Packet Transactions: High-Level Programming for Line-Rate Switches

Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, Steve Licking

Programmability at line rate

- Programmable: Can we express new data-plane algorithms?
  - Active queue management
  - Congestion control
  - Measurement
  - Load balancing

- Line rate: Highest capacity supported by dedicated hardware

Programmable switching chips

Same performance as fixed-function chips, some programmability

E.g., FlexPipe, Xpliant, Tofino

Where do programmable switches fall short?

- Hard to program data-plane algorithms today
  - Hardware good for stateless tasks (forwarding), not stateful ones (AQM)
  - Low-level languages (P4, POF).

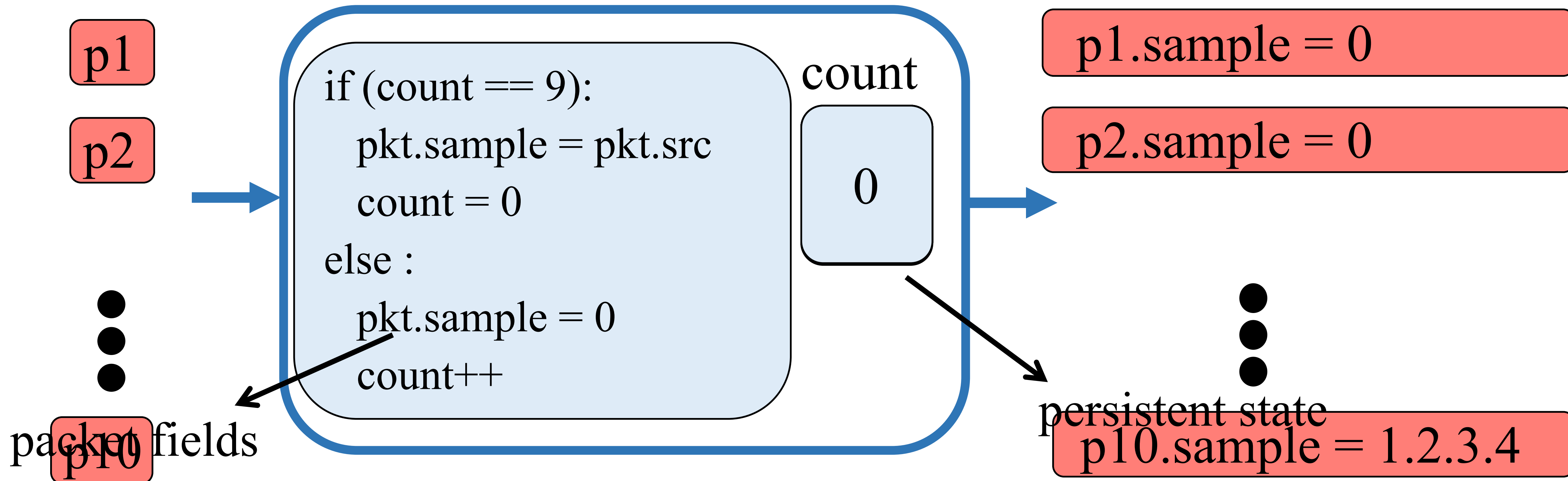
- Challenges
  - Can we program data-plane algorithms in a high-level language?
  - Can we design a stateful instruction set supporting these algorithms?

Contributions

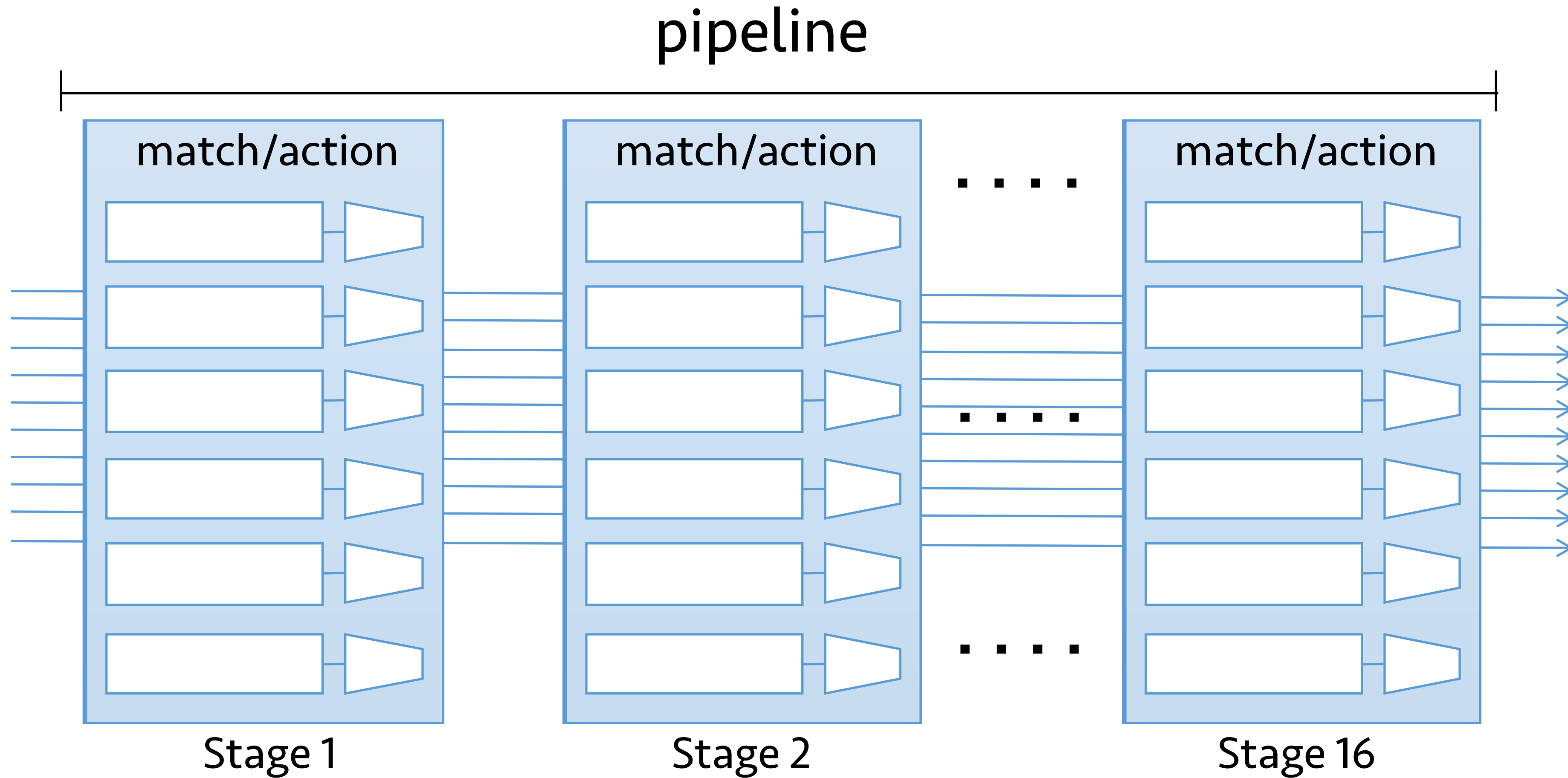
- Packet transaction: High-level abstraction for data-plane algorithms
  - Examples of several algorithms as packet transactions
- Atoms: A representation for switch instruction sets
  - Seven concrete stateful instructions
- Compiler from packet transactions to atoms
  - Allows us to iteratively design switch instruction sets

