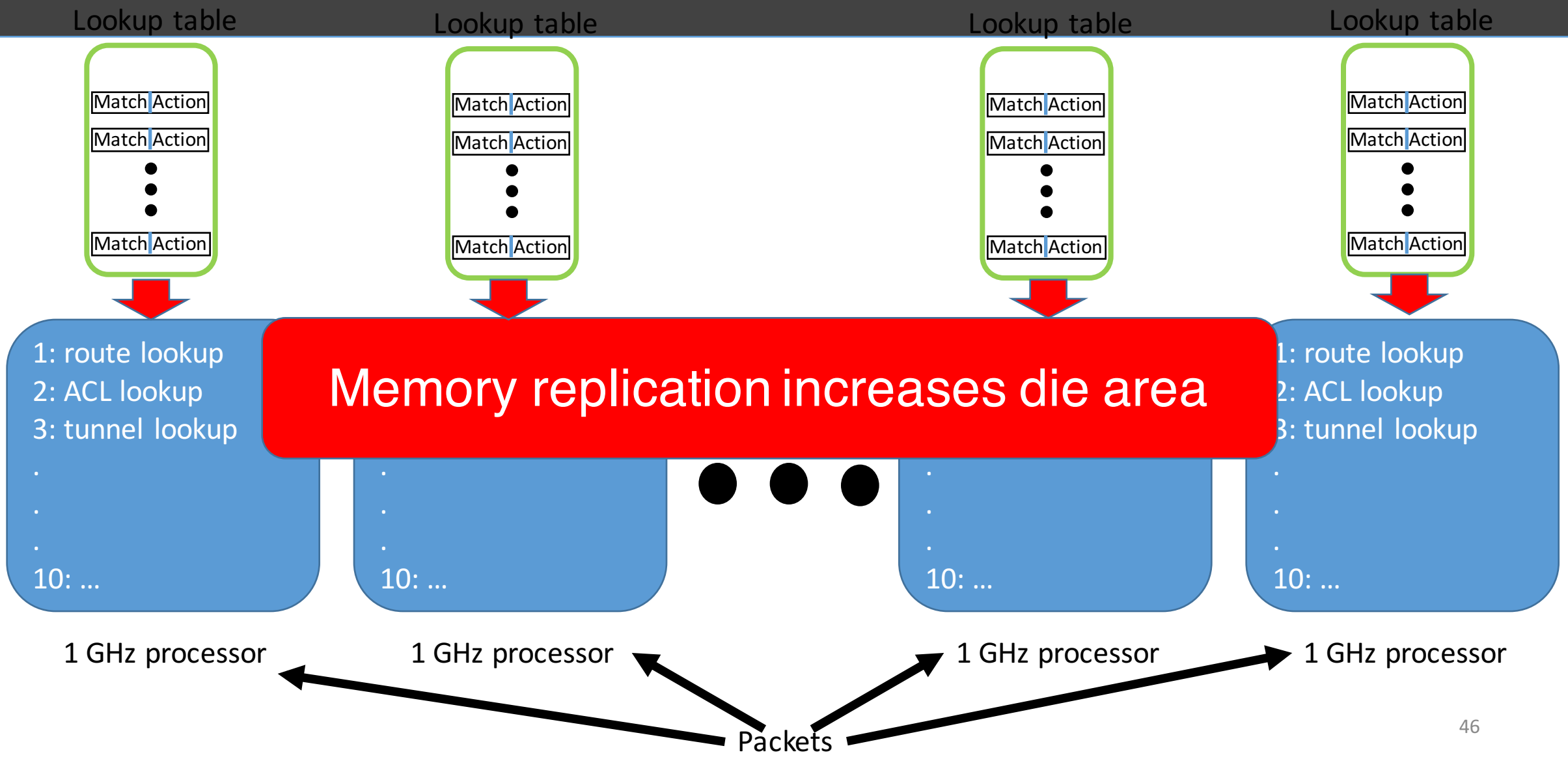
Single processor architecture



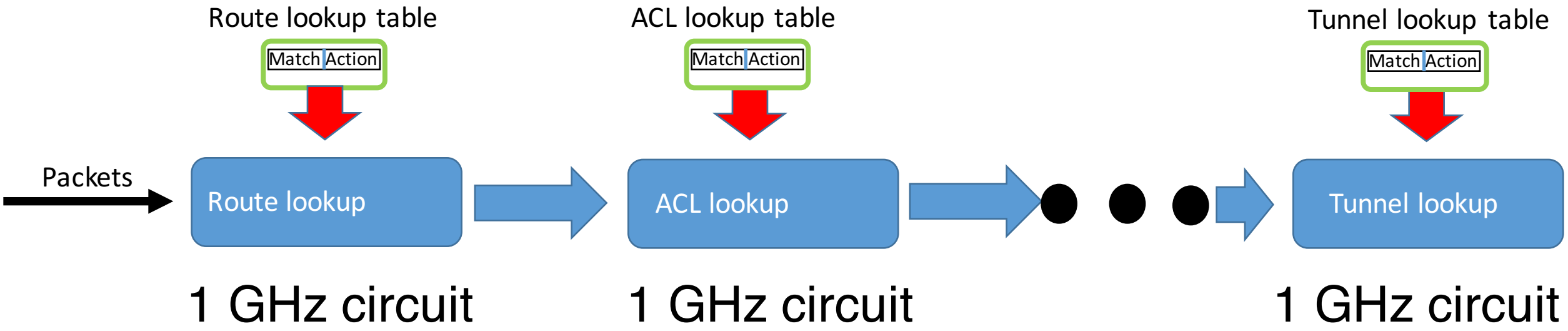
Packet-parallel architecture



Packet-parallel architecture

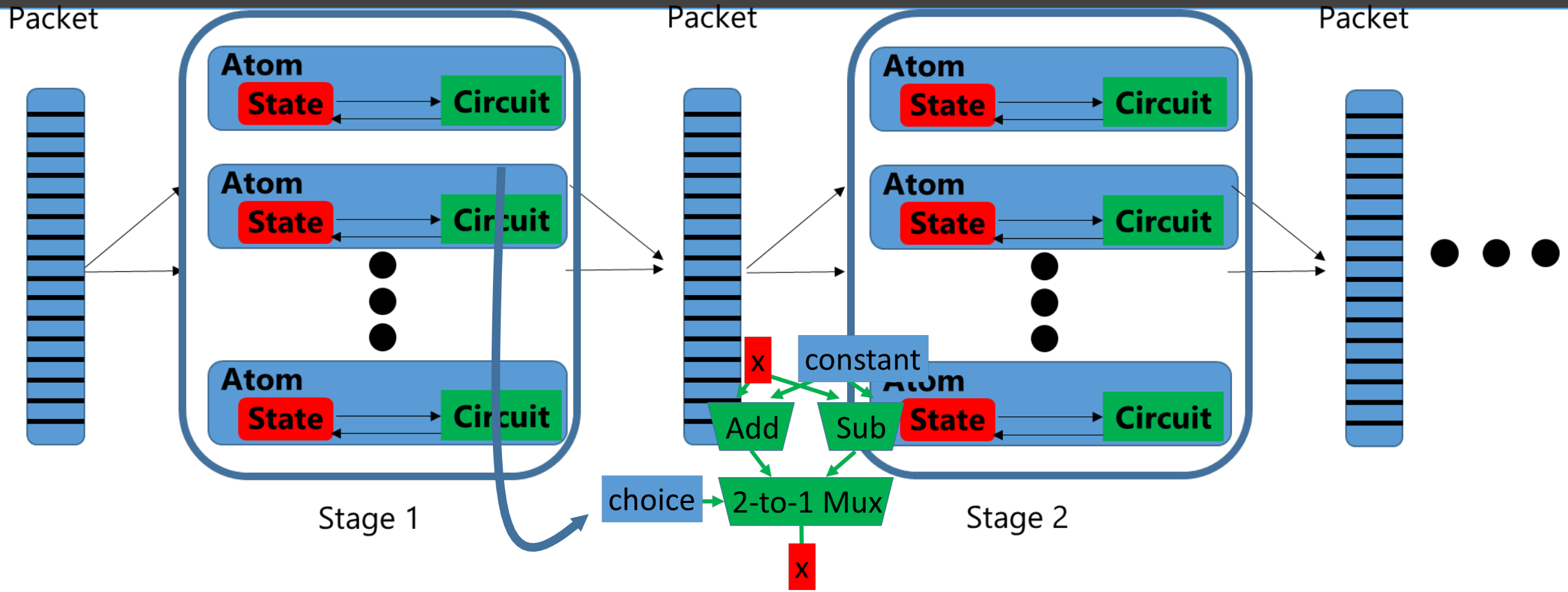


Function-parallel or pipelined architecture



- Factors out global state into per-stage local state
- Replaces full-blown processor with a circuit
- But, needs careful circuit design to run at 1 GHz

A machine model for line-rate routers



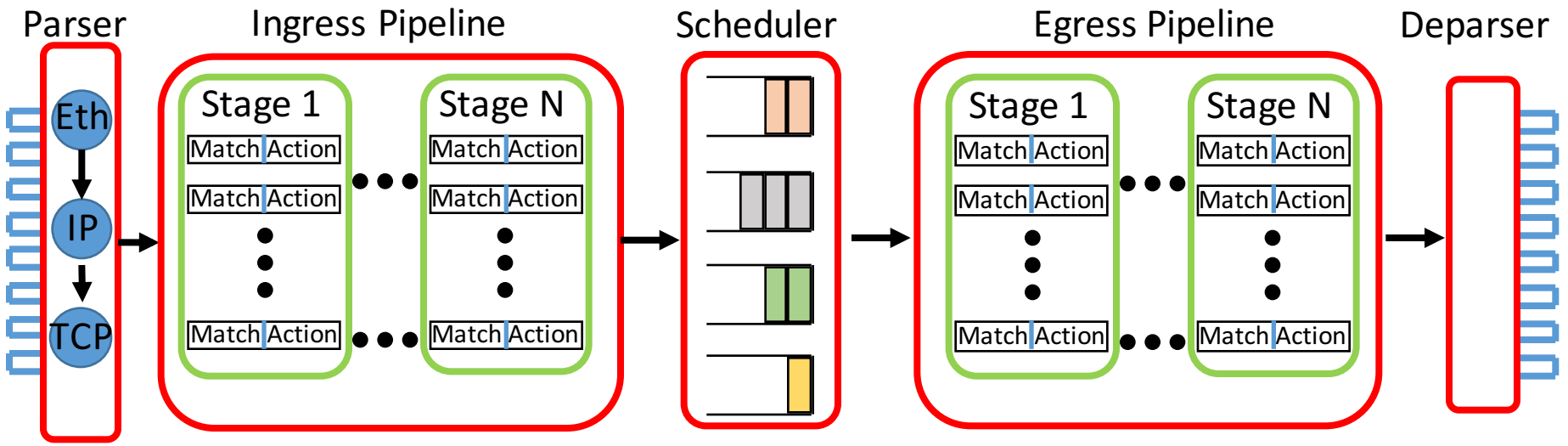
- Deterministic pipeline
- Atoms: Smallest unit of atomic packet processing / state update
- A router's atoms constitute its instruction set

