Programming Line-Rate Switches

Anirudh Sivaraman

Massachusetts Institute of Technology
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

pipeline

match/action

Stage 1

match/action

Stage 2

match/action

Stage 16

...
A machine model for line-rate switches

Packet Header

Stage 1

Stage 2

Stage 16

Stage 1

Stage 2

Stage 16
A machine model for line-rate switches

stage 1

stage 2

stage 16

Typical requirement: 1 pkt / nanosecond
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: \( \text{pkt.f4} = \text{pkt.f1} + \text{pkt.f2} - \text{pkt.f3} \)

Can pipeline stateless operations
Stateless vs. stateful operations

Stateful operation: \( x = x + 1 \)

\( X = 0 \)

\( X \) should be 2, not 1!
Stateless vs. stateful operations

Stateful operation: \( x = x + 1 \)

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

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

```
p1.sample = 0
p2.sample = 0
```

```
p10.sample = 1.2.3.4
```
Packet transactions are expressive

<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>
Compiling packet transactions

Packet Sampling Algorithm

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

Packet Sampling 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`

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

 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
(1) Serial code to codelet pipeline

\[
\begin{align*}
\text{pkt.old} &= \text{count} \\
\text{pkt.tmp} &= \text{pkt.old} == 9 \\
\text{pkt.new} &= \text{pkt.tmp} ? 0 : (\text{pkt.old} + 1) \\
\text{pkt.sample} &= \text{pkt.tmp} ? \text{pkt.src} : 0 \\
\text{count} &= \text{pkt.new}
\end{align*}
\]
(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
```
(1) Serial code to codelet pipeline

```
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
(1) Serial code to codelet pipeline

**Stage 1**

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

**Stage 2**

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

**Code pipelining**
(2) Codelets to atoms

Stage 1

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

Stage 2

\[
\text{pkt.sample} = \text{pkt.tmp} \ ? \text{pkt.src} : 0
\]

Assign each codelet to one atom. Reject if you run out of atoms.
(2) Codelets to atoms

\[ x = x \times x \] doesn’t map, reject code

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

Use program synthesis for mapping problem.

Determines if algorithm can run at line rate
The compiler as a tool for switch design

Algorithm doesn't compile?
- Modify pipeline geometry or atom.
- Algorithm compiles
- Move on to another algorithm
Demo
## Stateful atoms for programmable switches

<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>IfElseRAW</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>
Compiling packet transactions to atoms

<table>
<thead>
<tr>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom filter</td>
</tr>
<tr>
<td>Heavy hitter detection</td>
</tr>
<tr>
<td>Rate-Control Protocol</td>
</tr>
<tr>
<td>Flowlet switching</td>
</tr>
<tr>
<td>Sampled NetFlow</td>
</tr>
<tr>
<td>HULL</td>
</tr>
<tr>
<td>Adaptive Virtual Queue</td>
</tr>
<tr>
<td>CONGA</td>
</tr>
<tr>
<td>CoDel</td>
</tr>
</tbody>
</table>
## Compiling packet transactions to atoms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Most expressive stateful atom required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom filter</td>
<td>R/W</td>
</tr>
<tr>
<td>Heavy hitter detection</td>
<td>RAW</td>
</tr>
<tr>
<td>Rate-Control Protocol</td>
<td>PRAW</td>
</tr>
<tr>
<td>Flowlet switching</td>
<td>PRAW</td>
</tr>
<tr>
<td>Sampled NetFlow</td>
<td>IfElseRAW</td>
</tr>
<tr>
<td>HULL</td>
<td>Sub</td>
</tr>
<tr>
<td>Adaptive Virtual Queue</td>
<td>Nested</td>
</tr>
<tr>
<td>CONGA</td>
<td>Pairs</td>
</tr>
<tr>
<td>CoDel</td>
<td><strong>Doesn’t map</strong></td>
</tr>
</tbody>
</table>
Compiling packet transactions to atoms

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

~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.

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

<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?