Packet transactions

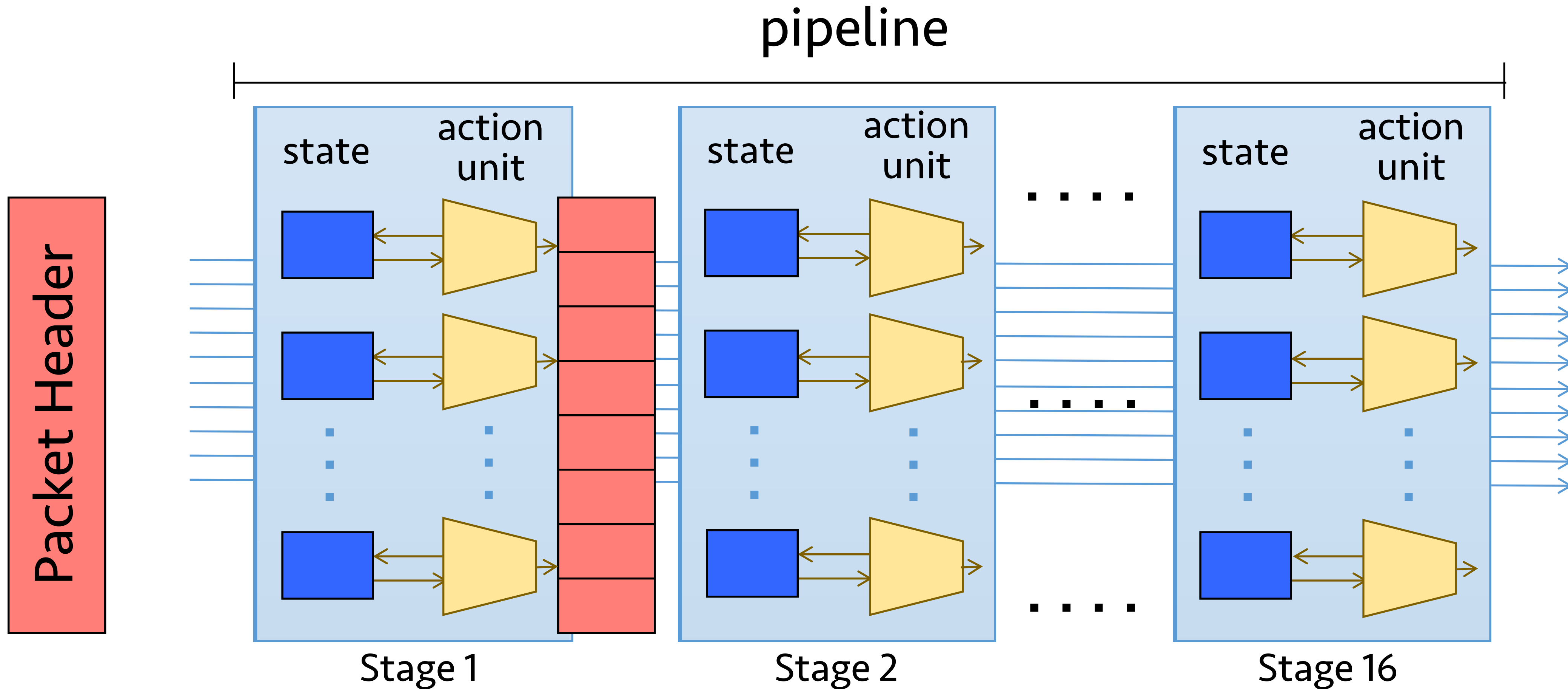
- Packet transaction: block of imperative code
- Transaction runs to completion, one packet at a time, serially



