Use cases of Programmable Dataplane (P4)

ECE/CS598HPN

Radhika Mittal
Which paper(s) did you read?

• (A) BeauCoup: Network Monitoring

• (B) Elmo: Multicast

• (C) Both

• (D) Neither
Network Monitoring

• Most popular usecase of programmable dataplanes.

• Lots of recent papers!

• Key challenges:
  • Dealing with small amount of memory.
  • Ensuring high line rate (small processing capability, limited memory access)
  • Supporting a wide variety of queries.
BeauCoup: Answering many network traffic queries, one memory update at a time!

1[bo’ku] Adv. many, a lot.

Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, Jennifer Rexford

SIGCOMM 2020 New York City USA
Network traffic query

DDoS: Are there many Source IPs sending to one particular Destination IP?

Select $\text{DstIP}$ where $\text{distinct(SrcIP)} > 1000$

Key

Attribute

Threshold

Internet
Many network traffic queries

<table>
<thead>
<tr>
<th>Query</th>
<th>Key</th>
<th>Attribute</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDoS?</td>
<td>DstIP</td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>Worm?</td>
<td>SrcIP</td>
<td></td>
<td>300</td>
</tr>
<tr>
<td>Port Scan?</td>
<td>SrcIP, DstIP</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Different keys/attrs, need multiple data structures
Many network traffic queries

I have 42 queries

Run 42 data structures?

I can’t...

Spec for today’s commodity programmable switch:

- XX Tbps aggregated throughput
- YY MB data-plane memory
- Can only access ZZ bytes of memory per packet

(True for CPU, FPGA, etc., as well... Moore’s law!)
One memory update at a time?

• Constant memory update per packet, regardless of the number of queries?

• Game plan:
  1. Each query uses only $o(1)$ memory update per packet \textit{on average}
  2. Combine many different queries, on average uses $O(1)$
  3. Coordinate, \textit{at most} $O(1)$ per packet
Today’s talk

• Challenge:
  many queries, few memory updates

• Achieving $o(1)$ memory access:
  coupon collectors

• System design:
  query compiler + data plane program

• Evaluation
The coupon collector problem

- 4 different coupons, collect all of them
- Random draws
- How many total draws are required?
Naïve Approach

Query: Select $DstIP$ where $\text{distinct}(SrcIP) > 100$

- Map each $SrcIP$ to a coupon
  - How many total coupons?
  - How many do you need to collect?

- Issues with this approach:
  - Too much memory
  - Each packet results in a coupon collection.
    - Exceed $O(1)$ access when multiple such queries are combined.
BeauCoup coupon collector

\[ f(SrclP) \rightarrow \text{Coupon} \]

- Mapping
- Select \( DstIP \) where \( \text{distinct}(SrclP) > 100 \)
- Collect different coupons

Key: 162.249.4.107
Coupons: A, B, C, D

- \( f(10.0.1.15) \rightarrow \text{Coupon C} \)
- \( f(10.0.1.33) \rightarrow \text{Coupon B} \)
- \( f(10.0.1.15) \rightarrow \text{Coupon C} \)
- \( f(10.0.1.42) \rightarrow \text{No Coupon} \)
**BeauCoup** coupon collector

\[ f(SrcIP) \rightarrow \text{Coupon} \]

- Generalization: \((m, p, n)\)-coupon collector
- \(m \times p < 1\), most packets collect no coupon

Example:
\((m=8, p=1\%, n=4)\)

Given a new SrcIP, each coupon is drawn with probability 1%
System design

• Query compiler: finds coupon collector configurations
  • Stops near query thresholds, minimize error
  • Hardware limits (e.g., memory access limit)
  • Fairness across queries

• Data plane program: collect coupons into in-memory table
  • Simultaneously run many queries
  • At most one coupon per packet
  • Update queries on-the-fly
Query compiler

Query set
$Q = \{q_1, q_2, \ldots\}$

Total memory update limit: $\Gamma$ per packet

Per-query limit:
$\gamma_q$ per packet

$f(q) = \gamma_q = \Gamma / |Q|$ (fair allocation)