Stateless vs. stateful atoms

- Stateless operations
  - E.g., pkt.f4 = pkt.f1 + pkt.f2 – pkt.f3
  - Can be easily pipelined into two stages
  - Suffices to provide simple stateless atoms alone

- Stateful operations
  - E.g., x = x + 1
  - Cannot be pipelined; needs an atomic read+modify+write instruction
  - Explicitly design each stateful operation in hardware for atomicity

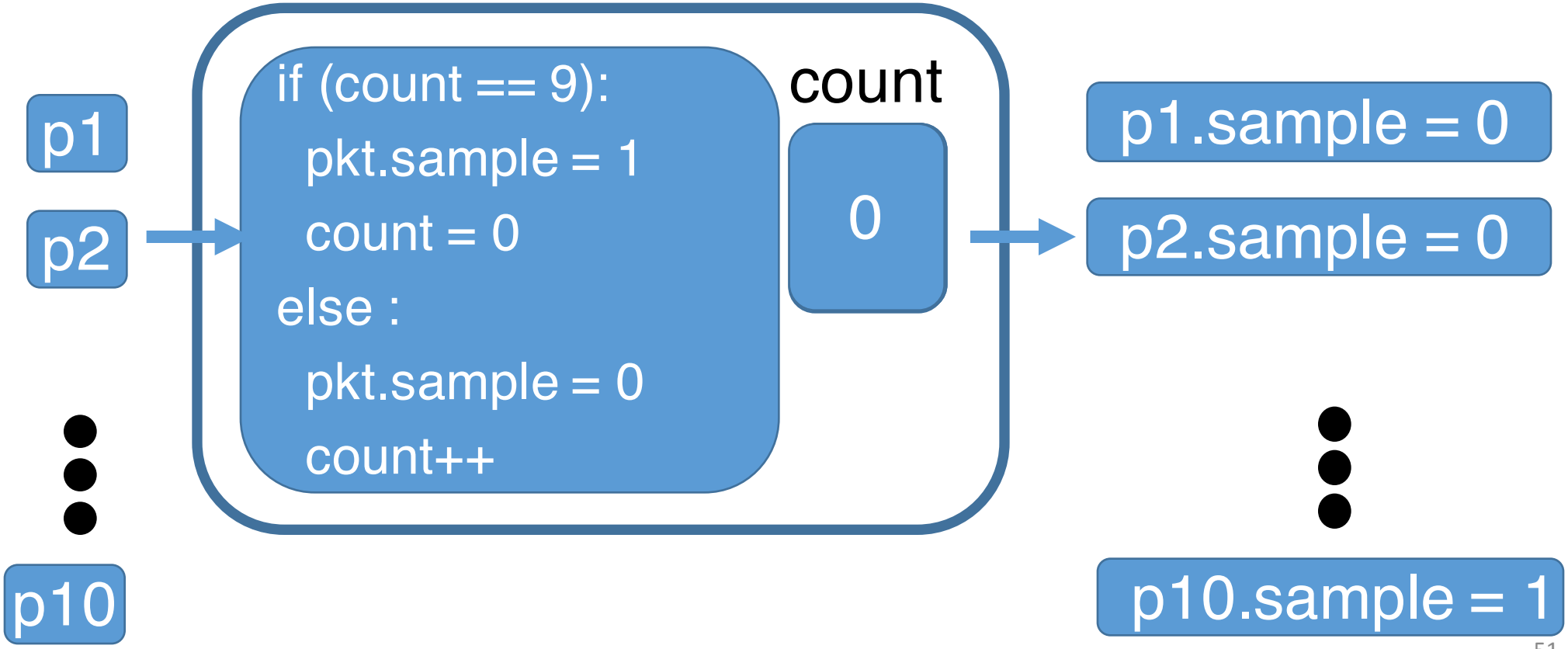
My work



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

Packet transactions

- Packet transaction: Block of imperative code
- A transaction runs to completion, processes one packet at a time, serially

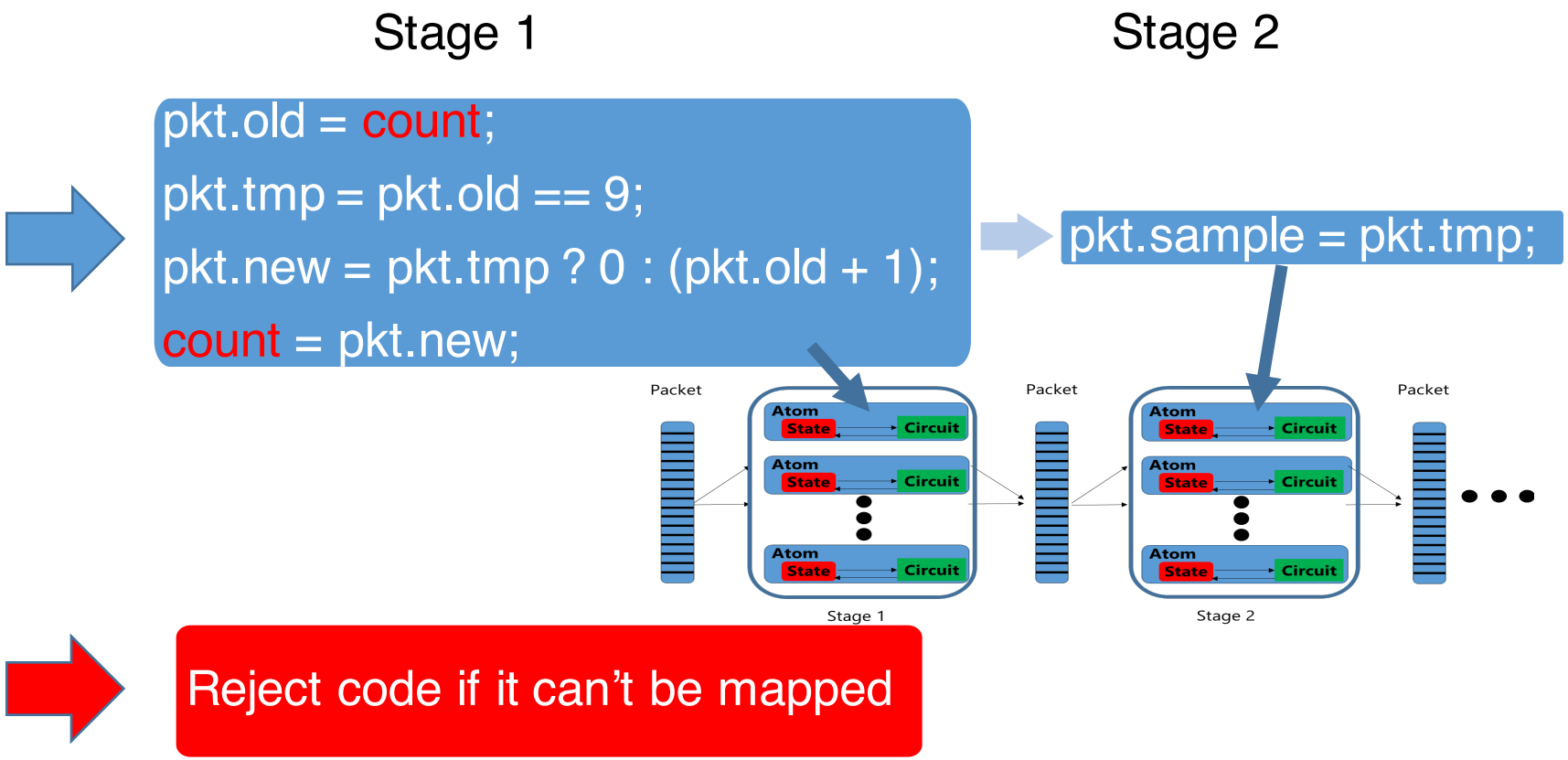


Programming with packet transactions

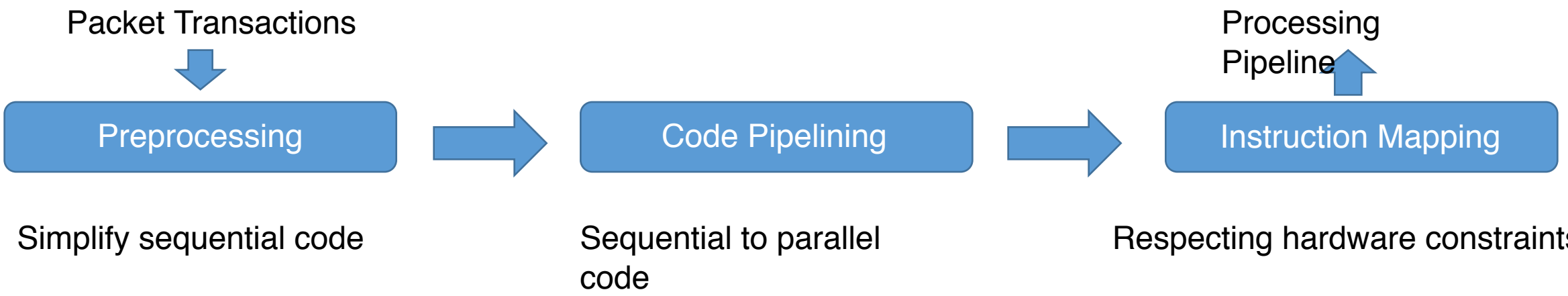
Packet sampling algorithm

```
if (count == 9):
  pkt.sample = 1
  count = 0
else :
  pkt.sample = 0
  count++
```