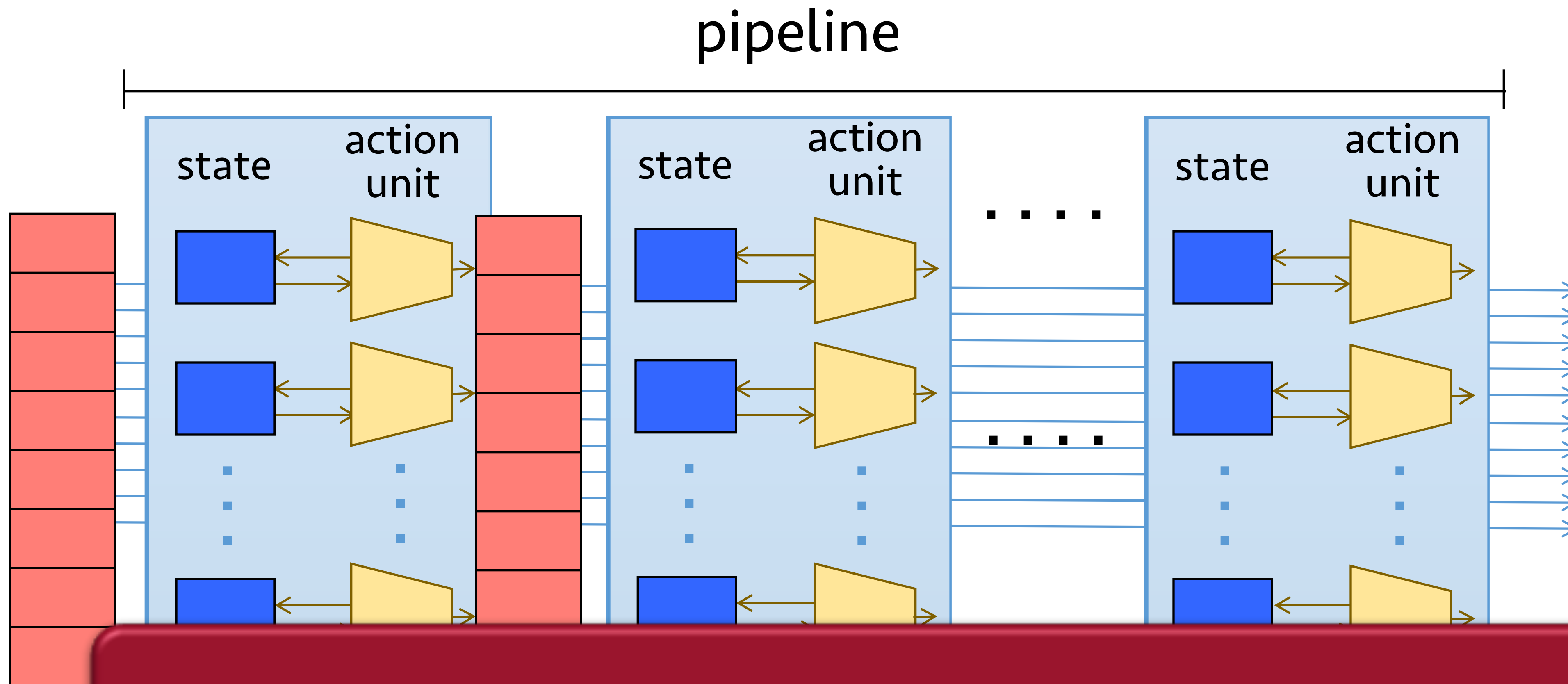
Under the hood …



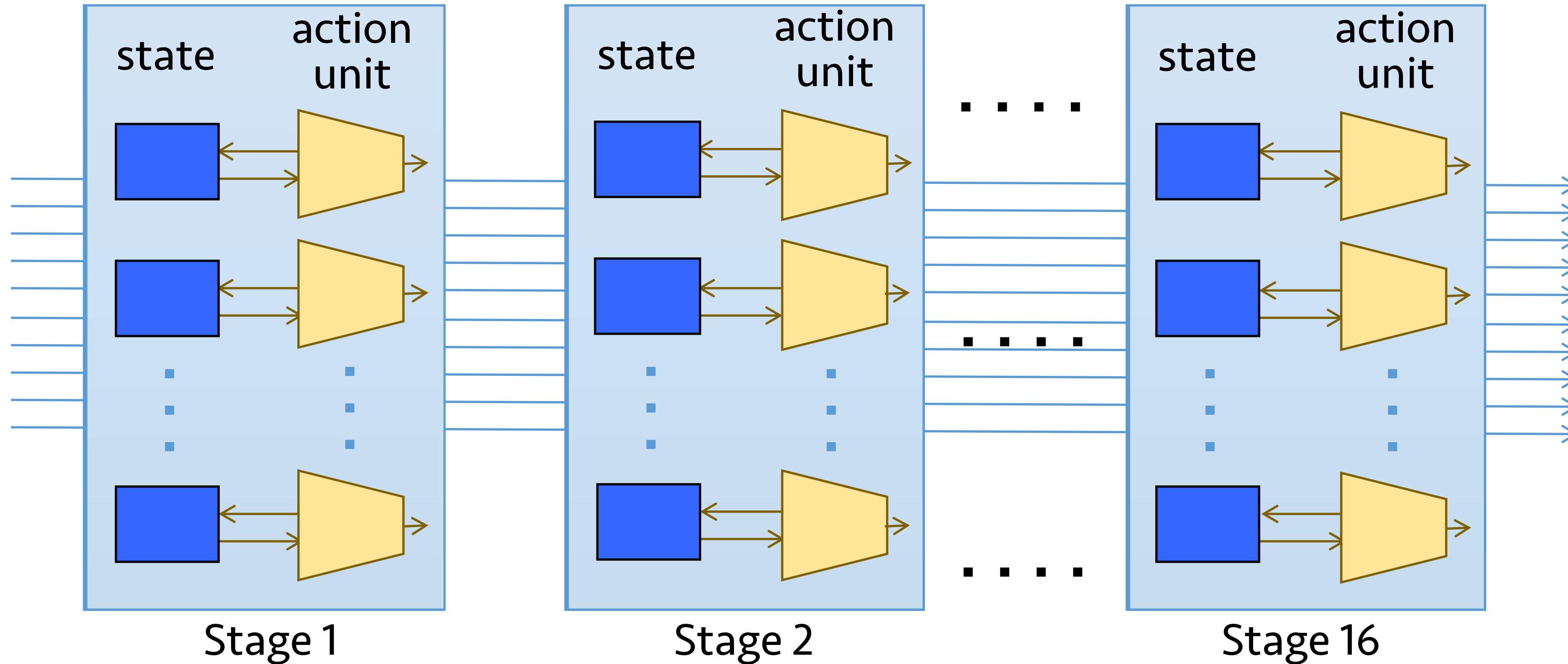
A machine model for line-rate switches



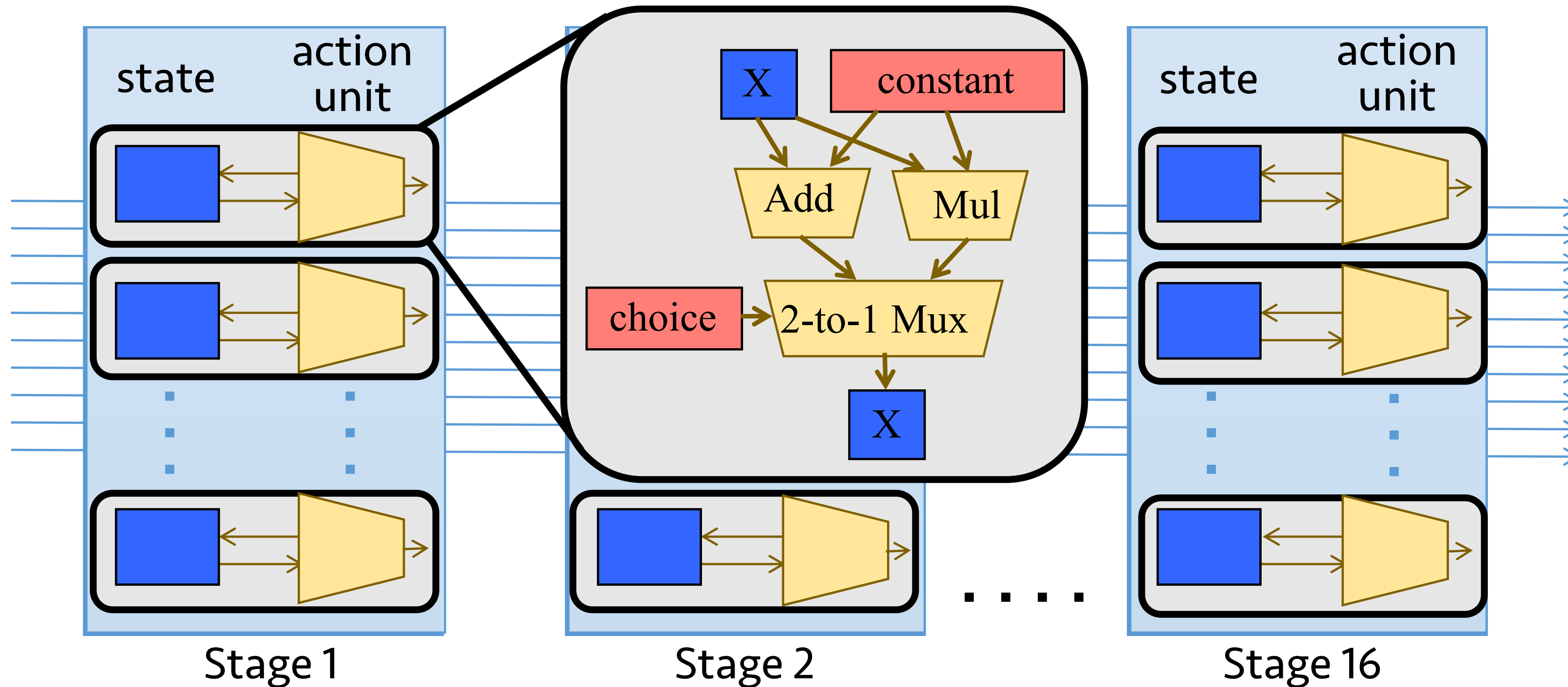
A machine model for line-rate switches



A machine model for line-rate switches



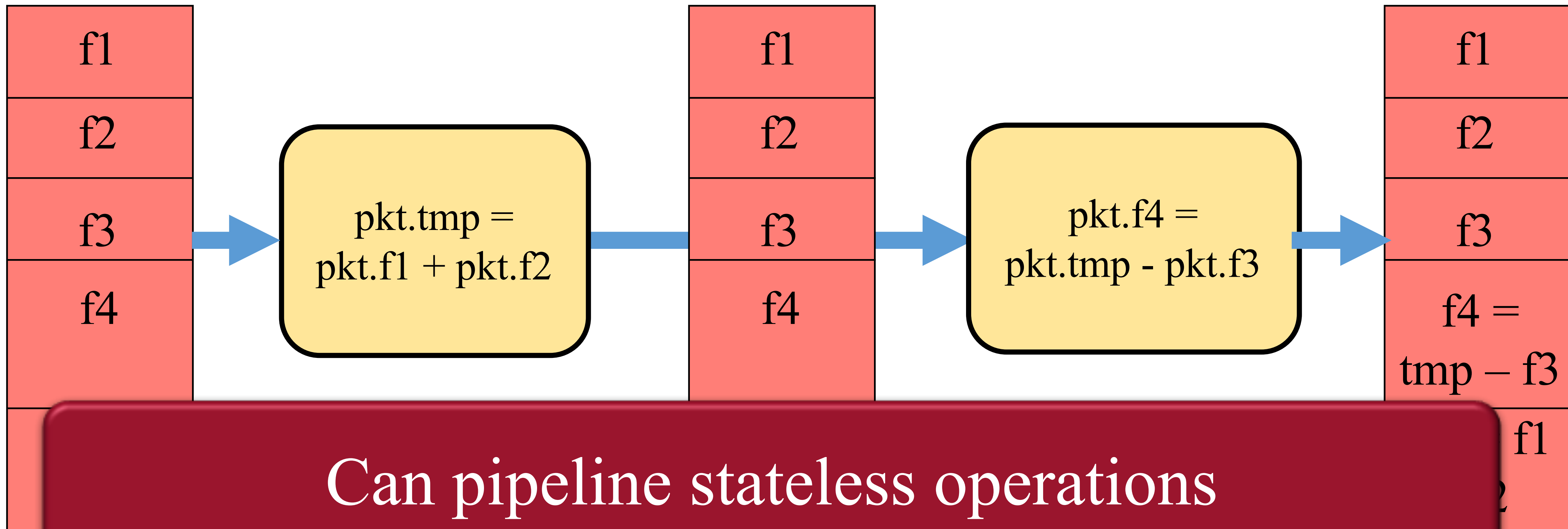
A machine model for line-rate switches



A switch's atoms constitute its instruction set

