Programmable Packet Scheduling at Line Rate

Sachin Katti
Anirudh Sivaraman, Stanford, MIT, Cisco, Barefoot

(to appear in ACM SIGCOMM 2016)
Programmable switching chips

Programming the packet scheduler is off-limits for today’s 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

\[ f = \text{flow}(\text{pkt}) \]
\[ p.\text{tmp} = T[f] + p.\text{len} \]
\[ ... \]
\[ ... \]
\[ p.\text{rank} = 2 \times p.\text{tmp} \]

(programmable)

PIFO Scheduler

(fixed logic)
A programmable scheduler

Parser → Ingress pipeline → Rank Computation → Egress pipeline → Deparser

Schedule

PIFO

Queues/Schedule
Weighted Fair Queuing

1. $f = \text{flow}(p)$
2. $p.\text{start} = \max(T[f].\text{finish}, \text{virtual\_time})$
3. $T[f].\text{finish} = p.\text{start} + p.\text{len} / p.w$
4. $p.\text{rank} = p.\text{start}$
Traffic Shaping

Ingress pipeline

1. tokens = min(tokens + rate * (now – last), burst)
2. p.send = now + max( (p.len – tokens) / rate, 0)
3. last = now
4. p.rank = p.send

PIFO Schedule

Egress pipeline

Deparser

Parser

Rank Computation

In

Out
pFabric (SRPT)
pFabric (SRPT)

Rank Computation
1. \( f = \text{flow}(p) \)
2. \( p.\text{rank} = f.\text{rem}\_\text{size} \)

PIFO Scheduler

[Diagram showing PIFO Scheduler with numbers 9, 8, 5, 2]
Beyond a single PIFO

Hierarchical Packet Fair Queuing

Hierarchical scheduling algorithms need hierarchy of PIFOs
Tree of PIFOs

Hierarchical Packet Fair Queuing

Root

Red (0.5)

a (0.99)
b (0.01)

Blue (0.5)

x (0.5)
y (0.5)

PIFO-root
(WFQ on Red & Blue)

PIFO-Red
(WFQ on a & b)

PIFO-Blue
(WFQ on x & y)
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

Enqueue: (logical PIFO, rank, flow)

Dequeue: (logical PIFO)
A PIFO mesh
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
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 visible to its parent?)
- E.g., HPFQ:
Hierarchical packet-fair queueing (HPFQ)

Composing PIFOs

PIFO-root
(WFQ on A and B)

PIFO-A
(WFQ on 1 and 2)

PIFO-B
(WFQ on 3 and 4)
Shortest remaining processing time

1. \( f = \text{flow}(p) \)
2. \( p.prio = f.\text{rem}_\text{size} \)
Weighted fair queuing

1. \( f = \text{flow}(p) \)
2. \( p.\text{start} = \max(T[f].\text{finish}, \text{virtual}\_\text{time}) \)
3. \( T[f].\text{finish} = p.\text{start} + \frac{p.\text{len}}{p.w} \)
4. \( p.\text{prio} = p.\text{start} \)

Ingress Pipeline

Scheduler

Push-In-First-Out (PIFO) Queue
Traffic Shaping

1. update tokens
2. p.send = now + (p.len - tokens) / rate;
3. p.prio = p.send

Ingress Pipeline

Scheduler

Push-In-First-Out (PIFO) Queue
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

Parser

Ingress pipeline

Queue/Scheduler

Egress pipeline

Deparser
Today: Packet scheduling is off-limits for programming

- 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:
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

Only sustains 100 M packets per second

- 1: route lookup
- 2: ACL lookup
- 3: tunnel lookup
- ...
- 10: ...

1 GHz processor
Packet-parallel architecture

Lookup table

1: route lookup
2: ACL lookup
3: tunnel lookup
...
10: ...

1 GHz processor

Packets
Packet-parallel architecture

Memory replication increases die area

1: route lookup
2: ACL lookup
3: tunnel lookup
10: ...