Packet sampling pipeline

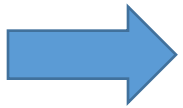


The compiler



Code after preprocessing

```
if (count == 9):
  pkt.sample = 1
  count = 0
else :
  pkt.sample = 0
  count++
```



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

Code Pipelining

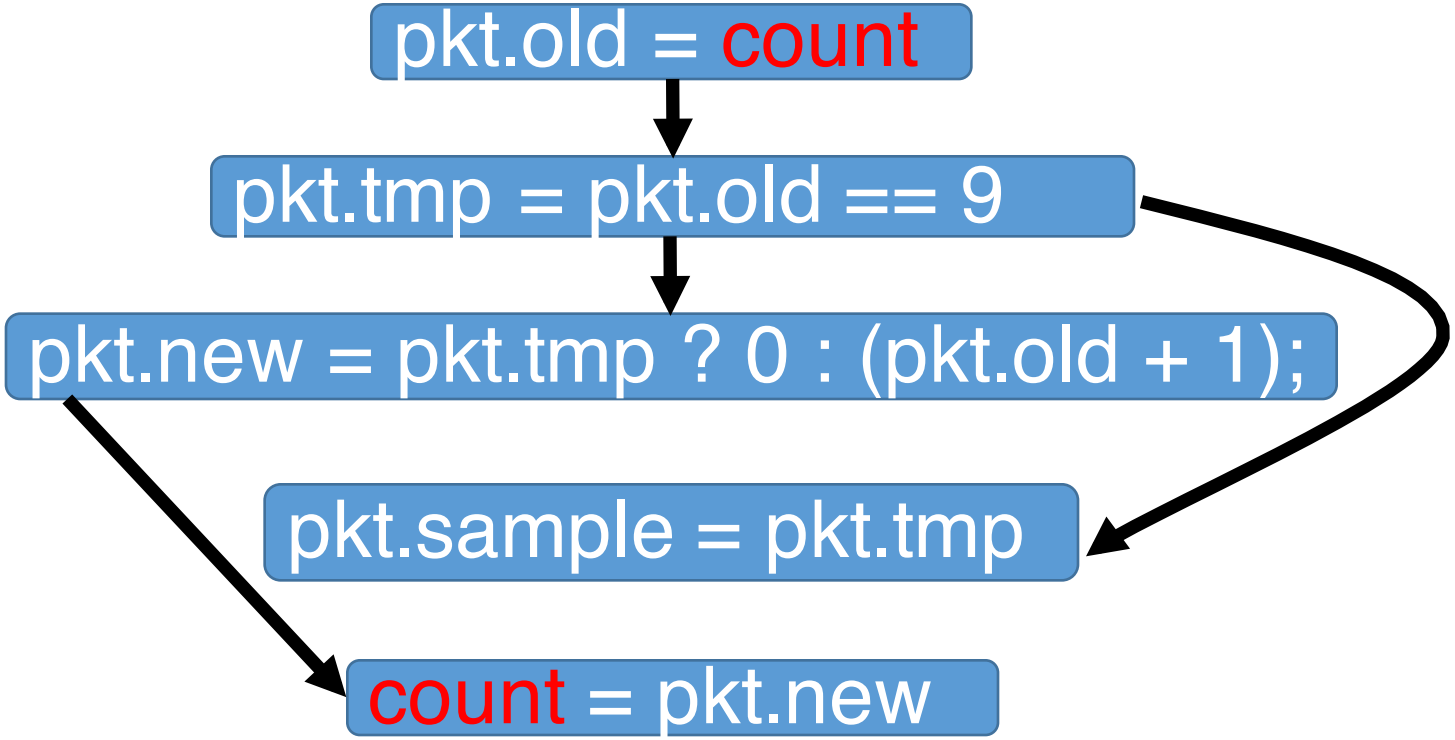
Preprocessing → Sequential to parallel code → Hardware constraints

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

Create one node for each instruction.