Compiler

$q_i$’s Collector Configuration
Total coupons: $m$
Each probability: $p$
Coupons to collect: $n$

Goal:
I. Stop near Threshold
II. Update limit $m*p \leq \gamma_q$
III. HW limit, e.g., $m \leq 32$. 
Query compiler

Query set
\[ Q = \{q_1, q_2, \ldots\} \]

Total coupons:
\[ m \]
Each probability:
\[ p \]
Coupons to collect:
\[ n \]

Threshold = 1000, \( \gamma_q = 0.01 \) (\( m = 20, p = 1/2048, n = 8 \))

Switch Data Plane
Stacking queries: same attribute

\[ q_1: f(\text{SrcIP}) \rightarrow \text{Coupon} \]
\[ m_1 = 4, \ p_1 = 1/8 \]
\[ q_2: f(\text{SrcIP}) \rightarrow \text{Coupon} \]
\[ m_2 = 3, \ p_2 = 1/16 \]

Hash function \( h_1(\text{SrcIP}) \rightarrow [0,1) \)

4 coupons for \( q_1 \)
3 coupons for \( q_2 \)
One hash function for each attribute

\[ q_1: f(SrcIP) \rightarrow \text{Coupon} \]
\[ m_1=4, p_1=1/8 \]

\[ q_6: g(DstIP) \rightarrow \text{Coupon} \]
\[ m_6=3, p_6=1/8 \]
TCAM for selecting a coupon

Match $h_A(SrcPort)$ | Query#,Coupon#
---|---
Match $h_B(DstPort)$ | Query#,Coupon#
Match $h_C(SrcIP)$ | Query#,Coupon#
Match $h_D(DstIP)$ | Query#,Coupon#

<table>
<thead>
<tr>
<th></th>
<th>Query#,Coupon#</th>
</tr>
</thead>
<tbody>
<tr>
<td>000*****</td>
<td>(6,1)</td>
</tr>
<tr>
<td>001*****</td>
<td>(6,2)</td>
</tr>
<tr>
<td>010*****</td>
<td>(6,3)</td>
</tr>
<tr>
<td>01101***</td>
<td>(8,1)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

$h_A(SrcPort)$\[1\] \xrightarrow{} 001\[2\] \rightarrow \text{No coupon}

$h_B(DstPort)$\[1\] \xrightarrow{} 010\[3\] \rightarrow \text{No coupon}

$h_C(SrcIP)$\[1\] = 1010111... \rightarrow \text{Collect coupon (q\_6, #3)}

$h_D(DstIP)$\[1\] = 0101011... \rightarrow \text{Random tiebreak if >1 coupons}

Packet
SrcPort: 25012
DstPort: 443
SrcIP: 10.0.1.15
DstIP: 162.249.4.107
## Coupon collector table

### Space efficiency:
- Keys from all queries multiplexed into one table
- Only keep rows for “active keys” (at least one coupon)
- Clear rows after timeout

### Query
- **q₆**
  - Key: 10.0.1.33
  - Coupon #1
  - Packet:
    - SrcPort: 27000
    - DstPort: 443
    - SrcIP: 10.0.1.33
    - DstIP: 162.249.4.107

- **q₆**
  - Key: 10.0.1.15
  - Coupon #3
  - Packet:
    - SrcPort: 25012
    - DstPort: 443
    - SrcIP: 10.0.1.15
    - DstIP: 162.249.4.107

**SrcIP 10.0.1.33 is sending to >1000 distinct DstIPs.**
Installing queries into switches

- The installed rules represent query set $Q$.
- Update queries on the fly, without recompiling.

![Diagram](image)
Evaluation highlights

• How efficient is BeauCoup?
  We use 4x~10x fewer memory access than the state-of-the-art to achieve the same accuracy.

• How much hardware resource?
  On the Barefoot Tofino programmable switch, BeauCoup occupies <50% of each resource.
BeauCoup: Answering many network traffic queries, one memory update at a time!