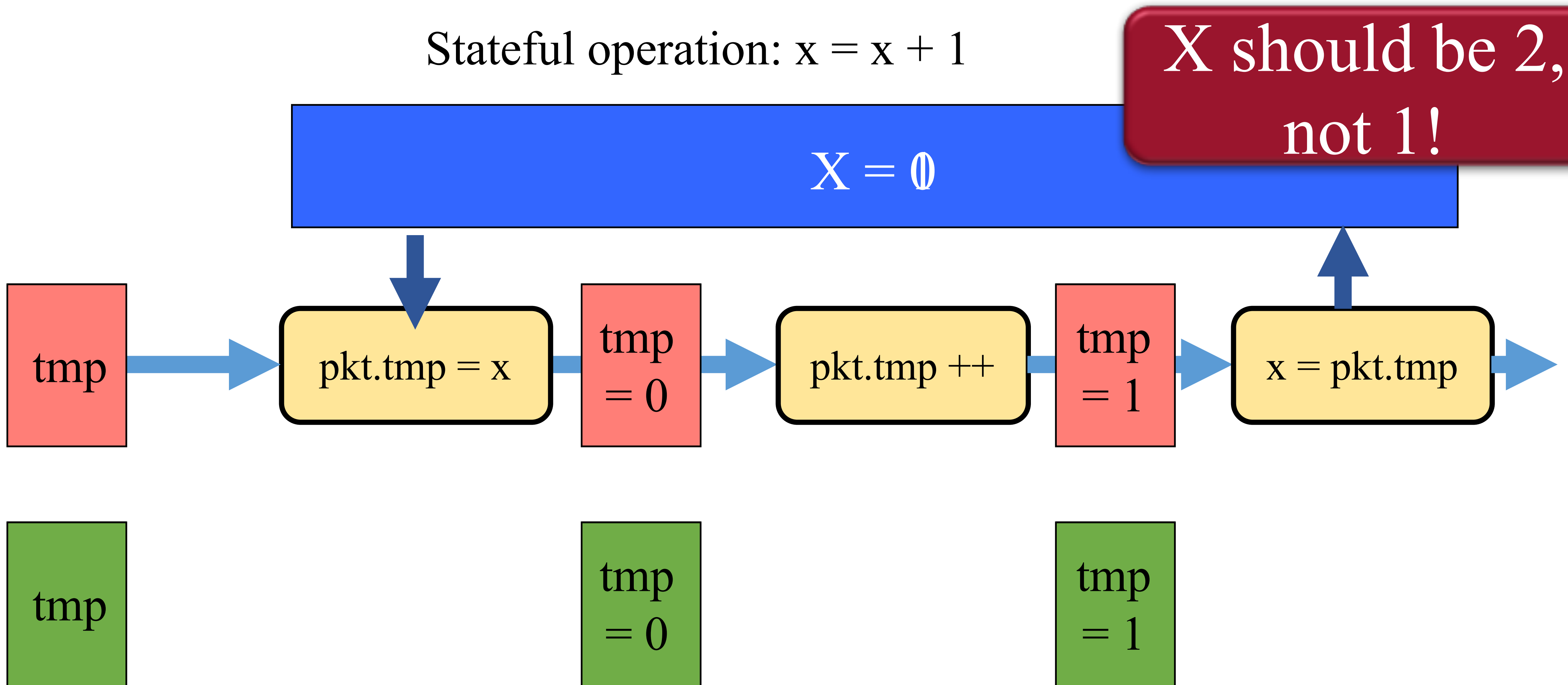
Stateless vs. stateful operations

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

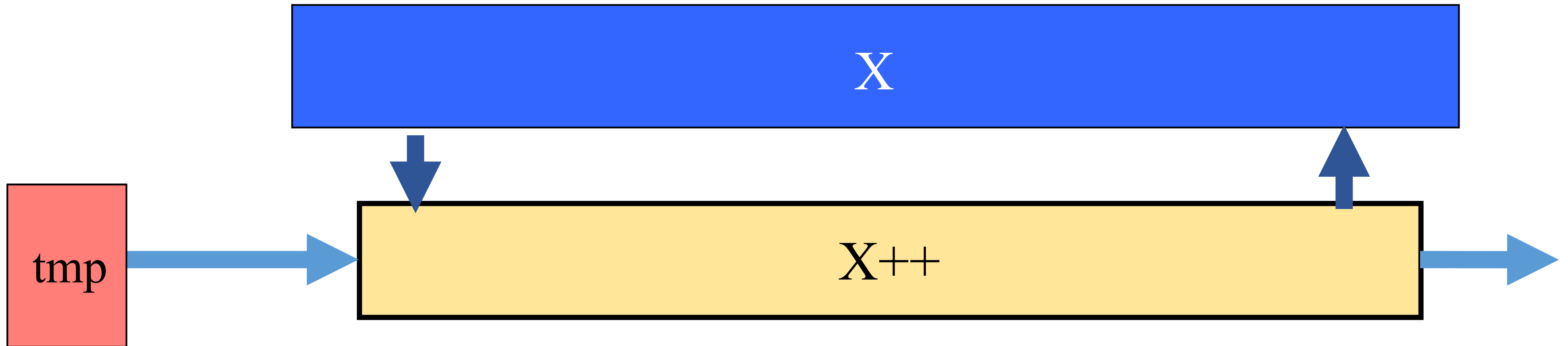


Can pipeline stateless operations

Stateless vs. stateful operations

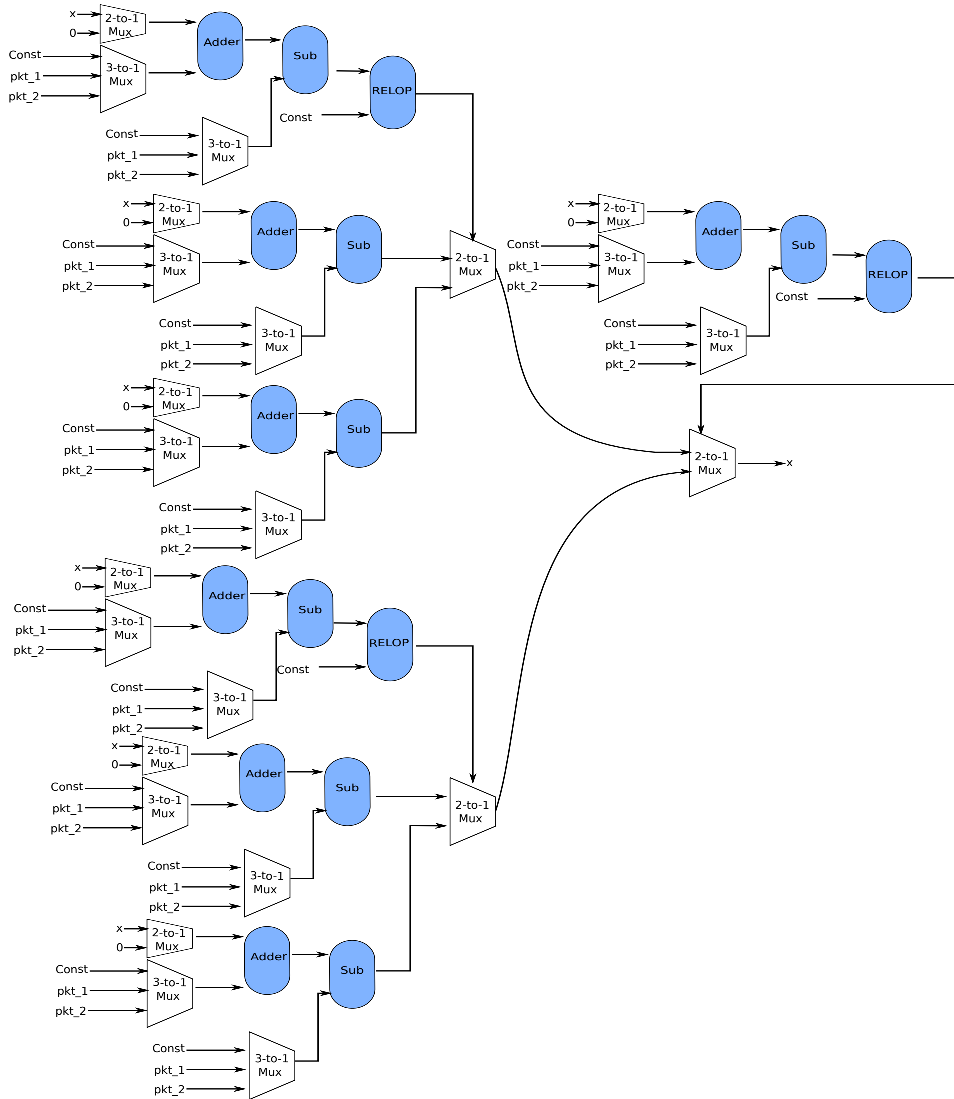


Stateless vs. stateful operations



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.