Preprocessing → Sequential to parallel code → Hardware constraints

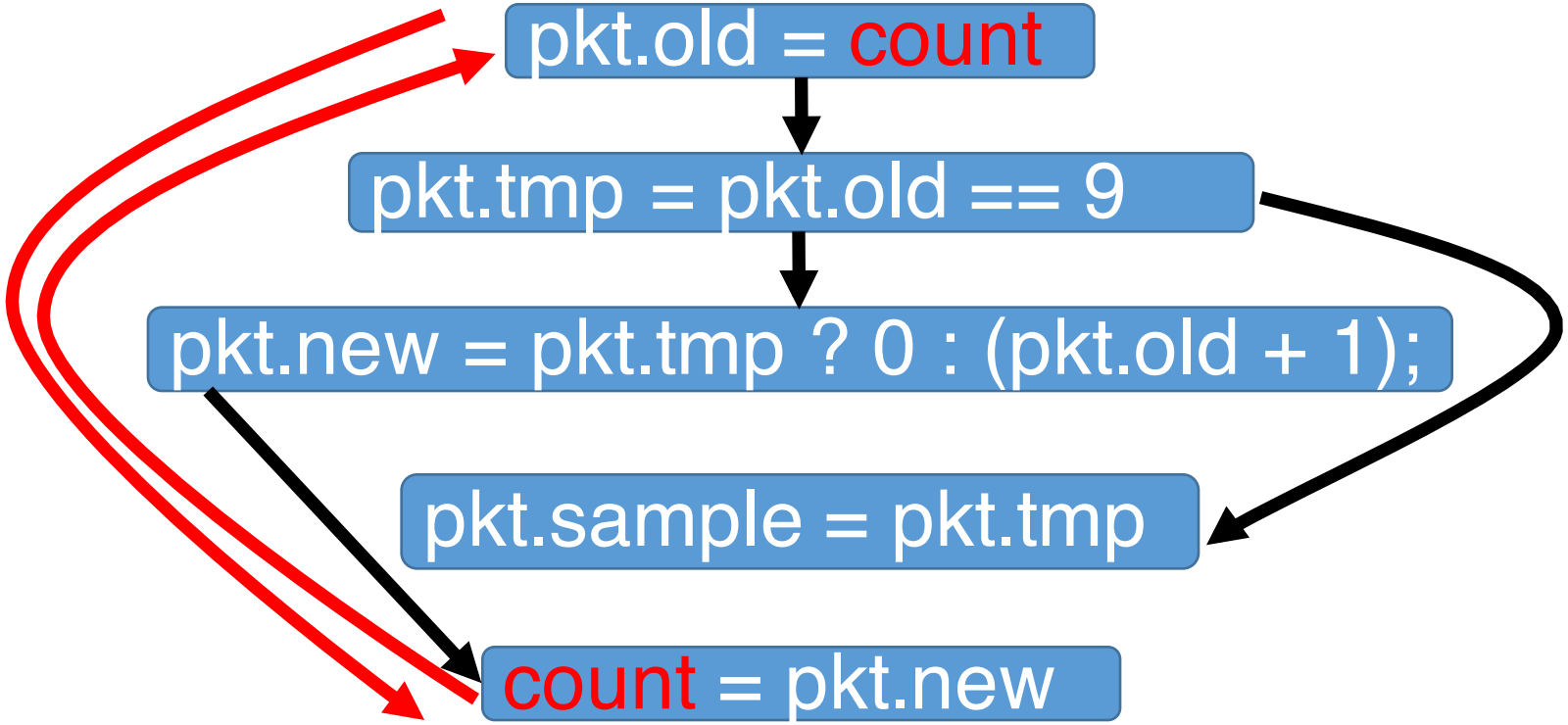
Code Pipelining



Packet field dependencies

Preprocessing → Sequential to parallel code → Hardware constraints

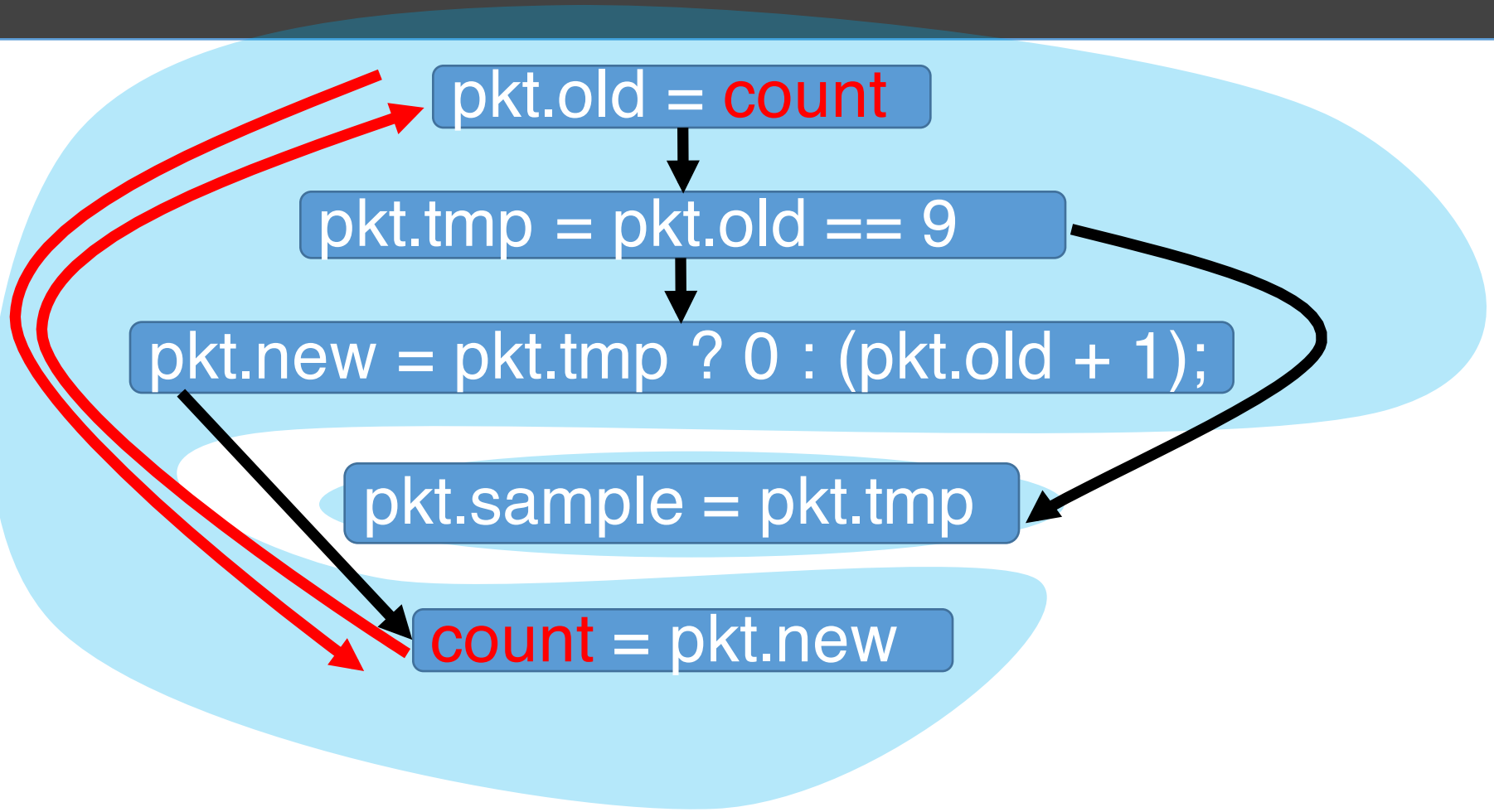
Code Pipelining



State dependencies

Preprocessing → Sequential to parallel code → Hardware constraints

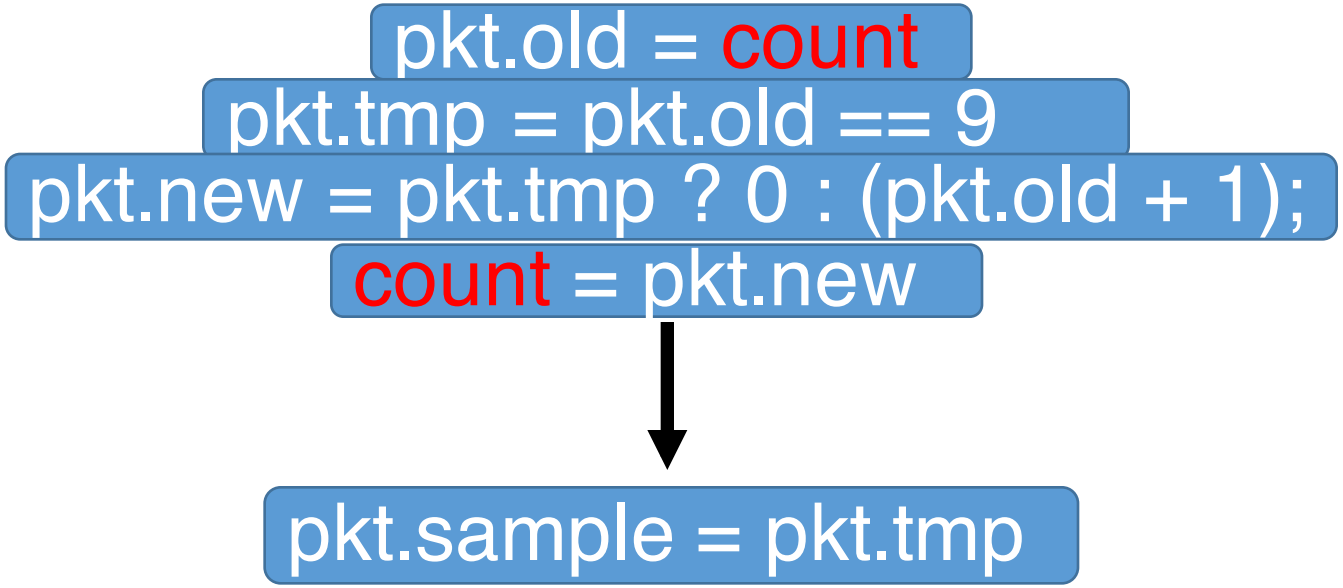
Code Pipelining



Strongly connected components

Code Pipelining

Preprocessing → Sequential to parallel code → Hardware constraints



Condensed DAG

Canonicalization → Sequential to parallel code → Hardware constraints

Code Pipelining

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;
```

Code pipeline

Canonicalization → Sequential to parallel code → Hardware constraints

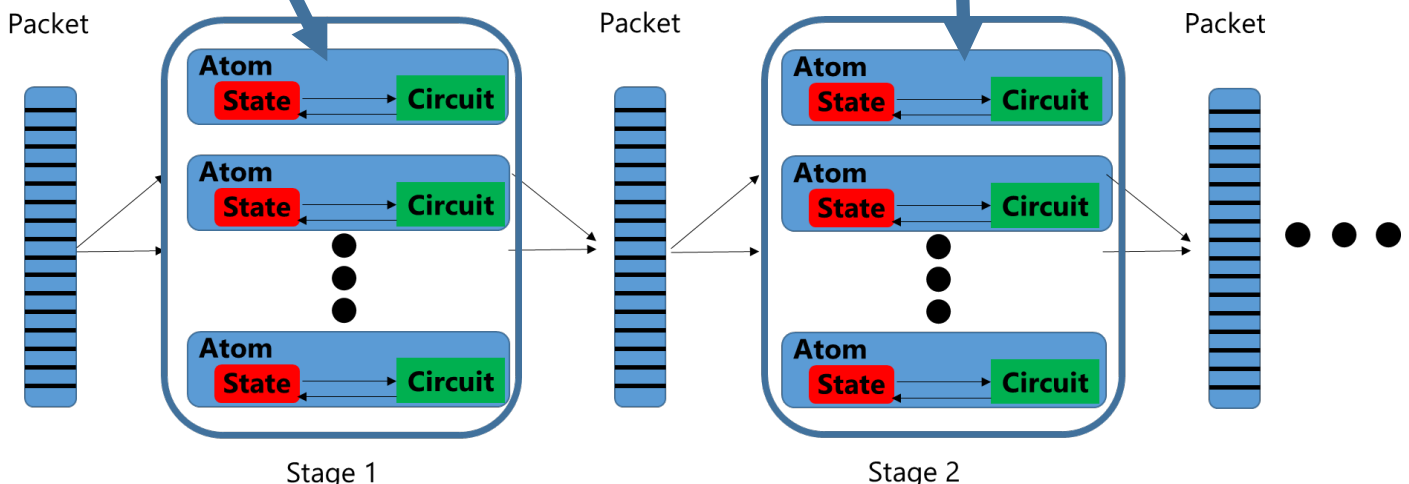
Instruction mapping

Stage 1

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

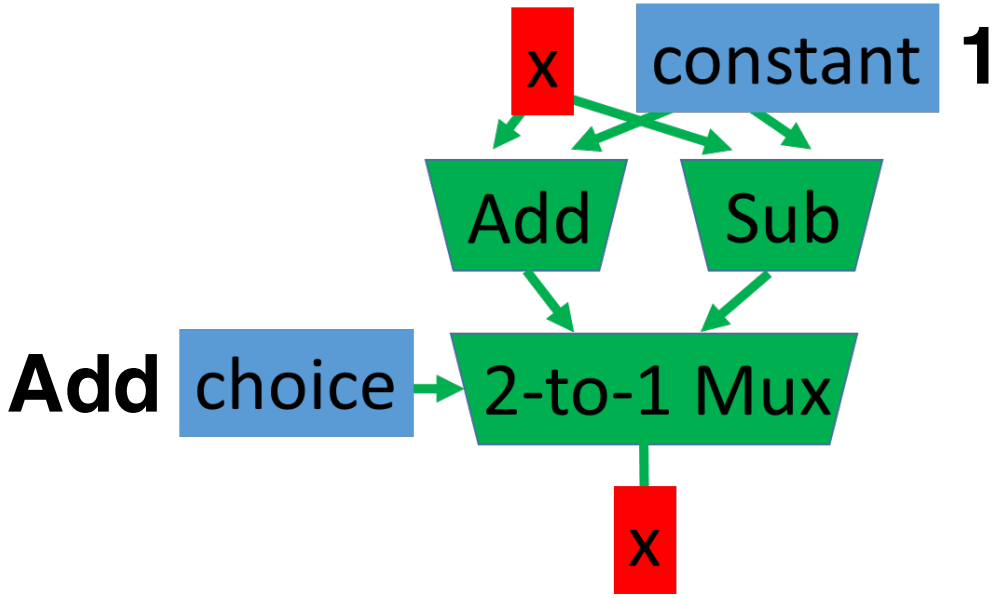
Stage 2

```
pkt.sample = pkt.tmp;
```



Instruction mapping: example

x = x + 1 maps to this atom
x = x * x doesn't map



Evaluation

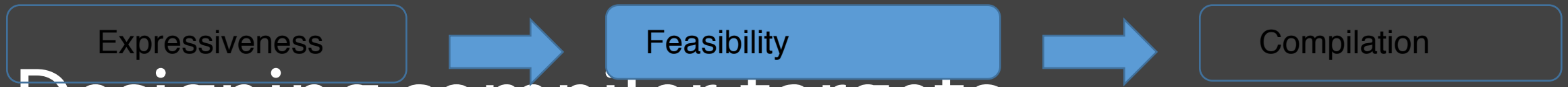
- Expressiveness: Can we program real algorithms using packet transactions?
- Feasibility: Can we design compiler targets with small area overheads?
- Compilation: Can the algorithms be compiled to the targets?

Expressiveness of packet transactions

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

Expressiveness of packet transactions

Algorithm	LOC	Auto-generated 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



Designing compiler targets

- Design both stateless and stateful atoms
  - Stateless: easy because stateless operations can be pipelined
  - Stateful: determines which algorithms can run at line rate
- 1 GHz clock frequency
  - 300 each for stateful, stateless atoms (10 atoms per stage, 30 stages)
- Synthesize atoms to 32-nm transistor library
  - Estimate area overhead relative to 200 sq. mm chip.

Atoms used in targets

Stateless