- **Scalable**: built upon *coupon collectors*, runs many queries simultaneously
- **Versatile**: change queries on the fly, without recompiling P4 program
- **Efficient**: achieve the same accuracy using 4x-10x fewer memory accesses
Is this a good use case of programmable dataplanes?
What are the limitations?
Your opinions

• Pros
  • The idea of using coupon collector problem
  • Ability to limit memory usage and memory accesses, while maintaining relatively high accuracy.
  • Thorough evaluation.
Your opinions

• Cons
  • Is distinct counts sufficient?
  • Can we do this at the endhosts?
  • Is fair allocation across queries the right strategy?
  • Fig 8 shows that accuracy for aggregated queries may not increase much with increased memory access.
  • Overhead of configuration?
Your opinions

• Ideas
  • Supporting a broader range of queries
  • Coordination across switches
  • Generate adversarial workloads
  • Can we achieve better accuracy?
Elmo: Source Routed Multicast for Public Clouds

Muhammad Shahbaz
Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira
I-to-Many Communication in Cloud
I-to-Many Communication in Cloud

Distributed Programming Frameworks
State Replication
Publish-Subscribe Systems
Infrastructure Applications
Streaming Telemetry

and more ...
I-to-Many Communication in Cloud

10,000s of tenants

→ 100s of workloads

→ Millions of groups

amazon  Google  Microsoft
1-to-Many Communication in Cloud

10,000s of tenants

→ 100s of workloads

→ Millions of groups

Multicast

Amazon  Google  Microsoft
I-to-Many Communication in Cloud

10,000s of tenants

100s of workloads

Millions of groups

Multicast

Amazon | Google | Microsoft
Limitations of **Native Multicast**

Controller
Limitations of **Native Multicast**

- **Processing overhead**
- **Excessive control churn** due to membership and topology changes
- **Limited group entries**
Restricted to **Unicast-based Alternatives**

![Diagram showing processing overhead]

- Controller
- Processing overhead
- R
- R
Restricted to **Unicast-based Alternatives**

Controller

- **Processing overhead**
- **Traffic overhead**
I-to-Many Communication in the Cloud

- Processing overhead
- Excessive control churn due to membership and topology changes
- Traffic overhead
- Limited group entries

Controller
I-to-Many Communication in the Cloud

Need a scheme that **scales** to millions of groups **without** excessive control, end-host CPU, and traffic **overheads**!
Proposal: Source Routed Multicast
Proposal: Source Routed Multicast

Controller

[S] [R] [R]
Proposal: Source Routed Multicast

- Little processing overhead
- Minimal control churn
- No traffic overhead
- No group entries needed*
- Negligible processing overhead
**A Naïve Source Routed Multicast**

A multicast group encoded as a list of *(Switch, Ports)* pairs

For a data center with:
- 1000 switches
- 48 ports per switch

Switch 1: [Ports ]
Switch 2: [ ... ... ]
Switch 3: [ ... ... ]
Switch 4: [ ... .x .. ]
Switch 5: [ .x ... ... ]

O(30) bytes per switch
O(30,000) bytes header for a group spanning 1000 switches

20x the packet size!
Enabling Source Routed Multicast in Public Clouds

Key attributes:

- **Efficiently encode** multicast forwarding policy inside packets

- **Process** this encoding at **hardware speed** in the switches

- **Execute** tenants’ applications **without modification**
Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of (Switch, Ports) pairs

| Switch 1: | [Ports ] |
| Switch 2: | [.. . . . . ] |
| Switch 3: | [.. . . . . ] |
| Switch 4: | [.. . . . . .x . . ] |
| Switch 5: | [x . . . . . ] |
Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of (Switch, Ports) pairs

Switch 1: [Bitmap]
Switch 2: [. . . . .]
Switch 3: [. . . . .]
Switch 4: [. . . . x . .]
Switch 5: [. x . . . . .]

Bitmap is the internal data structure that switches use for replicating packets

1. Encode switch ports as a bitmap
Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of (Switch, Ports) pairs

Switch 1: [Bitmap]
Switch 2: [.. .. ..]
Switch 3: [.. .. ..]
Switch 4: [.. .. .x ..]
Switch 5: [.x .. .. ..]