1 GHz processor

Packets

1 GHz processor

1 GHz processor

1 GHz processor
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., $\text{pkt.f4} = \text{pkt.f1} + \text{pkt.f2} - \text{pkt.f3}$
  - 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
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

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

```
p1
p2
```

```
p10
```

```
p1.sample = 0
p2.sample = 0
p10.sample = 1
```
if (count == 9):
    pkt.sample = 1
    count = 0
else:
    pkt.sample = 0
    count++
The compiler

Packet Transactions

- Preprocessing
  - Simplify sequential code

- Code Pipelining
  - Sequential to parallel code

- Instruction Mapping
  - Respecting hardware constraints

Processing Pipeline
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;
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.
pkt.old = count

pkt.tmp = pkt.old == 9

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

pkt.sample = pkt.tmp

count = pkt.new

Packet field dependencies
pkt.old = count

pkt.tmp = pkt.old == 9

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

pkt.sample = pkt.tmp

count = pkt.new

State dependencies
pkt.old = count

pkt.tmp = pkt.old == 9

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

pkt.sample = pkt.tmp

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

\[
\begin{align*}
pkt.\text{old} &= \text{count}; \\
pkt.\text{tmp} &= pkt.\text{old} == 9; \\
pkt.\text{new} &= pkt.\text{tmp} \ ? \ 0 : (pkt.\text{old} + 1); \\
\text{count} &= pkt.\text{new};
\end{align*}
\]

Stage 2

\[
\begin{align*}
pkt.\text{sample} &= pkt.\text{tmp};
\end{align*}
\]
**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;

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

pkt.sample = pkt.tmp;
Instruction mapping: example

\[ x = x + 1 \] maps to this atom
\[ x = x \times x \] 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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom filter</td>
<td>29</td>
</tr>
<tr>
<td>Heavy hitter detection</td>
<td>35</td>
</tr>
<tr>
<td>Rate-Control Protocol</td>
<td>23</td>
</tr>
<tr>
<td>Flowlet switching</td>
<td>37</td>
</tr>
<tr>
<td>Sampled NetFlow</td>
<td>18</td>
</tr>
<tr>
<td>HULL</td>
<td>26</td>
</tr>
<tr>
<td>Adaptive Virtual Queue</td>
<td>36</td>
</tr>
<tr>
<td>CONGA</td>
<td>32</td>
</tr>
<tr>
<td>CoDel</td>
<td>57</td>
</tr>
</tbody>
</table>
### Expressiveness of packet transactions

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>LOC</th>
<th>Auto-generated P4 LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom filter</td>
<td>29</td>
<td>104</td>
</tr>
<tr>
<td>Heavy hitter detection</td>
<td>35</td>
<td>192</td>
</tr>
<tr>
<td>Rate-Control Protocol</td>
<td>23</td>
<td>75</td>
</tr>
<tr>
<td>Flowlet switching</td>
<td>37</td>
<td>107</td>
</tr>
<tr>
<td>Sampled NetFlow</td>
<td>18</td>
<td>70</td>
</tr>
<tr>
<td>HULL</td>
<td>26</td>
<td>95</td>
</tr>
<tr>
<td>Adaptive Virtual Queue</td>
<td>36</td>
<td>147</td>
</tr>
<tr>
<td>CONGA</td>
<td>32</td>
<td>89</td>
</tr>
<tr>
<td>CoDel</td>
<td>57</td>
<td>271</td>
</tr>
</tbody>
</table>
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

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

Stateful

Read/Write (R/W)

\[ x = (\text{pkt.f1} l \text{constant}); \]

ReadAddWrite (RAW)

\[ x = (x l 0) + (\text{pkt.f1} l \text{constant}); \]

Predicated ReadAddWrite (PRAW)

\[ \text{if (predicate(x, pkt.f1, pkt.f2))} \]
\[ x = (x l 0) + (\text{pkt.f1} l \text{pkt.f2 l constant}); \]
\[ \text{else:} \]
\[ x = x \]
Atoms used in targets