```
pkt.f3 =(pkt.f1 | constant)
          OP
          (pkt.f2 | constant);
where
OP = {+, -, AND, OR, > ,<, ...}
```

+

Stateful

Read/Write (R/W)

```
x = (pkt.f | constant);
```

ReadAddWrite (RAW)

```
x = (x | 0) + (pkt.f | constant);
```

Predicated ReadAddWrite (PRAW)

```
if (predicate(x, pkt.f1, pkt.f2))
  x = (x | 0) + (pkt.f1 | pkt.f2 | constant);
else:
  x = x
```

Expressiveness → Feasibility → Compilation

Atoms used in targets

Stateful

Stateless

pkt.f3 =(pkt.f1 | constant)
OP
(pkt.f2 | constant);
where
OP = {+, -, AND, OR, > ,<, ...}

+

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

Expressiveness → Feasibility → Compilation

Atoms used in targets

Stateful

Stateless (0.22 %)

```
pkt.f3 =(pkt.f1 | constant)
            OP
            (pkt.f2 | constant);
where
OP = {+, -, AND, OR, > ,<, ...}
```

+

Atom	Description	Overhead
R/W	Read or write state	0.04%
RAW	Read, add, and write back	0.07%
PRAW	Predicated version of RAW	0.13%
IfElseRAW	2 RAWs, one each when a predicate is true or false	0.16%
Sub	IfElseRAW with a stateful subtraction capability	0.24%
Nested	4-way predication (nests 2 IfElseRAWs)	0.58%
Pairs	Update a pair of state variables	0.96%

Expressiveness → Feasibility → Compilation

Compiling packet transactions

Algorithm	LOC	Stages (max 30)	Max. atoms/ stage (max 10)	Most expressive stateful atom required
Bloom filter	29	4	3	R/W
Heavy hitter detection	35	10	9	RAW
Rate-Control Protocol	23	6	2	PRAW
Flowlet switching	37	3	3	PRAW
Sampled NetFlow	18	4	2	IfElseRAW
HULL	26	7	1	Sub
Adaptive Virtual Queue	36	7	3	Nested
CONGA	32	4	2	Pairs
CoDel	57	15	3	**Doesn't map**

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.

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.

Other future work

- Instruction-set design for programmable routers
- Approximate semantics for packet transactions
- Sharing memory between pipeline stages
- Programmable NICs

Canonicalization → Sequential to parallel code → Hardware constraints

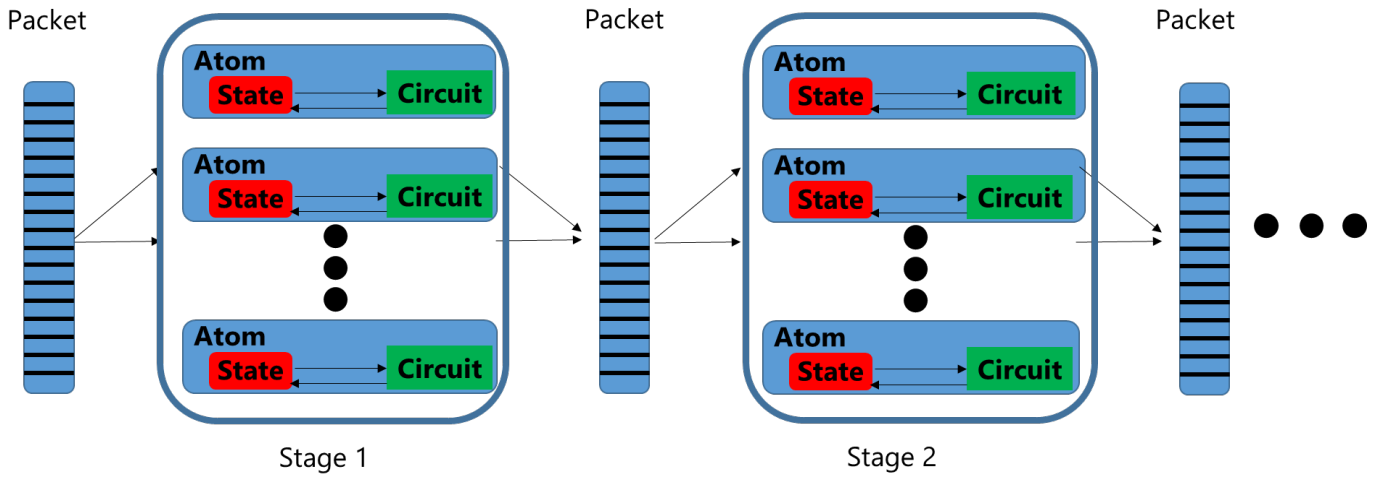
Instruction mapping: bin packing

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;
```

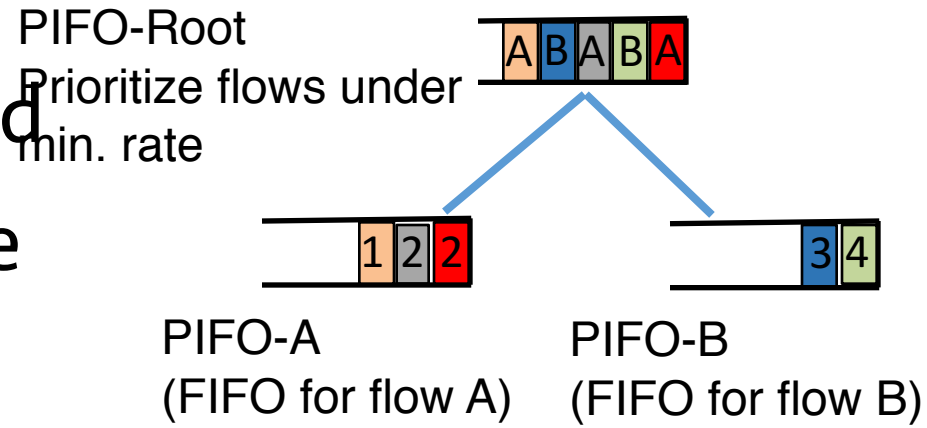


Composing PIFOs: min. rate guarantees

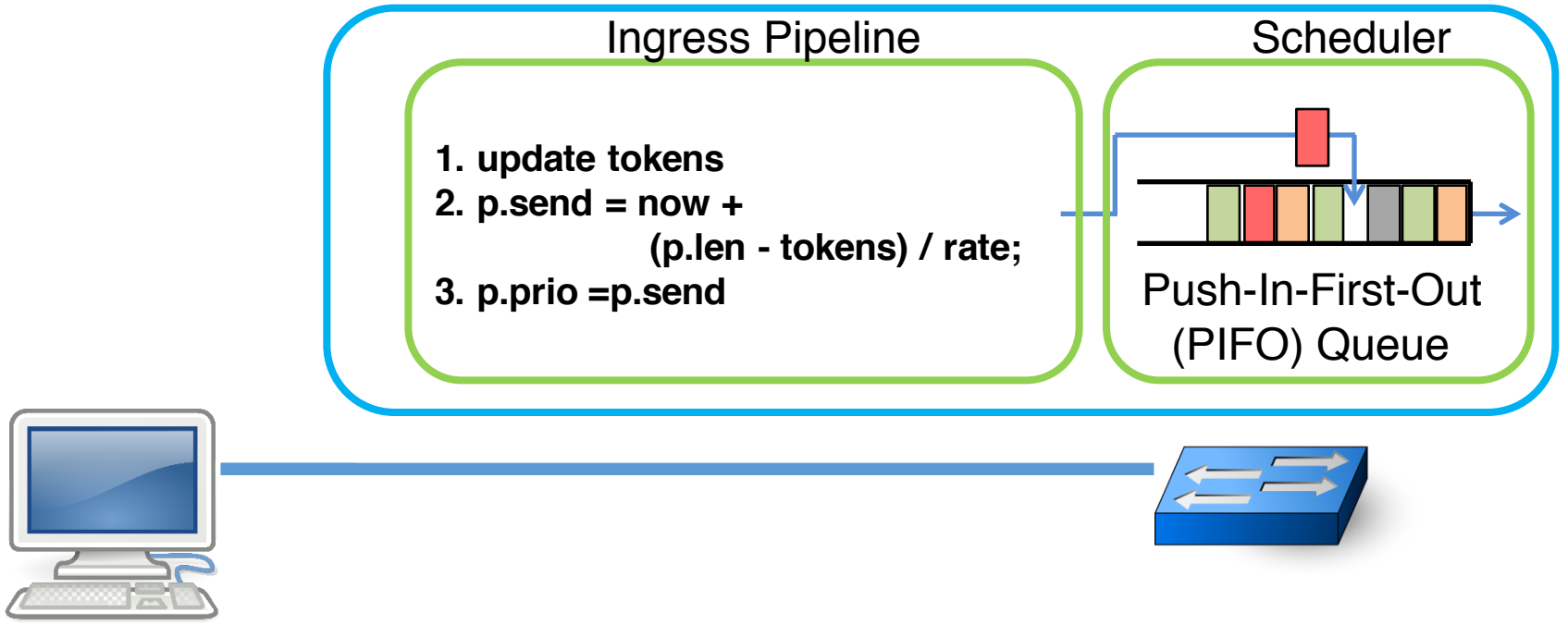
Minimum rate guarantees:

Provide each flow a guaranteed rate provided the sum of these guarantees is below capacity.