2 Group switches into layers
Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of \((\text{Switch}, \text{Ports})\) pairs

<table>
<thead>
<tr>
<th>Switch 1</th>
<th>[Bitmap]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 2</td>
<td>[. . . . .]</td>
</tr>
<tr>
<td>Switch 3</td>
<td>[. . . . .]</td>
</tr>
<tr>
<td>Switch 4</td>
<td>[. . . . .x .]</td>
</tr>
<tr>
<td>Switch 5</td>
<td>[.x . . . . .]</td>
</tr>
</tbody>
</table>

More precisely: upstream leaf, upstream spine, core, downstream spine, downstream leaf
Encoding a Multicast Policy in Elmo
## Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of \((\text{Switch}, \text{Ports})\) pairs

<table>
<thead>
<tr>
<th>Switch 1: [Bitmap]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 2: [.. .. ..]</td>
</tr>
<tr>
<td>Switch 3: [.. .. ..]</td>
</tr>
<tr>
<td>Switch 4: [.. .. .x ..]</td>
</tr>
<tr>
<td>Switch 5: [.x .. .. ..]</td>
</tr>
</tbody>
</table>

3. Switches within a layer with the same ports share a bitmap.
Encoding a Multicast Policy in Elmo

Sender $H_a$

|Outer header(s)| VXLAN | Sender-specific leaf, spine, and core $p$-rules|  |
|---|---|---|---|---|---|---|
|  |  | type | $u$-leaf | $u$-spine | d-core |  |
|  |  | 01|M | 00|M | 0011 |  |

Sender $H_k$

|Outer header(s)| VXLAN | Sender-specific leaf, spine, and core $p$-rules|  |
|---|---|---|---|---|---|---|
|  |  | type | $u$-leaf | $u$-spine | d-core |  |
|  |  | 00|M | 00|M | 1001 |  |

Common downstream spine and leaf $p$-rules

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$-spine</td>
<td>$d$-leaf</td>
</tr>
<tr>
<td>10:[P_0]</td>
<td>11:[L_7] Default</td>
</tr>
<tr>
<td>01:[P_2]</td>
<td>11:[L_6,L_8]</td>
</tr>
<tr>
<td>01:[P_2]</td>
<td>01:[L_7] Default</td>
</tr>
</tbody>
</table>

Packet body

P0: forward to L0
P2: forward to L5
P3: forward to L6, L7
L0: forward to H_a, H_b
L5: forward to H_e
L6: forward to H_m, H_n
L7: forward to H_p
Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of (Switch, Ports) pairs

Switch 1: [Bitmap]
Switch 2,3: [.......
Switch 4: [....x..]
Switch 5: [.x.......

3 Switches within a layer with same ports share a bitmap
Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of (Switch, Ports) pairs

<table>
<thead>
<tr>
<th>Switch 1: [Bitmap]</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 2,3: [.. .. ..]</td>
<td>Leaf</td>
</tr>
<tr>
<td>Switch 4: [.. .. ..]</td>
<td>Leaf</td>
</tr>
<tr>
<td>Switch 5: [.. x .. .. ..]</td>
<td>Leaf</td>
</tr>
</tbody>
</table>

Modern commodity switches can parse packet headers of 512 bytes

For a data center with:
- 628 switches
- 325 bytes header space

Supports 890,000 groups!

Switches within a layer with same ports share a bitmap
## Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of (Switch, Ports) pairs

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 1</td>
<td>[Bitmap]</td>
</tr>
<tr>
<td>Switch 2,3</td>
<td>[. . . . . .]</td>
</tr>
<tr>
<td>Switch 4</td>
<td>[. . . . .x . .]</td>
</tr>
<tr>
<td>Switch 5</td>
<td>[.x . . . . . .]</td>
</tr>
</tbody>
</table>

### Diagram

- **Core**
- **Spine**
- **Leaf**

### Note

3. Switches within a layer with same ports share a bitmap
Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of (Switch, Ports) pairs

Switch 1: [Bitmap]
Switch 2,3: [.. .. ..]
Switch 4: [.. .. .x ..]
Switch 5: [.x .. .. ..]

Core
Spine
Leaf

4 Switches within a layer with N different ports share a bitmap
Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of (Switch, Ports) pairs

Switch 1: [Bitmap]
Switch 2,3: [... ...]
Switch 4,5: [.x ... x ..]