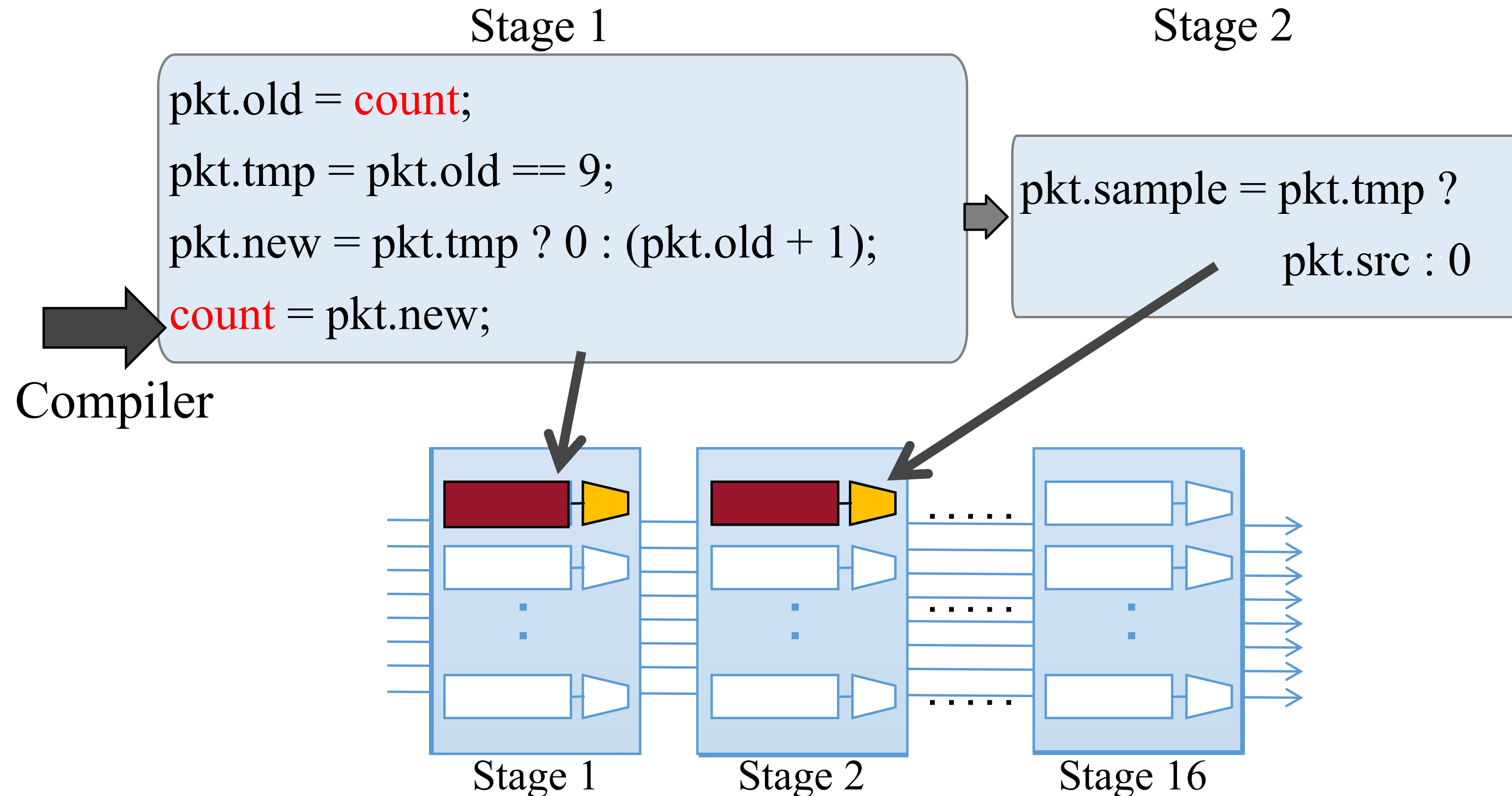
Each predicate can itself depend on the state.

Compiling packet transactions

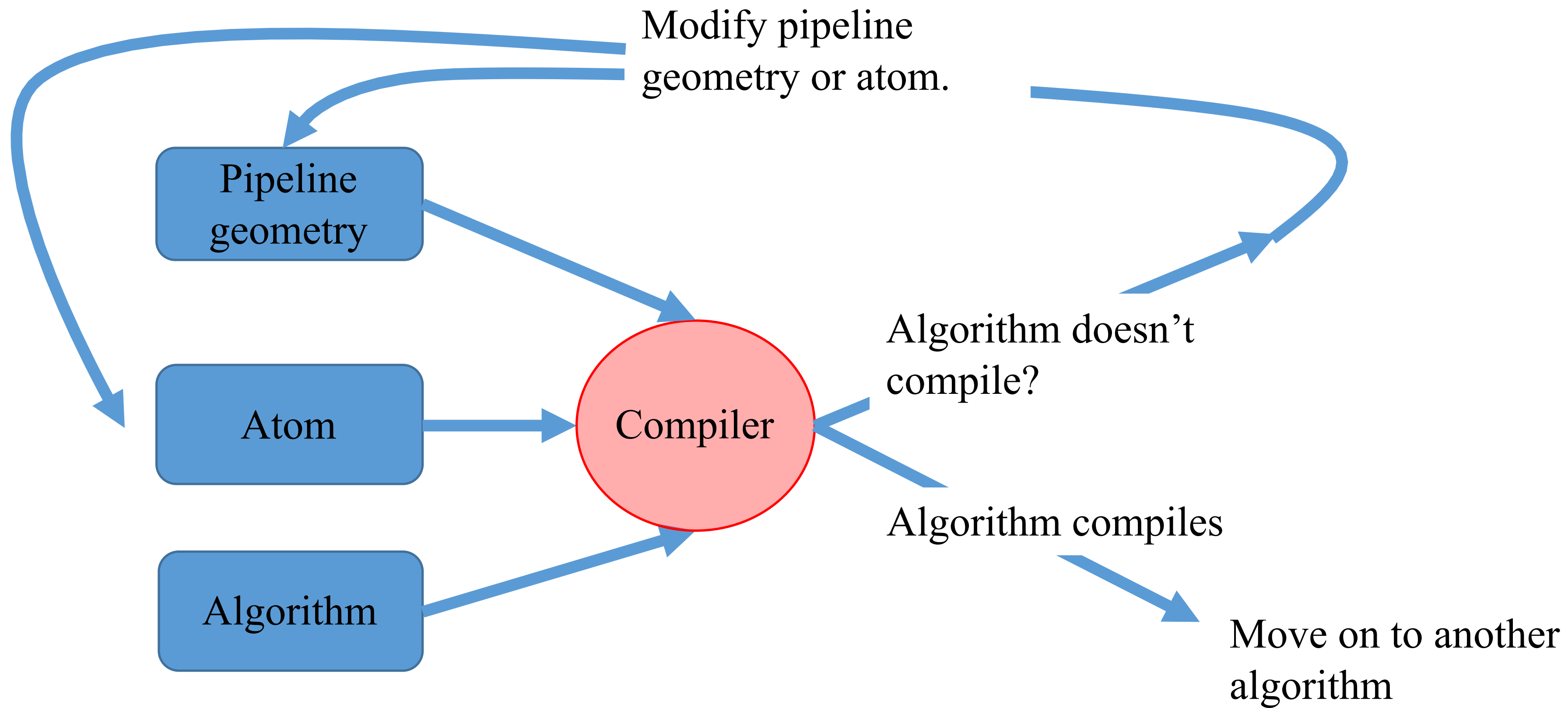
Packet Sampling Algorithm

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

Packet Sampling Pipeline



Designing programmable switches



Focus on stateful atoms, stateless operations are easily pipelined

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

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

Compilation results

Algorithm	LOC	Most expressive stateful atom required
Bloom filter	29	R/W
Heavy hitter detection	35	RAW
Rate-Control Protocol	23	PRAW
Flowlet switching	37	PRAW
Sampled NetFlow	18	IfElseRAW
HULL	26	Sub
Adaptive Virtual Queue	36	Nested
CONGA	32	Pairs
CoDel	57	**Doesn’t map**

Compilation results

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

~100 atom instances are sufficient

Modest cost for programmability

- 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

Conclusion

- Packet transactions: an abstraction for data-plane algorithms
- Atoms: a representation for switch instruction sets
- A blue print for designing switch instruction sets
- Source code: http://web.mit.edu/domino

