Packet Transactions: Programming the Data Plane at Line Rate

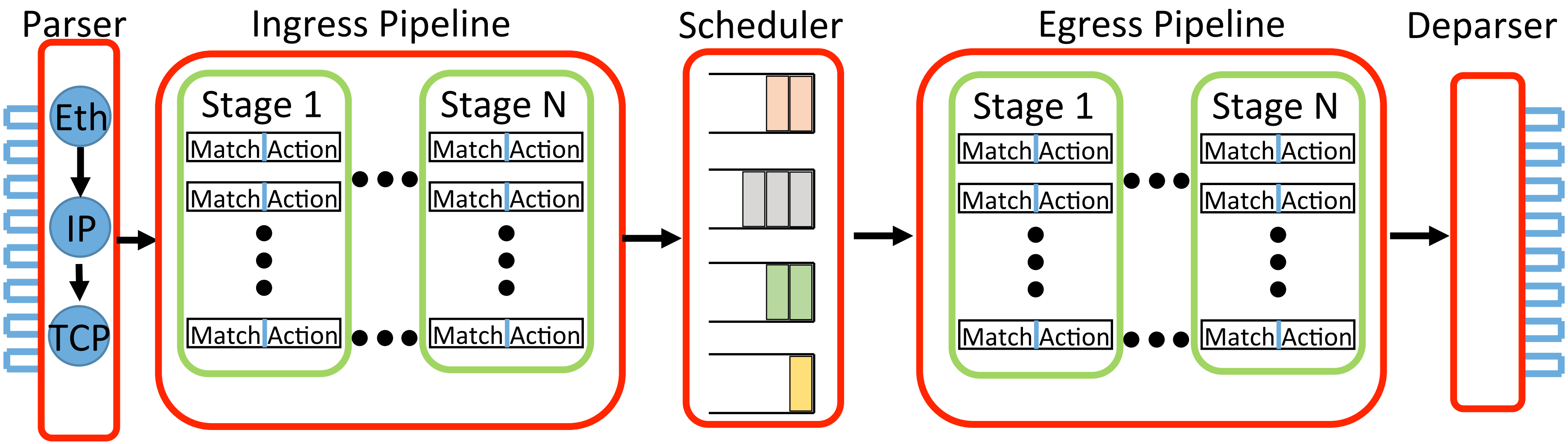
Anirudh Sivaraman, Mihai Budiu, Alvin Cheung, Changhoon Kim, Steve Licking, George Varghese, Hari Balakrishnan, Mohammad Alizadeh, Nick McKeown

Massachusetts Institute of Technology
BAREFOOT NETWORKS
UNIVERSITY of WASHINGTON SEATTLE
Microsoft Research
Stanford University

Programming the data-plane at line rate

- Programmable: Can we express a new data-plane algorithm?
- Line-rate: Highest capacity supported by a communication standard

Programmability at line-rate



- OpenFlow: Match-Action interface, fixed fields, fixed actions
- P4, RMT, FlexPipe, Xpliant: Protocol-independent match-action pipeline.

Isn’t P4 sufficient?

- Match-action is perfect for forwarding
- But, limiting for stateful algorithms
- Example: RED:

```
On enqueue:
    Calculate average queue size
     if min < avg < max
        calculate probability p
         mark packet with probability p
     else if avg > max:
          mark packet
```

Packet Transactions

- Imperative code block in subset of C (domino) that is atomic and isolated from other such blocks
- One packet transaction per pipeline
- More familiar to NPU, Click programmers

Programming with Packet Transactions

Domino

```c
#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];
}
```

P4

Stage 1

```
pkt.new_hop = hash3(pkt.sport,
                    pkt.dport,
pkt.arrival)
                    %NUM_HOPS;
```

```
pkt.id = hash2(pkt.sport, pkt.dport)
         % NUM_FLOWLETS
```

Stage 2

```
pkt.last_time = last_time[pkt.id];
last_time[pkt.id] = pkt.arrival;
```

Stage 3

```
pkt.tmp = pkt.arrival – pkt.last_time;
```

Stage 4

```
pkt.tmp2 = pkt.tmp > 5;
```

Stage 5

```
pkt.saved_hop = saved_hop[pkt.id];
saved_hop[pkt.id] = pkt.tmp2 ?
          pkt.new_hop :
          pkt.saved_hop;
```

Stage 6

```
pkt.next_hop = pkt.tmp2 ?
        pkt.new_hop :
        pkt.saved_hop ;
```

Language constraints on domino

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

If Conversion

```
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 ];
```

Read and Write Flanks

```
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
```