Programmable logic to decide order or time
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

Rank Computation

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

(programmable)

PIFO Scheduler

9 8 5 2

(fixed logic)
A programmable scheduler

Parser
Ingress pipeline
Rank Computation
... ...
PIFO Scheduler
Egress pipeline
Deparser

In
Out

Rank computation is a packet transaction
Fair queuing

Rank Computation

1. f = flow(p)
2. p.start = max(T[f].finish, virtual_time)
3. T[f].finish = p.start + p.len
4. p.rank = p.start

In

Parser

Ingress pipeline

PIFO Scheduler

Egress pipeline

Deparser

Out
Token bucket shaping

Rank Computation
1. tokens = min(
   tokens + rate * (now – last),
   burst)
2. p.send = now +
   max( (p.len – tokens) / rate, 0)
3. tokens = tokens - p.len
4. last = now
5. p.rank = p.send
Shortest remaining flow size

Parser | Ingress pipeline | PIFO Scheduler | Egress pipeline | Deparser

In | Out

Queues/Scheduler

Ingress pipeline

Egress pipeline
Shortest remaining flow size

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

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, 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

Flow Scheduler (flip-flops)

A 0  B 1  C 3

Rank Store (SRAM)

A 2 3
B 2 4
C 4 5
D

Dequeue

Enqueue

A 2
B 4
C 5
D 4
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 mm² area for 5-level programmable hierarchical scheduler
  - < 4% relative to a 200 mm² 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.
<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></td>
<td>Superoptimization</td>
<td>Counter-example-guided, not brute force</td>
</tr>
</tbody>
</table>
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

```c
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;
```

```c
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

\[ \text{pkt.tmp} = \text{pkt.arrival} - \text{last_time}[\text{pkt.id}] > \text{THRESHOLD}; \]
\[ \text{saved_hop}[\text{pkt.id}] = \text{pkt.tmp} \]
\[ \quad ? \text{pkt.new_hop} \]
\[ \quad : \text{saved_hop}[\text{pkt.id}]; \]

\[ \text{pkt.tmp} = \text{pkt.arrival} - \text{last_time}[\text{pkt.id}]; \]
\[ \text{pkt.tmp2} = \text{pkt.tmp} > \text{THRESHOLD}; \]
\[ \text{saved_hop}[\text{pkt.id}] = \text{pkt.tmp2} \]
\[ \quad ? \text{pkt.new_hop} \]
\[ \quad : \text{saved_hop}[\text{pkt.id}]; \]
Instruction mapping: results

- Generic method to handle fairly complex templates

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

- Example results:
  - Flowlet switching requires conditional execution to save next hop information:
    
    ```
    saved_hop[pkt.id] = pkt.tmp2 ? pkt.new_hop : saved_hop[pkt.id]
    ```
  - 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
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

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

Branch Removal

Handle state variables

Code Pipelining

Sequential to parallel code

Instruction Mapping

Respecting hardware constraints

Processing Pipeline
Code Pipelining

```plaintext
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.saved_hop = saved_hop[pkt.id]

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

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

Pair up read/write flanks

Sequential to parallel code

Hardware constraints

Canonicalization
```
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
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.saved_hop = saved_hop[pkt.id]

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

 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

 Add packet-field dependencies

 Sequential to parallel code

 Hardware constraints

 Canonicalization
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];
}
```

Pipeline

```
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;
```
The quest for programmability

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

<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>
</tbody>
</table>
The quest for programmability

Programmability $\Rightarrow$ 10--100x slower than line rate.
Compiler targets: diagram

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

pkt.f1/constant -> pkt.f3
pkt.f2/constant

pkt.f -> 2-to-1 Mux -> x
constant

pkt.f -> 2-to-1 Mux -> Adder -> x
constant

x -> 2-to-1 Mux -> Adder
0
```
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

Can’t build a 10 GHz processor!

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

match/action

10 GHz processor

Packets
Packet-parallel architecture

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

1 GHz processor

Packets
Packet-parallel architecture

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

Memory replication increases die area

1 GHz processor

Packets
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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>LOC</th>
<th>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>