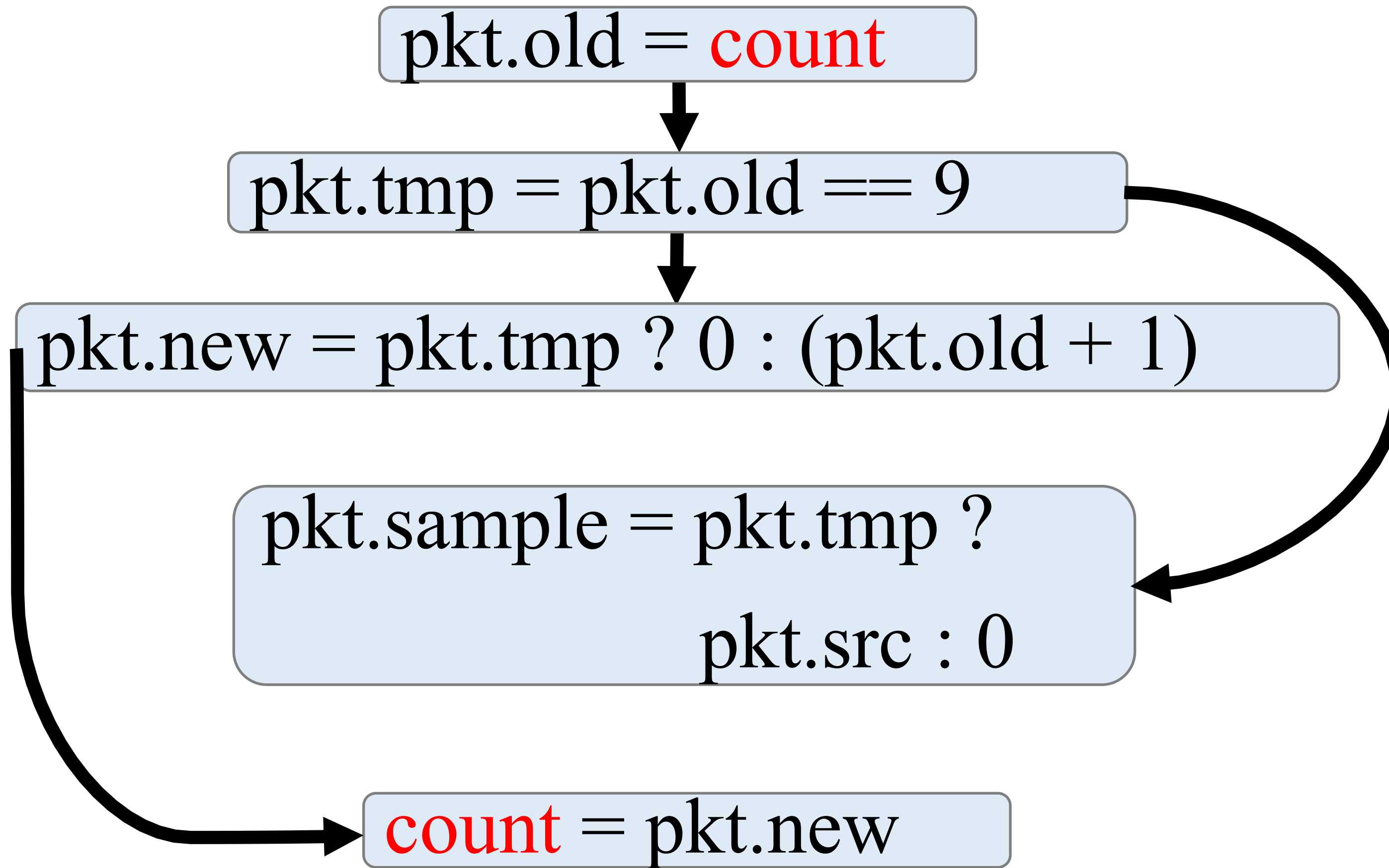
Backup slides

Sequential to pipelined code

```
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
```