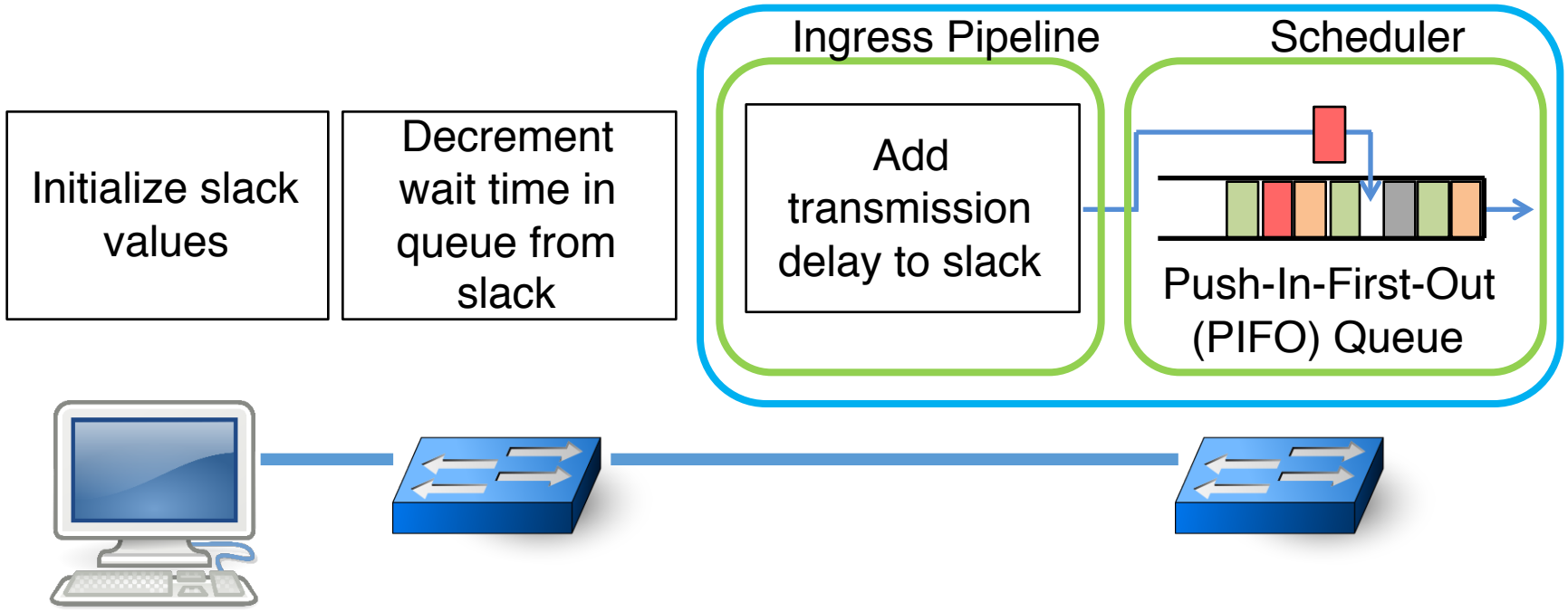
Composing PIFOs



Traffic Shaping



LSTF


Initialize slack values
Decrement wait time in queue from slack
Ingress Pipeline
Add transmission delay to slack
Scheduler
Push-In-First-Out (PIFO) Queue

Packet transactions: conclusion

- More familiar abstraction
- Programming line-rate switches need not be hard
- Simple user interface: code that compiles runs at line rate

The PIFO abstraction in one slide

- PIFO: A sorted array that let us insert an entry (packet or PIFO pointer) into a PIFO based on a programmable priority
- Entries are always dequeued from the head
- If an entry is a packet, dequeue and transmit it
- If an entry is a PIFO, dequeue it, and continue recursively

Motivating packet transactions

- Example: count number of packets
- On enqueue:
- Calculate average queue size
- if min < avg < max
- calculate probability p
- mark packet with probability p
- else if avg > max:
- mark packet
- Runs to completion, process one packet at a time

Language constraints on Domino

- No loops (for, while, do-while)
- No unstructured control flow (break, continue, goto)
- No pointers, heaps

Canonicalization → Sequential to parallel code → Hardware constraints

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 ];
```

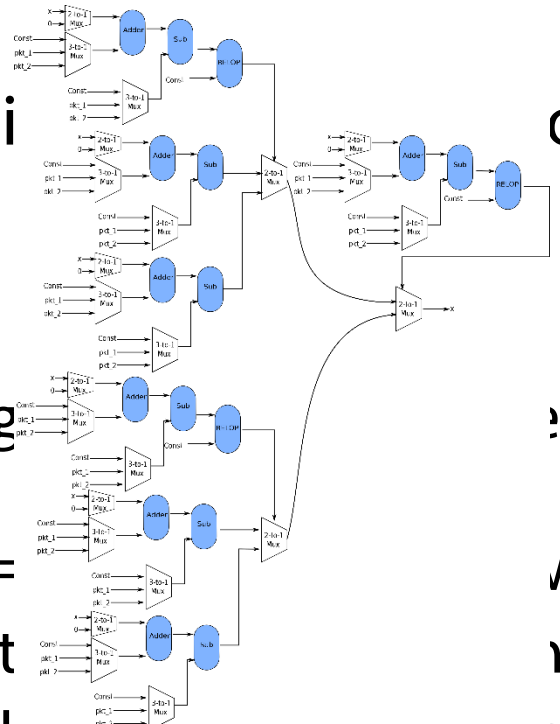
```
pkt.tmp  = pkt.arrival - last_time[pkt.id];
pkt.tmp2 = pkt.tmp > THRESHOLD;
saved_hop [ pkt . id ] = pkt.tmp2
                         ? pkt . new_hop
                         : saved_hop [ pkt . id ];
```

Instruction mapping: results

- Generic method to handle fairly complex templates
- Templates determi ogram can run at line rate.
- Example results:
  - Flowlet switching xecution to save next hop information:

    saved_hop[pkt.id] = v_hop : saved_hop[pkt.id]
  - Simple increment itter detection

    count_min_sketch[hash] = count_min_sketch[hash] + 1



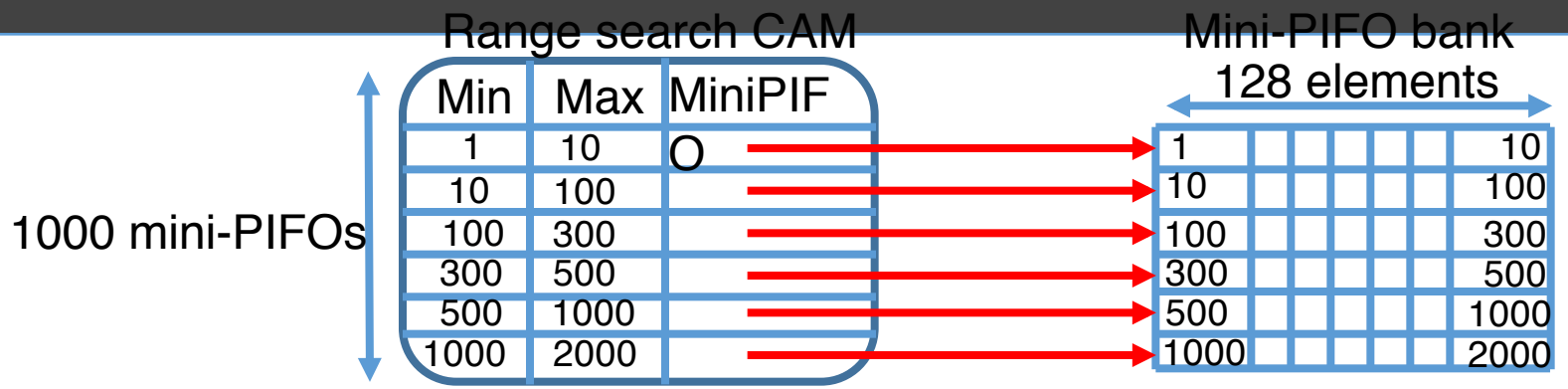
Generating P4 code

- 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

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

PIFO in hardware: HotNets version



- Meets timing at 1 GHz on a 16 nm node
- 5 % area overhead for 3-level hierarchy
- Challenges wisdom that sorting is hard

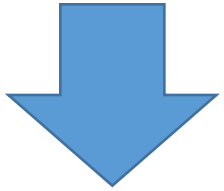
Branch Removal

```
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,

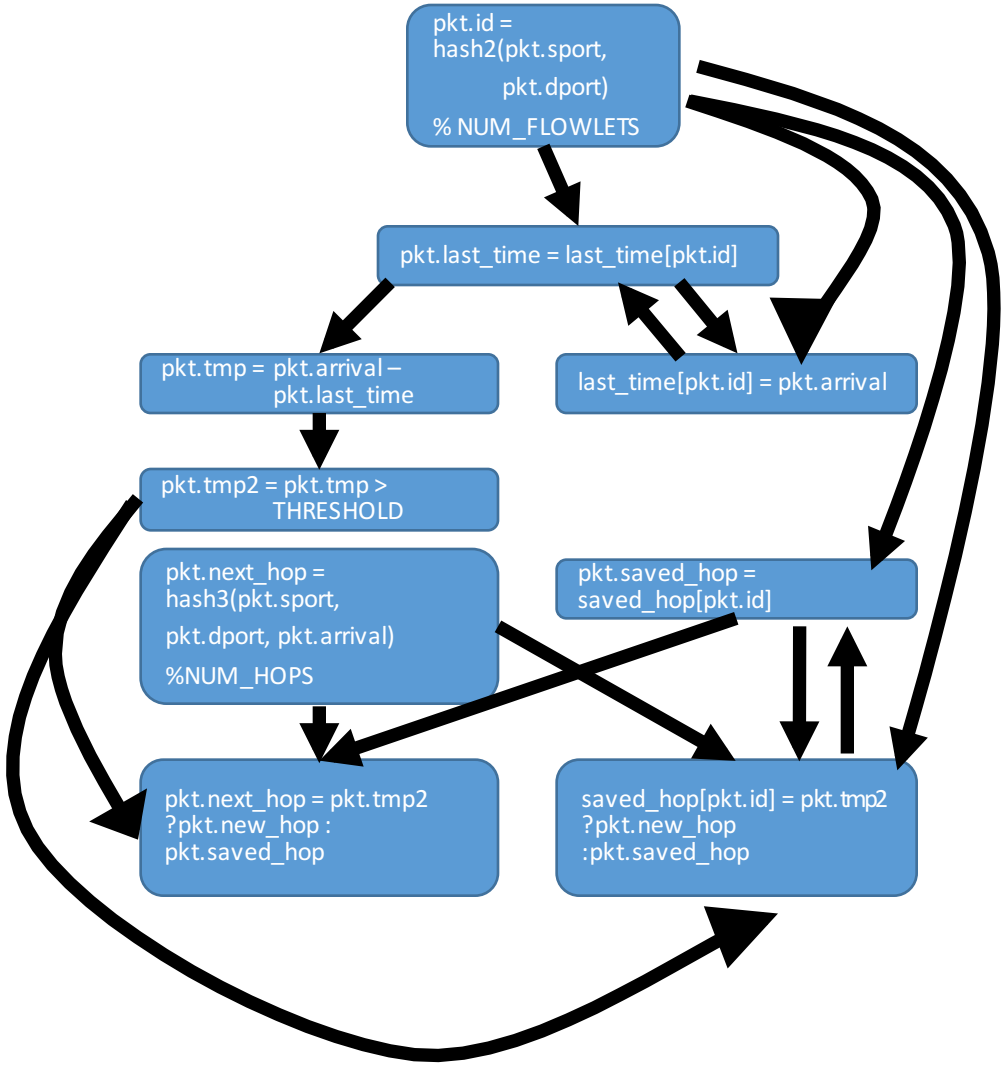
  codelet and template functions agree?
- Quantified boolean satisfiability (QBF) problem
- Use the SKETCH program synthesis tool to automate it

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.