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 ;
```

Critical Path Scheduling

pkt.id =
hash2(pkt.sport,
pkt.dport)
% NUM_FLOWLETS

pkt.last_time = last_time[pkt.id]

pkt.tmp = pkt.arrival –
pkt.last_time

last_time[pkt.id] = pkt.arrival

pkt.tmp2 = pkt.tmp >
THRESHOLD

pkt.next_hop =
hash3(pkt.sport,
pkt.dport, pkt.arrival)
%NUM_HOPS

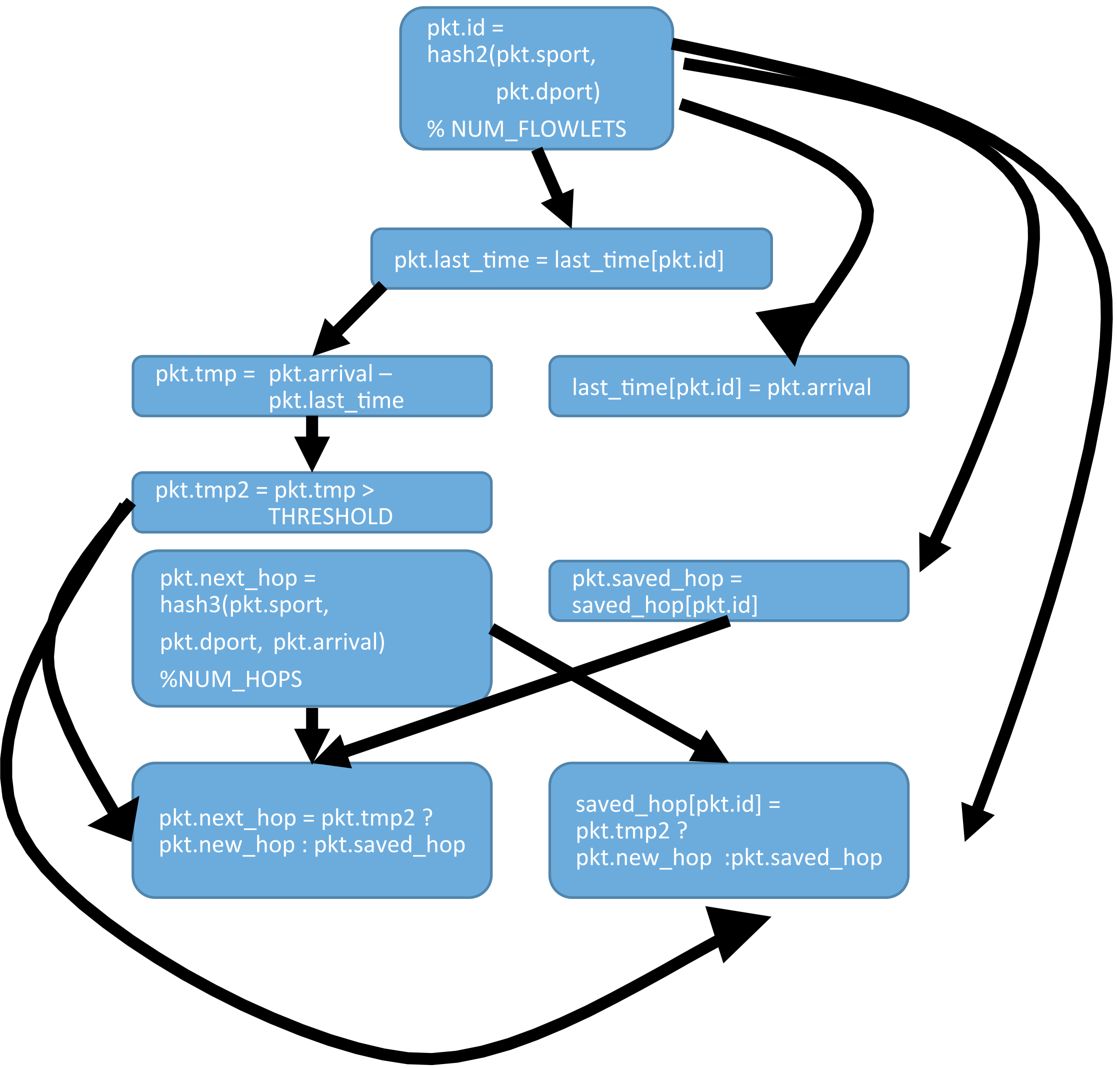
pkt.saved_hop =
saved_hop[pkt.id]

pkt.next_hop = pkt.tmp2 ?