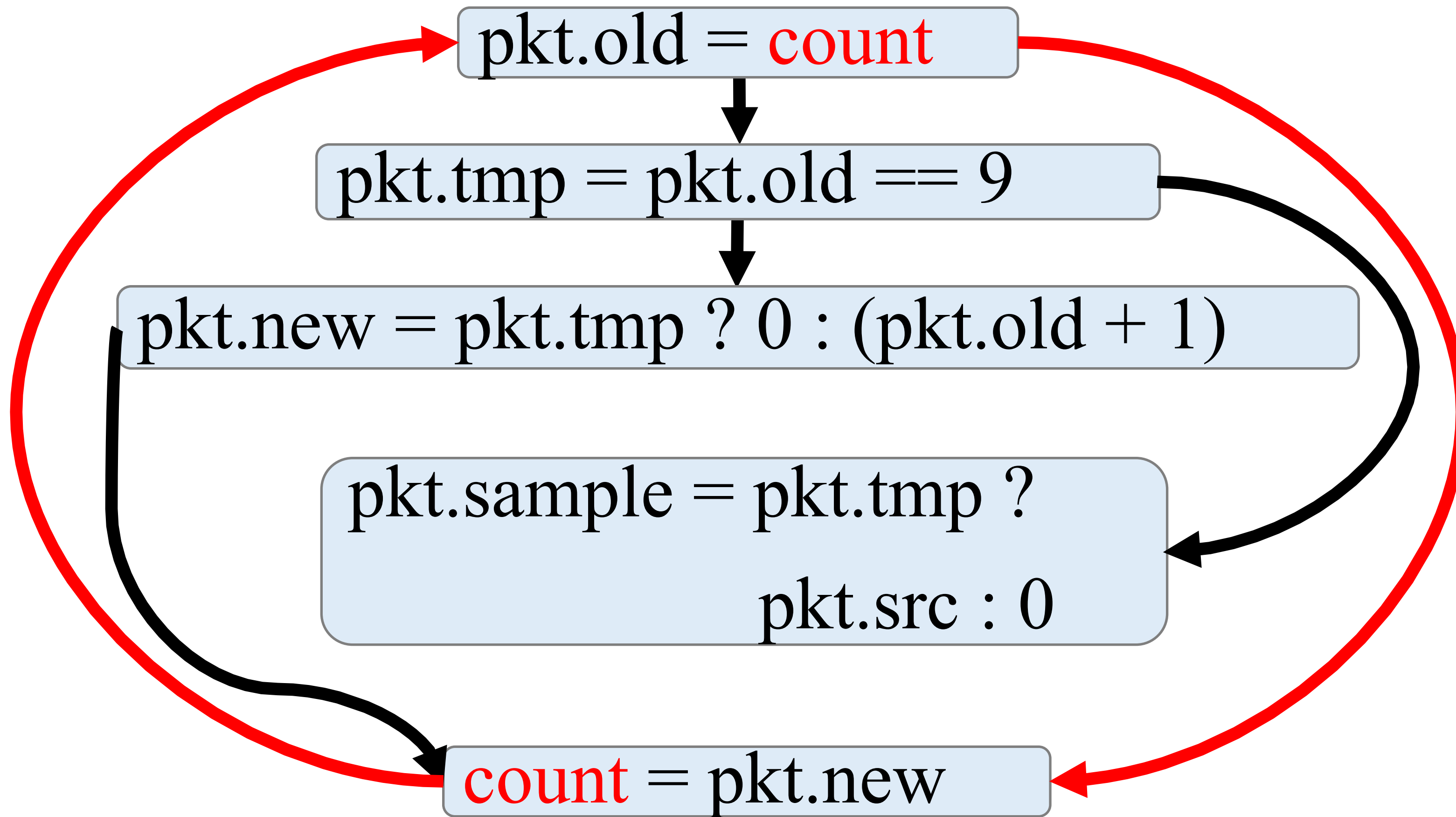
Create one node for each instruction

Sequential to pipelined code



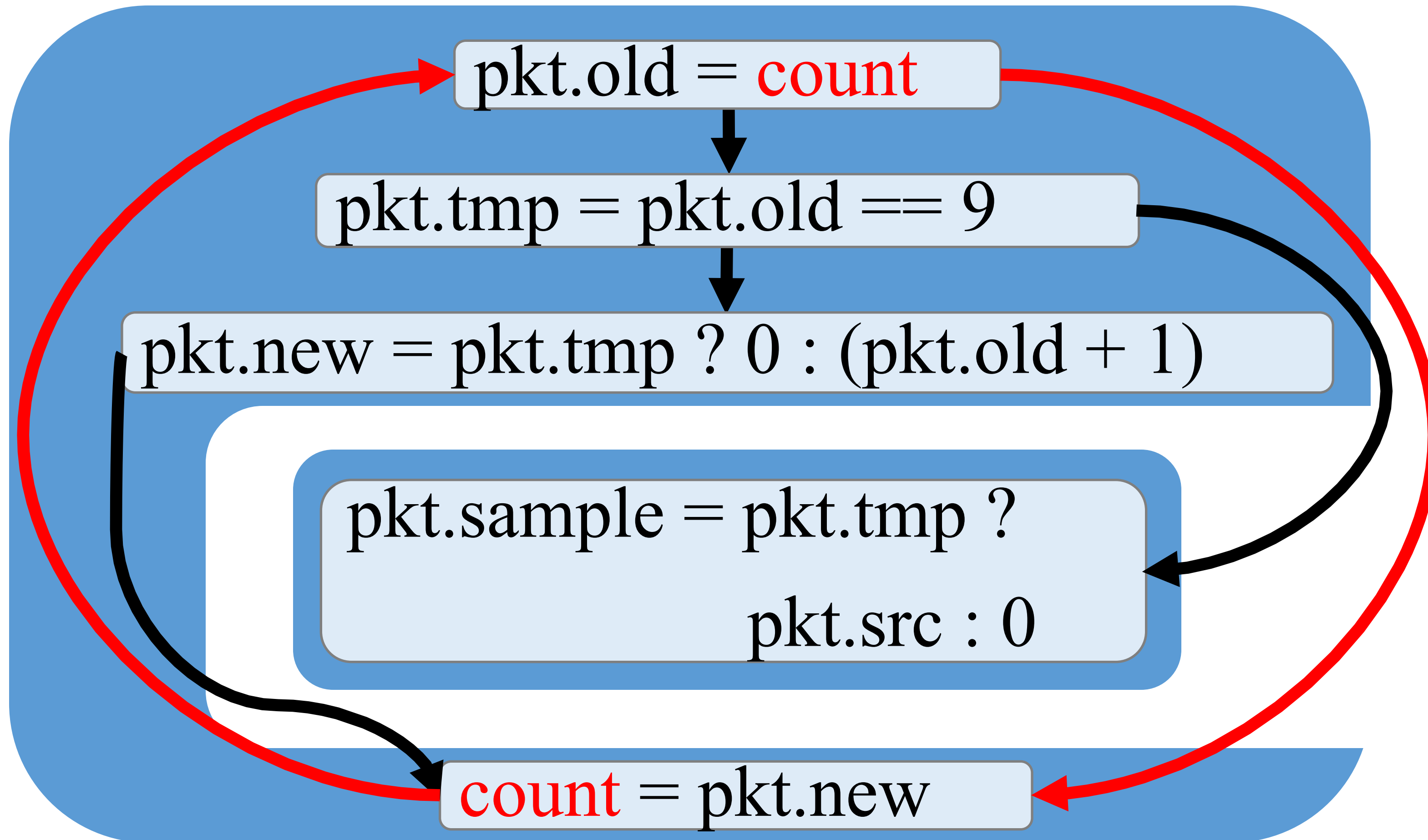
Packet field dependencies

Sequential to pipelined code



State dependencies

Sequential to pipelined code



Strongly connected components

Sequential to pipelined code

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

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

Condensed DAG

Sequential to pipelined code

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

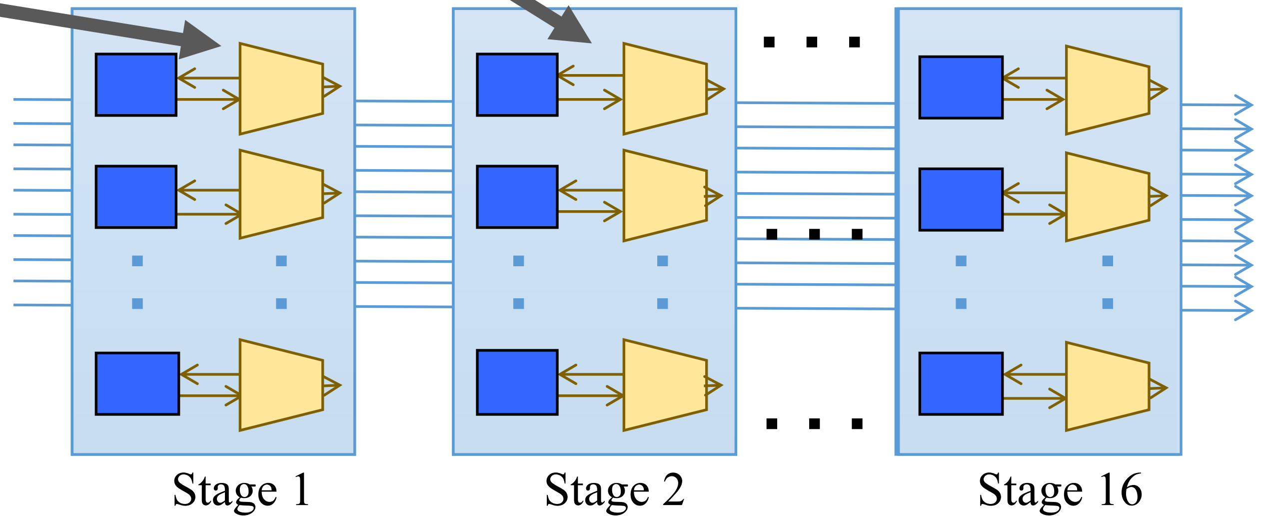
Hardware constraints

Stage 1

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

Stage 2

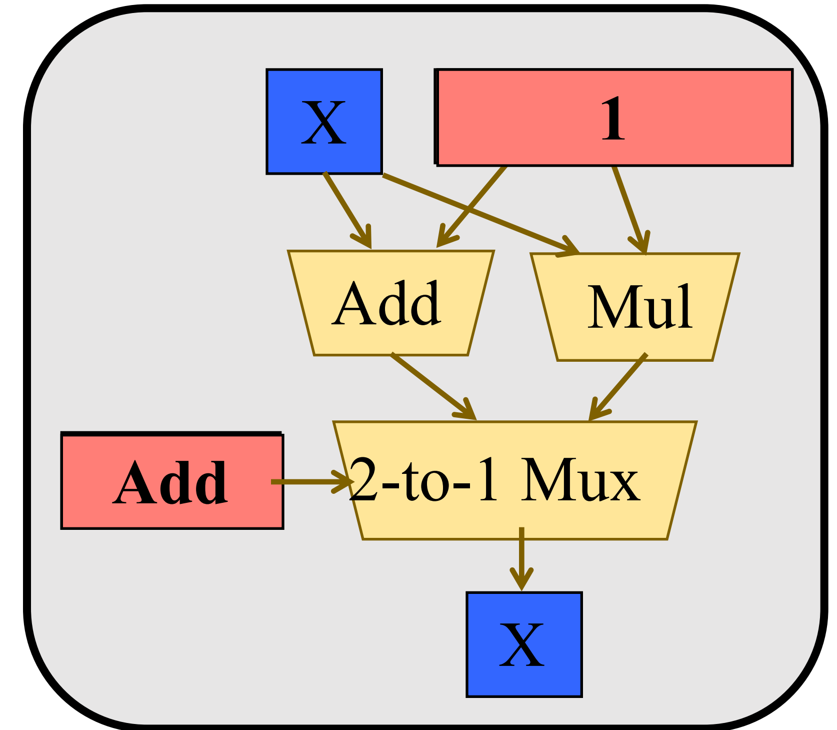
```
pkt.sample = pkt.tmp ?
             pkt.src : 0
```



Hardware constraints: example

x = x + 1 maps to this atom

x = x * x doesn't map



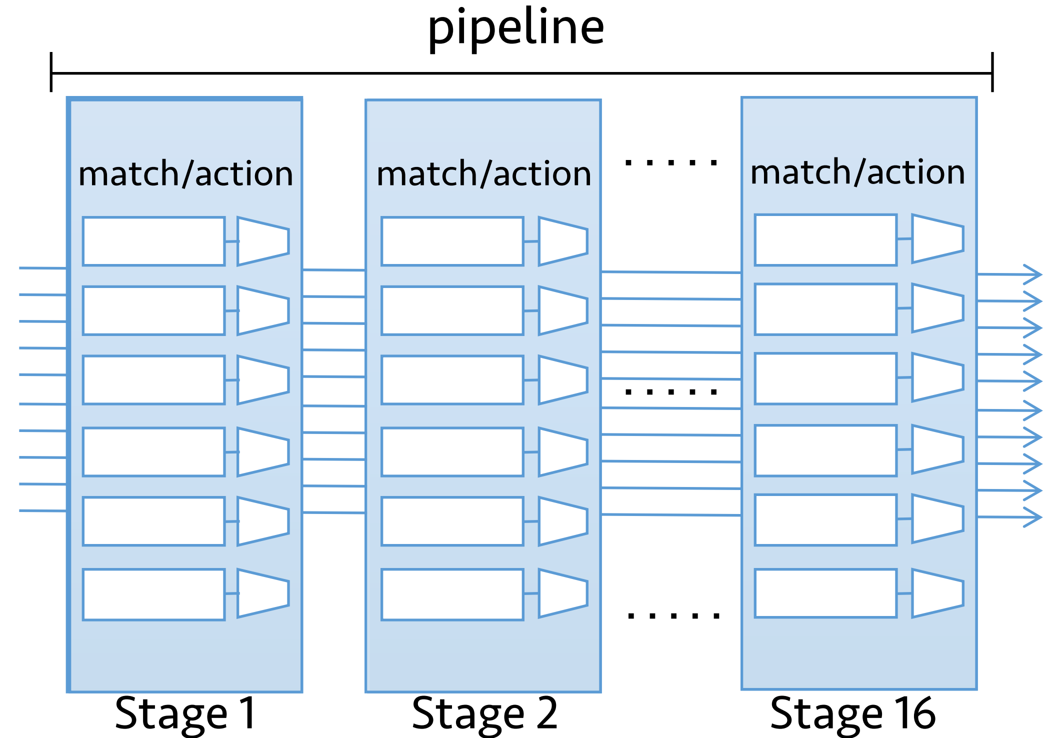
- Determines if algorithm can/cannot run at line rate

Our work

Packet transaction in Domino

```
For each packet
    Calculate average queue size
    if min < avg < max
        calculate probability p
        mark packet with probability p
    else if avg > max
        mark packet
```