Canonicalization → Sequential to parallel code → Hardware constraints

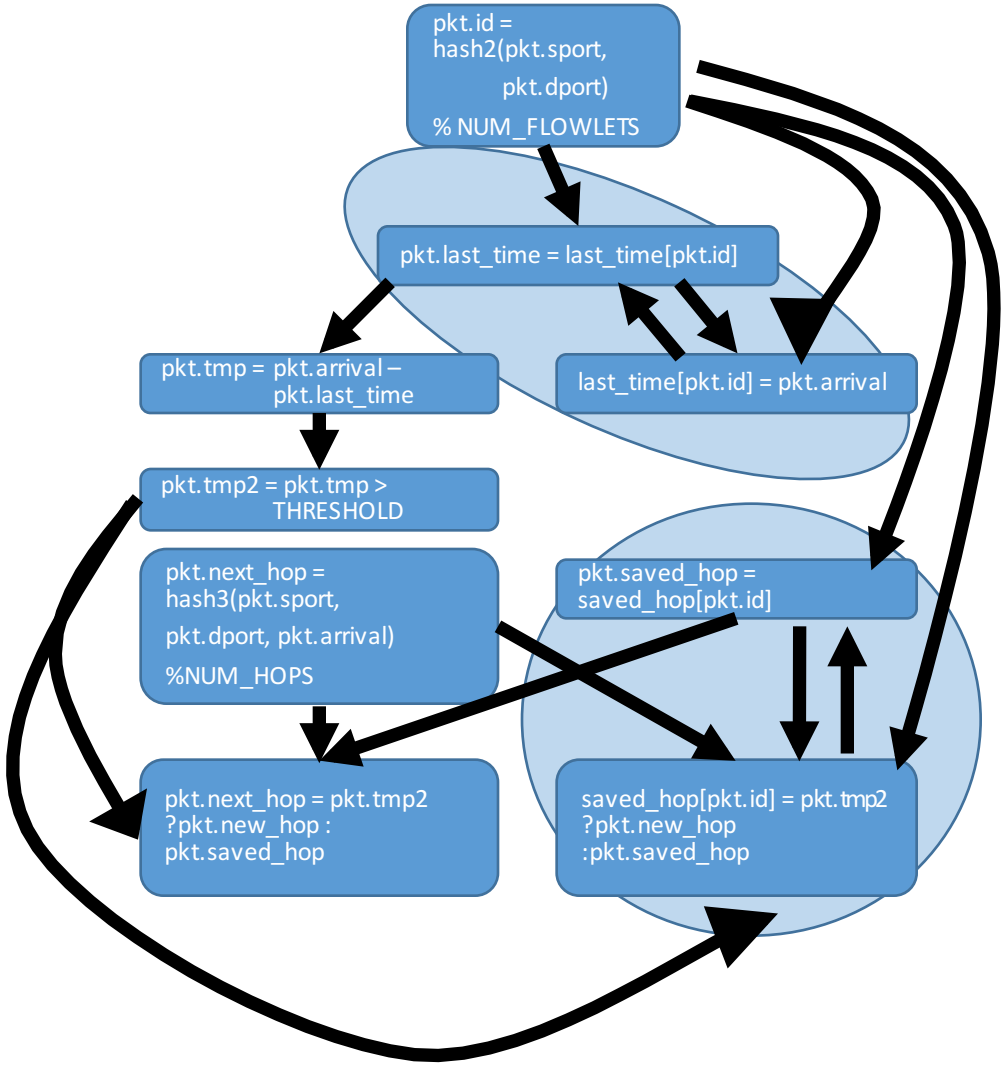
Code Pipelining



Pair up read/write flanks

Canonicalization → Sequential to parallel code → Hardware constraints

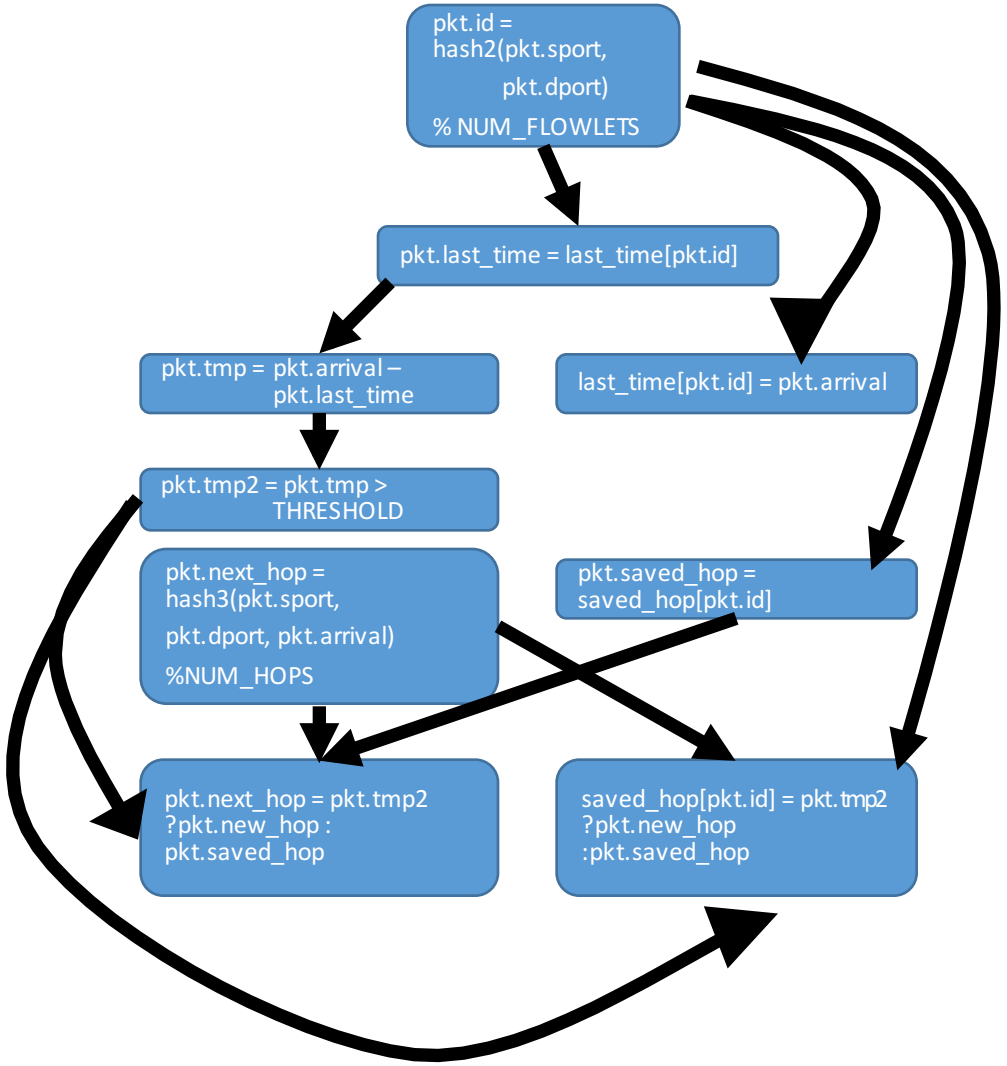
Code Pipelining



Condense strongly connected components into codelets

Canonicalization → Sequential to parallel code → Hardware constraints

Code Pipelining



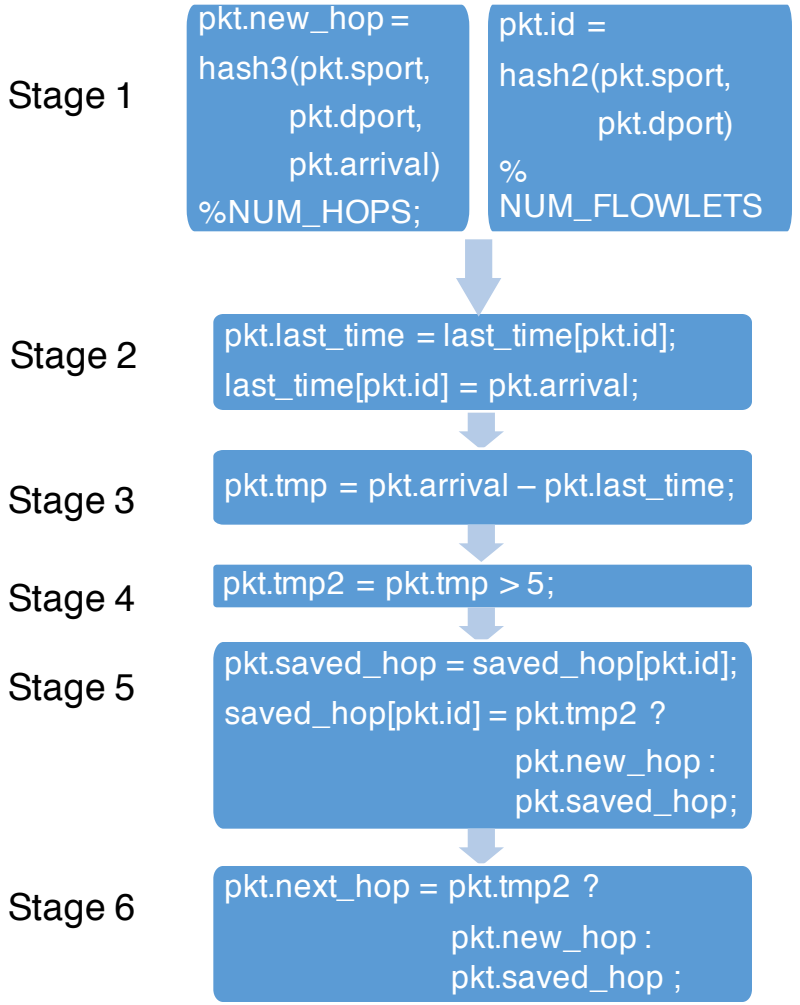