pkt.new_hop : pkt.saved_hop

saved_hop[pkt.id] =
pkt.tmp2 ?
pkt.new_hop :pkt.saved_hop

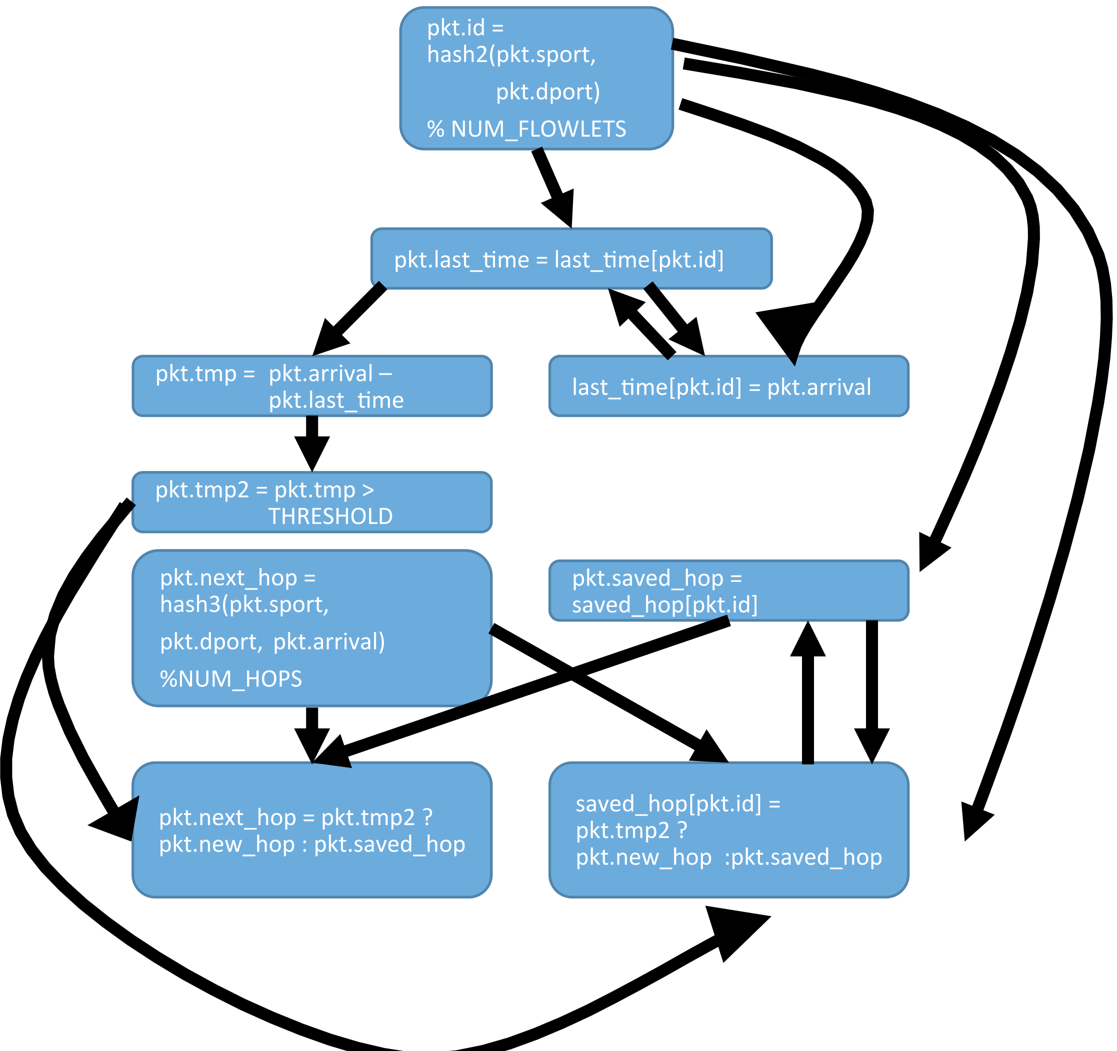
Create one node for each instruction.

Critical Path Scheduling



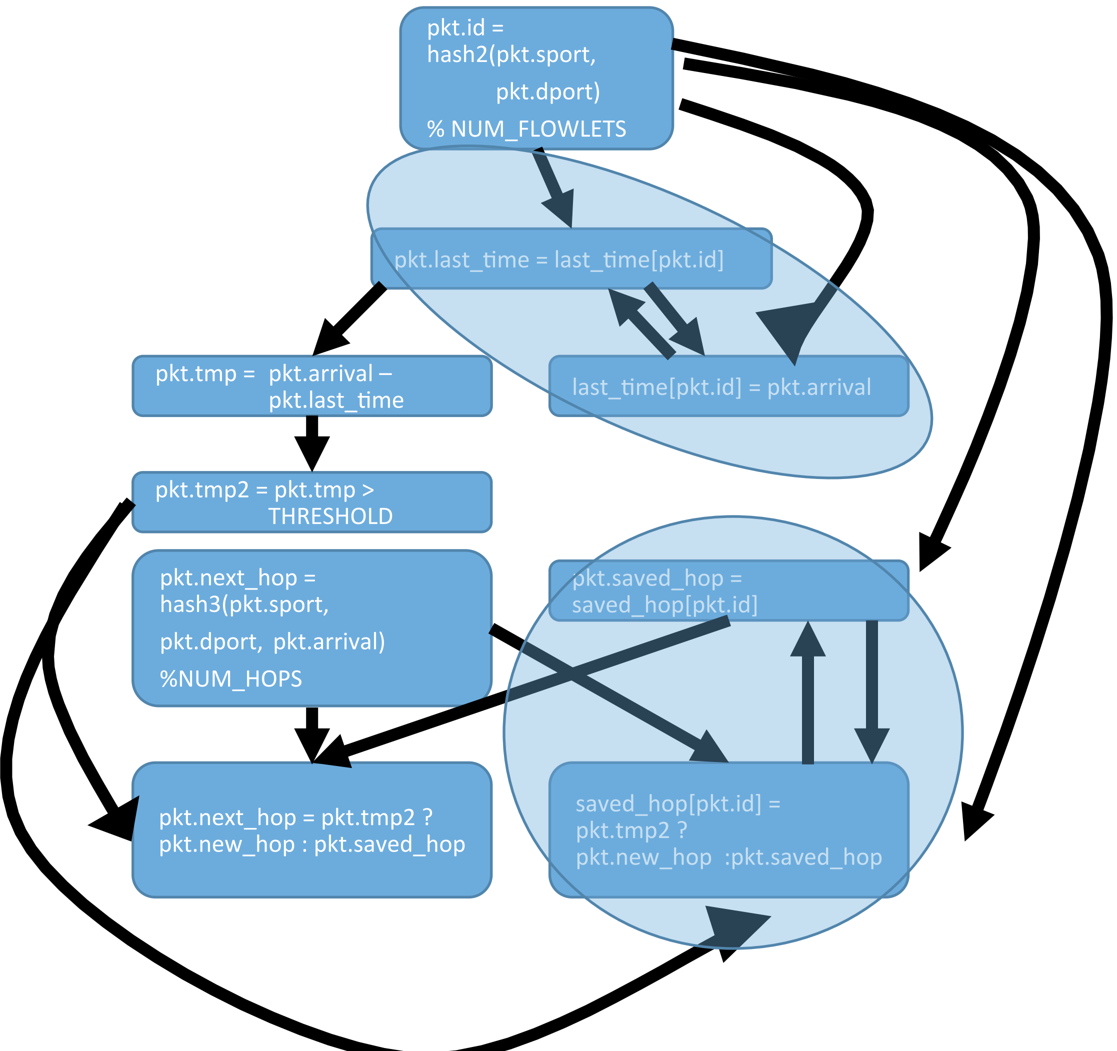
Add ReadAfterWrite dependencies

Critical Path Scheduling



Pair up read and write flanks

Critical Path Scheduling



Condense Strongly Connected Components

Critical Path Scheduling

Stage 1