Stateless

pkt.f3 = (pkt.f1 \text{ l constant})
\text{OP} \subseteq (pkt.f2 \text{ l constant});
where
\text{OP} = \{+, -, \text{AND}, \text{OR}, >, <, \ldots\}

Stateful

<table>
<thead>
<tr>
<th>Atom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>Read or write state</td>
</tr>
<tr>
<td>RAW</td>
<td>Read, add, and write back</td>
</tr>
<tr>
<td>PRAW</td>
<td>Predicated version of RAW</td>
</tr>
<tr>
<td>IfElseRAWW</td>
<td>2 RAWs, one each when a predicate is true or false</td>
</tr>
<tr>
<td>Sub</td>
<td>IfElseRAW with a stateful subtraction capability</td>
</tr>
<tr>
<td>Nested</td>
<td>4-way predication (nests 2 IfElseRAWs)</td>
</tr>
<tr>
<td>Pairs</td>
<td>Update a pair of state variables</td>
</tr>
</tbody>
</table>
### Stateless (0.22 %)

\[
\text{pkt.f3} = (\text{pkt.f1} \mid \text{constant}) \\
\text{OP} \\
(\text{pkt.f2} \mid \text{constant});
\]

where
\[
\text{OP} = \{+, -, \text{AND}, \text{OR}, >, <, \ldots\}
\]

### Atoms used in targets

<table>
<thead>
<tr>
<th>Atom</th>
<th>Description</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>Read or write state</td>
<td>0.04%</td>
</tr>
<tr>
<td>RAW</td>
<td>Read, add, and write back</td>
<td>0.07%</td>
</tr>
<tr>
<td>PRAW</td>
<td>Predicated version of RAW</td>
<td>0.13%</td>
</tr>
<tr>
<td>IfElseRAW W</td>
<td>2 RAWs, one each when a predicate is true or false</td>
<td>0.16%</td>
</tr>
<tr>
<td>Sub</td>
<td>IfElseRAW with a stateful subtraction capability</td>
<td>0.24%</td>
</tr>
<tr>
<td>Nested</td>
<td>4-way predication (nests 2 IfElseRAWS)</td>
<td>0.58%</td>
</tr>
<tr>
<td>Pairs</td>
<td>Update a pair of state variables</td>
<td>0.96%</td>
</tr>
</tbody>
</table>
## Compiling packet transactions

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

Stage 1

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

Stage 2

pkt.sample = pkt.tmp;

count = pkt.new;
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

1. update tokens
2. \( p.\text{send} = \text{now} + \frac{(p.\text{len} - \text{tokens})}{\text{rate}}; \)
3. \( p.\text{prio} = p.\text{send} \)

Ingress Pipeline

Scheduler

Push-In-First-Out (PIFO) Queue
LSTF

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

Scheduler

Ingress Pipeline
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
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

\[ pkt\text{.tmp} = pkt\text{.arrival} - \text{last\_time}[pkt\text{.id}] > \text{THRESHOLD}; \]
\[ \text{saved\_hop}[\ pkt\ .\ id\ ] = \ pkt\text{.tmp} \]
\[ ? \ pkt\ .\ new\_hop \]
\[ : \text{saved\_hop}[\ pkt\ .\ id\ ]; \]

\[ pkt\text{.tmp} = pkt\text{.arrival} - \text{last\_time}[pkt\text{.id}] ; \]
\[ pkt\text{.tmp2} = pkt\text{.tmp} > \text{THRESHOLD} ; \]
\[ \text{saved\_hop}[\ pkt\ .\ id\ ] = \ pkt\text{.tmp2} \]
\[ ? \ pkt\ .\ new\_hop \]
\[ : \text{saved\_hop}[\ pkt\ .\ id\ ]; \]
- Generic method to handle fairly complex templates

- Templates determine if a Domino program can run at line rate.