Add packet-field dependencies

Programming with Packet Transactions

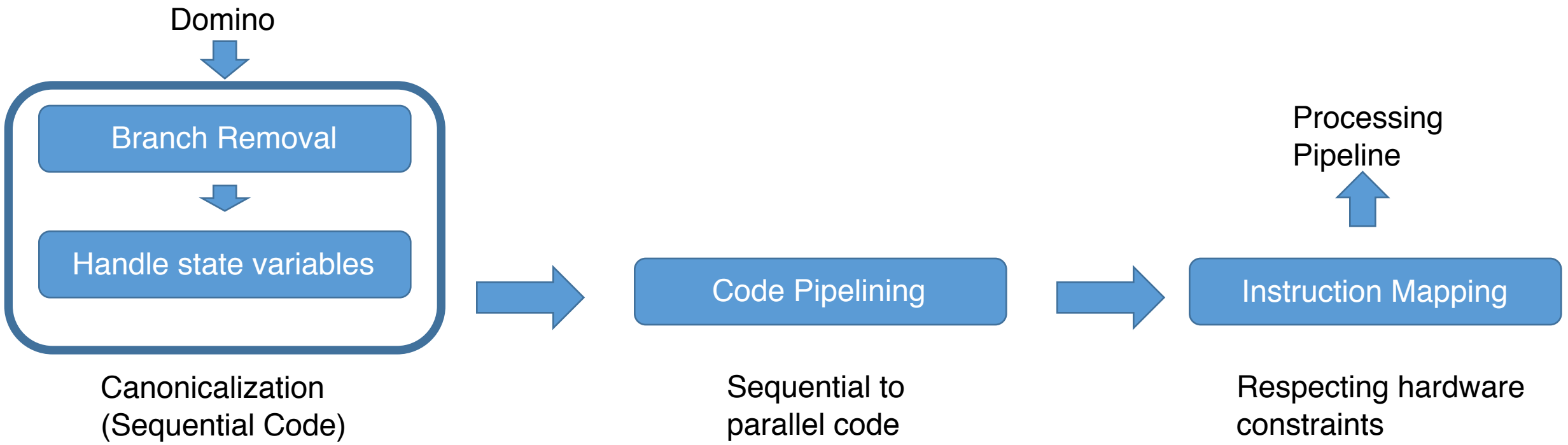
Domino

```
#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];
}
```

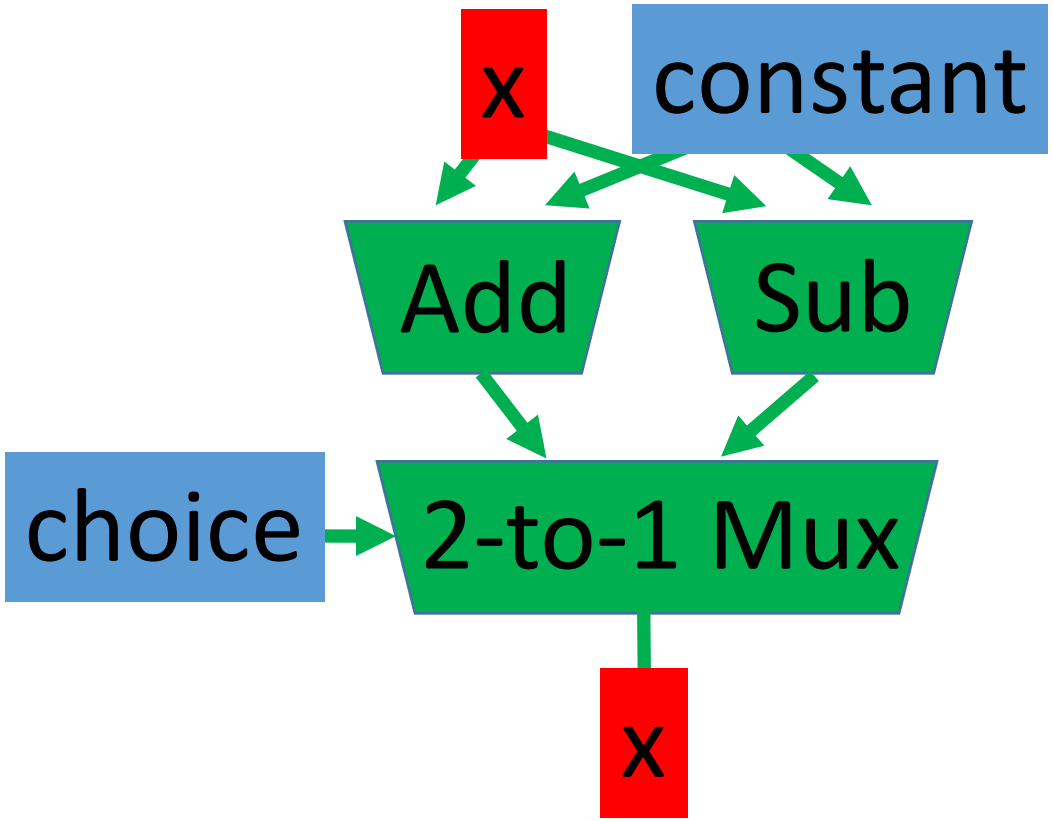
Pipeline



The Domino compiler



Diagrams


x
constant
Add
Sub
choice
2-to-1 Mux
x

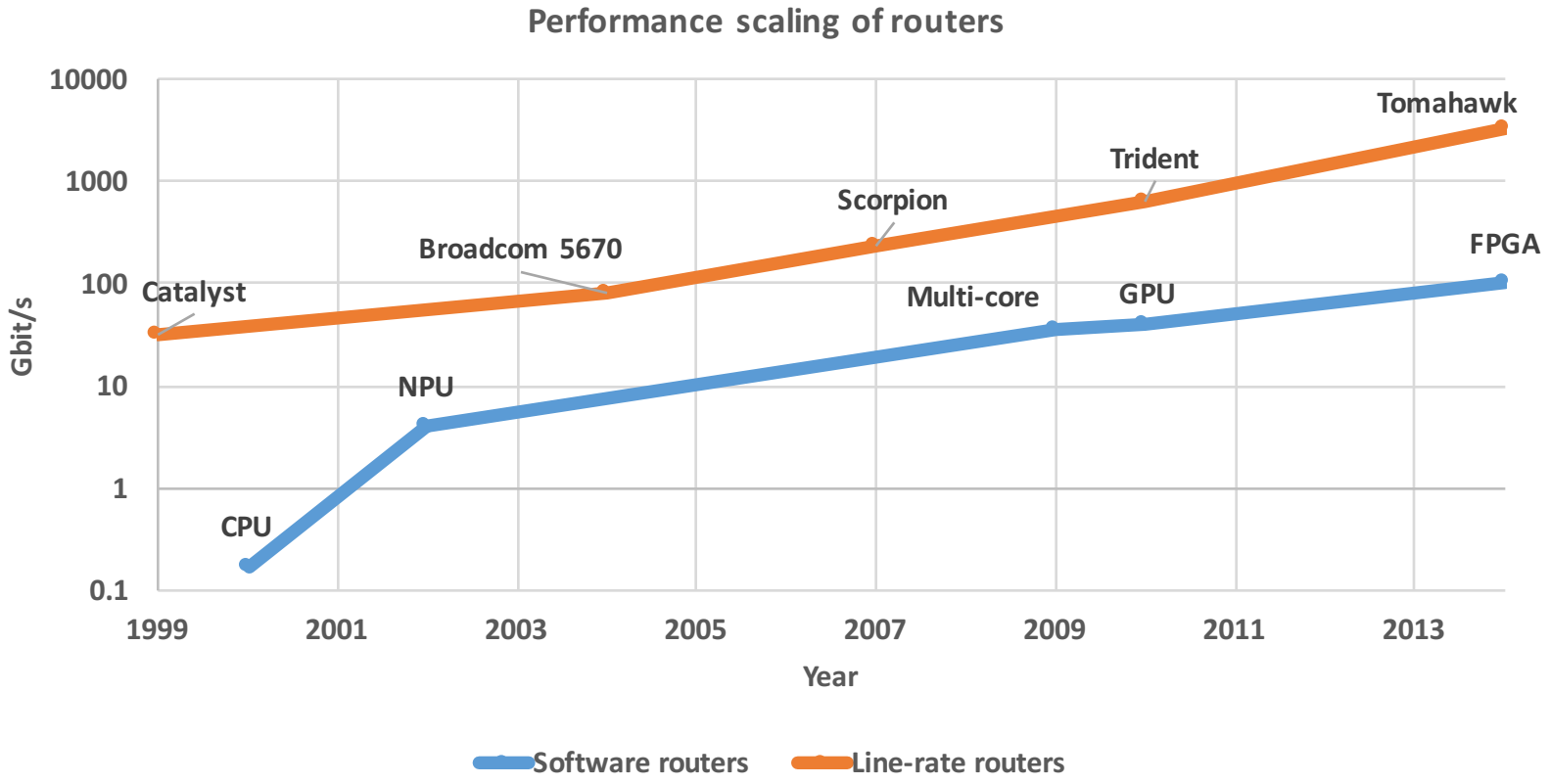
The quest for programmability

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