Compiler

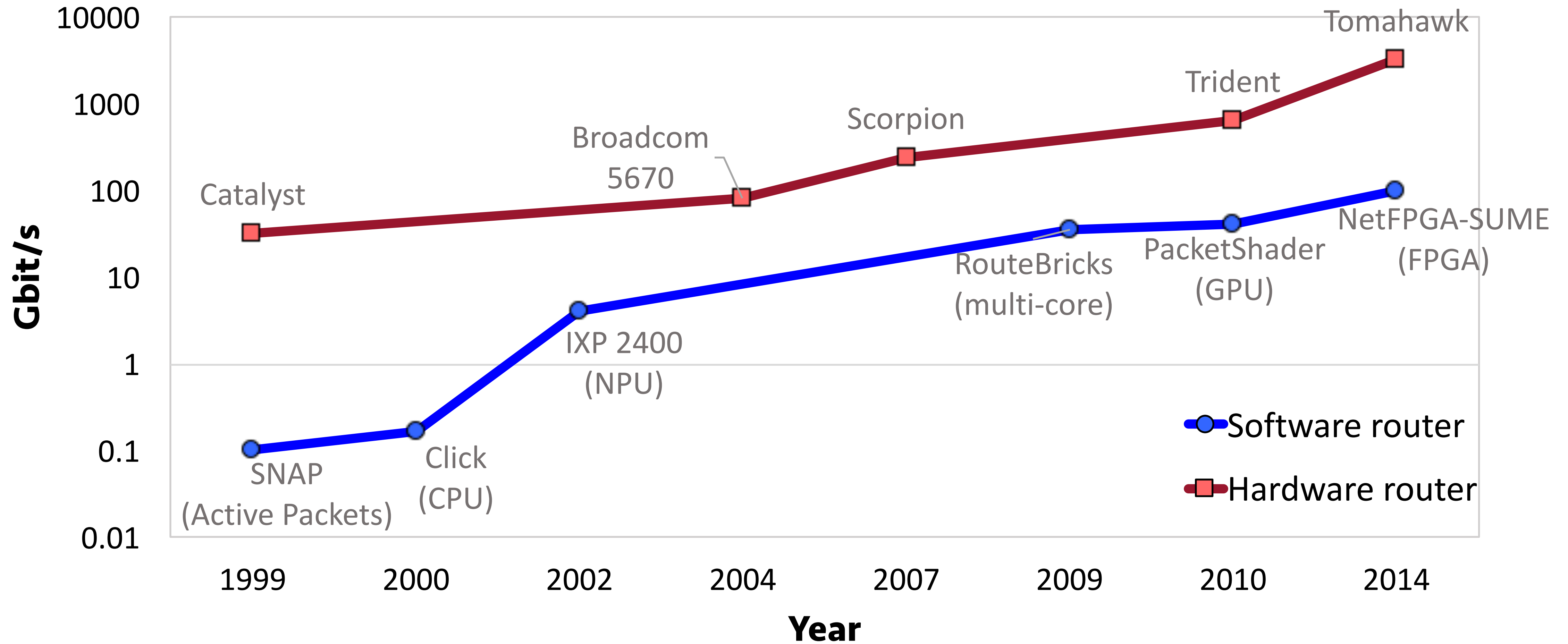


Program in imperative DSL, compile to run at line-rate

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
  - Determines which algorithms run at line rate

Software vs. hardware routers



Software routers (CPUs, NPUs, GPUs, multi-core, FPGA) lose 10—100x performance

Stateful atoms for programmable routers

Read/Write (R/W) (Bloom Filters)

```
pkt.f1 = x;
x = (pkt.f2 | constant);
```

ReadAddWrite (RAW) (Sketches)

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

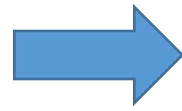
Predicated ReadAddWrite (PRAW) (RCP)

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

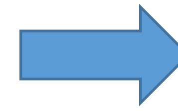
Language constraints on Domino

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

Canonicalization



Sequential to parallel code



Hardware constraints

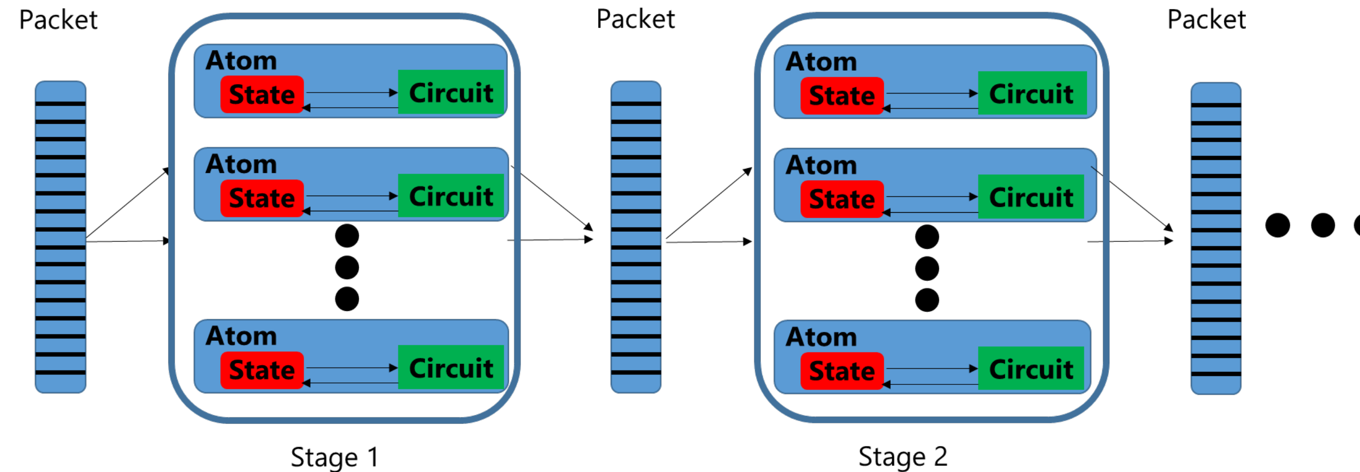
Instruction mapping: bin packing

Stage 1

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

Stage 2

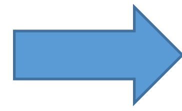
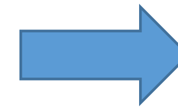
```
pkt.sample = pkt.tmp;
```



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.

Canonicalization  Sequential to parallel code  Hardware constraints

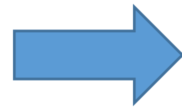
Instruction mapping: the SKETCH algorithm

- 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.

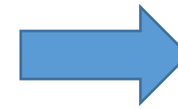
  for all inputs,

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

Canonicalization



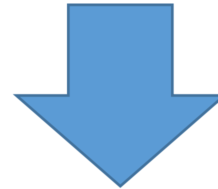
Sequential to parallel code



Hardware constraints

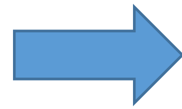
Static Single-Assignment

```
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 ;
```

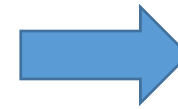


```
pkt.id0 = hash2(pkt.sport, pkt.dport) % NUM_FLOWLETS;
pkt.last_time0 = last_time[pkt.id0];
...
pkt.last_time1 = pkt.arrival;
…
last_time [pkt.id0] = pkt.last_time1 ;
```

Canonicalization



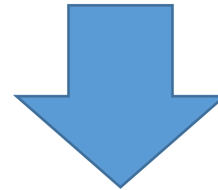
Sequential to parallel code



Hardware constraints

Expression Flattening

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



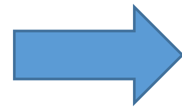
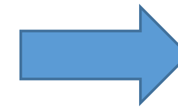
```
pkt.tmp   = pkt.arrival - last_time[pkt.id];
pkt.tmp2 = pkt.tmp > THRESHOLD;
saved_hop [ pkt . id ] = pkt.tmp2
                         ? pkt . new_hop
                         : saved_hop [ pkt . id ];
```

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

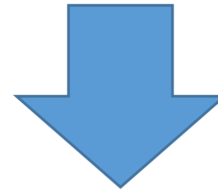
Relationship to prior compiler techniques

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

Canonicalization  Sequential to parallel code  Hardware constraints

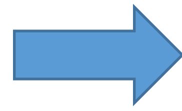
Branch Removal

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

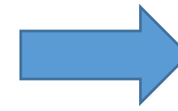


```
pkt.tmp = pkt.arrival - last_time[pkt.id] > THRESHOLD;
saved_hop [ pkt . id ] = pkt.tmp
                         ? pkt . new_hop
                         : saved_hop [ pkt . id ];
```

Canonicalization



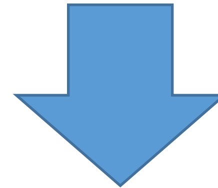
Sequential to parallel code



Hardware constraints

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
```

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.