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

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

Packet fields:
- p1
- p2
- p10

Persistent state:
- p1.sample = 0
- p2.sample = 0
- p10.sample = 1.2.3.4
Under the hood …
A machine model for line-rate switches

Packet Header

Stage 1

Stage 2

Stage 16

pipeline
A machine model for line-rate switches

pipeline

Typical requirement: 1 pkt / nanosecond
A machine model for line-rate switches
A machine model for line-rate switches

- **Atom:** smallest unit of atomic packet/state update

- **Stage 1**
  - State
  - Action unit

- **Stage 2**
  - X
  - Constant
  - Add
  - Mul
  - Choice
  - 2-to-1 Mux
  - X

- **Stage 16**
  - State
  - Action unit

- 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.
Compiling packet transactions

Packet Sampling Algorithm

```
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`
Designing programmable switches

Focus on stateful atoms, stateless operations are easily pipelined
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>

Least Expressive

Most Expressive
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>
## Compilation results

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>LOC</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>29</td>
<td>R/W</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Heavy hitter detection</td>
<td>35</td>
<td>RAW</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>Rate-Control Protocol</td>
<td>23</td>
<td>PRAW</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Flowlet switching</td>
<td>37</td>
<td>PRAW</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Sampled NetFlow</td>
<td>18</td>
<td>IfElseRAW</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>HULL</td>
<td>26</td>
<td>Sub</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Adaptive Virtual Queue</td>
<td>36</td>
<td>Nested</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>CONGA</td>
<td>32</td>
<td>Pairs</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>CoDel</td>
<td>57</td>
<td>Doesn’t map</td>
<td>15</td>
<td>3</td>
</tr>
</tbody>
</table>

~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.

<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
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
Sequential to pipelined code

- $\text{pkt.old} = \text{count}$
- $\text{pkt.tmp} = \text{pkt.old} == 9$
- $\text{pkt.new} = \text{pkt.tmp} \, ? \, 0 : (\text{pkt.old} + 1)$
- $\text{pkt.sample} = \text{pkt.tmp} \, ? \, \text{pkt.src} : 0$
- $\text{count} = \text{pkt.new}$

Packet field dependencies
Sequential to pipelined code

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

Strongly connected components
sequential to pipelined code

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

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

**Stage 2**

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

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

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)

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

ReadAddWrite (RAW) (Sketches)

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

Predicated ReadAddWrite (PRAW) (RCP)

\[
\text{if (predicate}(x, \ pkt.f1, \ pkt.f2)) \\
\quad x = (x \mid 0) + (pkt.f1 \mid pkt.f2 \mid \text{constant}); \\
\quad \text{else:} \\
\quad x = x
\]
Language constraints on Domino

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

\[ \text{pkt.tmp} = \text{pkt.arrival} - \text{last}_\text{time}[\text{pkt.id}] > \text{THRESHOLD}; \]
\[ \text{saved}_\text{hop}[\text{pkt.id}] = \text{pkt.tmp} \]
\[ \quad ? \text{pkt.new}_\text{hop} \]
\[ \quad : \text{saved}_\text{hop}[\text{pkt.id}]; \]
\[ \text{pkt.tmp} = \text{pkt.arrival} - \text{last}_\text{time}[\text{pkt.id}]; \]
\[ \text{pkt.tmp2} = \text{pkt.tmp} > \text{THRESHOLD}; \]
\[ \text{saved}_\text{hop}[\text{pkt.id}] = \text{pkt.tmp2} \]
\[ \quad ? \text{pkt.new}_\text{hop} \]
\[ \quad : \text{saved}_\text{hop}[\text{pkt.id}]; \]
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

<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>Superoptimization</td>
<td>Counter-example-guided, not brute force</td>
</tr>
</tbody>
</table>
Branch Removal

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

```c
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) % 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.