```
pkt.new_hop = hash3(pkt.sport,
                    pkt.dport,
pkt.arrival)
                       %NUM_HOPS;
```

```
pkt.id = hash2(pkt.sport, pkt.dport)
            % NUM_FLOWLETS
```

Stage 2

```
pkt.last_time = last_time[pkt.id];
last_time[pkt.id] = pkt.arrival;
```

Stage 3

```
pkt.tmp = pkt.arrival – pkt.last_time;
```

Stage 4

```
pkt.tmp2 = pkt.tmp > 5;
```

Stage 5

```
pkt.saved_hop = saved_hop[pkt.id];
saved_hop[pkt.id] = pkt.tmp2 ?
                 pkt.new_hop :
                 pkt.saved_hop;
```

Stage 6

```
pkt.next_hop = pkt.tmp2 ?
            pkt.new_hop :
            pkt.saved_hop ;
```

Schedule condensed graph

Generating P4 code

- Required changes to P4
  - Sequential execution semantics
  - Expression support
  - Both available in v1.1
- Encapsulate every SCC in a default action
- Need sequential execution for stateful components
- Thanks to Antonin Bas for help with P4 code generation!

Checking feasibility at line rate

- So far, haven't constrained action bodies
- Check if action bodies can be mapped to available hardware
- If it can, declare that we can run it at line rate.
  - x = x + 1 can be mapped to x = x + c
- If we can't, flag a compiler error E.g.
  - x = (pkt.y) ? (x + 1) : x can't be mapped to x = x + c;

Closing thoughts

- Constructive proof that we could run a subset of C at line rate
- More familiar abstraction for stateful algorithms (37 LOC for flowlet switching in domino vs 110 in P4)
- Vehicle for higher-level abstractions in packet processing
- Will be open sourcing code soon
- Come check out the demo!