- Example results:
  - Flowlet switching needs conditional execution to save next hop information:
    
    ```
    ```
  - Simple increment suffices for heavy hitter 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

<table>
<thead>
<tr>
<th>Technique</th>
<th>Prior work</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>If Conversion</td>
<td>Kennedy et al. 1983</td>
<td>No breaks, continue, gotos, loops</td>
</tr>
<tr>
<td>Static Single-Assignment</td>
<td>Ferrante et al. 1988</td>
<td>No branches</td>
</tr>
<tr>
<td>Strongly Connected Components</td>
<td>Lam et al. 1989 (Software Pipelining)</td>
<td>Scheduling in space instead of time</td>
</tr>
<tr>
<td>Synthesis for instruction mapping</td>
<td>Technology mapping</td>
<td>Map to 1 hardware primitive, not multiple</td>
</tr>
<tr>
<td>Superoptimization</td>
<td></td>
<td>Counter-example-guided, not brute force</td>
</tr>
</tbody>
</table>
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
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];
pkt.id = hash2(pkt.sport, pkt.dport) % NUM_FLOWLETS;
...
lsttme[pkt.id] = pkt.arrival;
...

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

 pkt.last_time = last_time[pkt.id]

 pkt.tmp = pkt.arrival - pkt.last_time

 pkt.tmp2 = pkt.tmp > THRESHOLD

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

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

 pkt.saved_hop = saved_hop[pkt.id]

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

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

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

 Pair up read/write flanks
Code Pipelining

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

 pkt.last_time = last_time[pkt.id]

 pkt.tmp = pkt.arrival - pkt.last_time

 pkt.tmp2 = pkt.tmp > THRESHOLD

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

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

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

 pkt.next_hop = pkt.tmp2

 Condense strongly connected components into codelets
pkt.id = hash2(pkt.sport, pkt.dport) % NUM_FLOWLETS

pkt.last_time = last_time[pkt.id]

pkt.tmp = pkt.arrival - pkt.last_time

pkt.tmp2 = pkt.tmp > THRESHOLD

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

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

pkt.saved_hop = saved_hop[pkt.id]

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

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

Add packet-field dependencies
#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];
}
The Domino compiler

- Canonicalization (Sequential Code)
  - Branch Removal
  - Handle state variables

- Code Pipelining
  - Sequential to parallel code

- Instruction Mapping
  - Respecting hardware constraints
Diagrams
Add \[ \text{constant} \]

Sub

\[ x \]

choice

2-to-1 Mux

\[ x \]
The quest for programmability

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

<table>
<thead>
<tr>
<th>Switch</th>
<th>Year</th>
<th>Line-rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cisco Catalyst</td>
<td>1999</td>
<td>32 Gbit/s</td>
</tr>
<tr>
<td>Broadcom 5670</td>
<td>2004</td>
<td>80 Gbit/s</td>
</tr>
<tr>
<td>Broadcom Scorpion</td>
<td>2007</td>
<td>240 Gbit/s</td>
</tr>
<tr>
<td>Broadcom Trident</td>
<td>2010</td>
<td>640 Gbit/s</td>
</tr>
<tr>
<td>Broadcom Tomahawk</td>
<td>2014</td>
<td>3.2 Tbit/s</td>
</tr>
</tbody>
</table>

Programmability => 10--100x slower than line rate.
The quest for programmability

Programmability => 10--100x slower than line rate.
Compiler targets: diagram

Operation:
+,
-, >, <,
AND, OR

pkt.f1/constant → Operation: +, -, >, <, AND, OR → pkt.f3

pkt.f2/constant → pkt.f

constant → 2-to-1 Mux → x

pkt.f → 2-to-1 Mux → x

constant → 2-to-1 Mux → x

 pkt.f

constant

0

2-to-1 Mux

2-to-1 Mux

Adder

x

101