4 Switches within a layer with N different ports share a bitmap
Encoding a Multicast Policy in **Elmo**

Sender **Hₐ**

<table>
<thead>
<tr>
<th>Outer header(s)</th>
<th>VXLAN</th>
<th>type</th>
<th>u-leaf</th>
<th>u-spine</th>
<th>d-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender <strong>H₉</strong></td>
<td></td>
<td>u</td>
<td>01</td>
<td>M</td>
<td>00</td>
</tr>
</tbody>
</table>

Sender-specific leaf, spine, and core p-rules

At L₀: forward to H₀ and multipath to P₀
P₀: multipath to C
C: forward to P₂, P₃

Common downstream spine and leaf p-rules

<table>
<thead>
<tr>
<th>d-spine</th>
<th>d-leaf</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:[P₀]</td>
<td>11:[P₃]</td>
</tr>
<tr>
<td>01:[P₂]</td>
<td>[L₃] Default</td>
</tr>
</tbody>
</table>

Packet body

<table>
<thead>
<tr>
<th>Sender <strong>H₉</strong></th>
<th>Outer header(s)</th>
<th>VXLAN</th>
<th>type</th>
<th>u-leaf</th>
<th>u-spine</th>
<th>d-core</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>u</td>
<td>00</td>
<td>M</td>
<td>00</td>
<td>M</td>
</tr>
</tbody>
</table>

At L₅: multipath to P₂
P₂: multipath to C
C: forward to P₇

Packet body

P₀: forward to L₀
P₂: forward to L₅
P₃: forward to L₆, L₇
L₀: forward to Hₙ, Hₙ
L₃: forward to Hₙ
L₇: forward to Hₙ
A multicast group encoded as a list of (Switch, Ports) pairs

Switch 1: [Bitmap]
Switch 2,3: [.. .. ..]
Switch 4,5: [.x .. .x ..]

4 Switches within a layer with N different ports share a bitmap

For a data center with:
- 628 switches
- 325 bytes header space

Supports 980,000 groups!
# Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of \((\text{Switch}, \text{Ports})\) pairs

<table>
<thead>
<tr>
<th>Switch 1: [Bitmap]</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 2,3: [.. .. ..]</td>
<td>Spine</td>
</tr>
<tr>
<td>Switch 4,5: [.. .x .. x ..]</td>
<td>Leaf</td>
</tr>
</tbody>
</table>

4. Switches within a layer with \(N\) different ports share a bitmap
Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of \((\text{Switch}, \text{Ports})\) pairs

- **Switch 1:** [Bitmap]
- **Switch 2,3:** [.. .. ..]
- **Switch 4,5:** [.x .. .x ..]

5. Use **switch entries** and a **default bitmap** for larger groups

Default Bitmap
Switch Table Entries
Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of (Switch, Ports) pairs

Switch 1: [Bitmap]
Switch 2,3: [.. .. ..]
Switch 4,5: [.x .. .x ..]

Use switch entries and a default bitmap for larger groups

For a data center with:
- 628 switches
- 325 bytes header space

Default Bitmap
Switch Table Entries

Traffic overhead

Switch entries

Difference in ports

Fixed Header Size

Core
Spine
Leaf
Encoding a Multicast Policy in Elmo

A multicast group encoded as a list of (Switch, Ports) pairs:

Switch 1: [Bitmap]
Switch 2,3: [.. .. ..]
Switch 4,5: [.x .. .x ..]

Core
Spine
Leaf

Default Bitmap
Switch Table Entries

1. Encode switch ports as a bitmap
2. Group switches into layers
3. Switches within a layer with:
   - same ports share a bitmap
4. N different ports share a bitmap
5. Use switch entries and a default bitmap for larger groups

For a data center with:
- 628 switches
- 325 bytes header space

Supports a Million groups!
Processing a Multicast Policy in Elmo

1. API

2. Computes the multicast policy

3. Installs entries in programmable
   - virtual switches to push Elmo headers on packets
   - hardware switches

- More flow entries and higher update rates than hardware switches
- No changes to the tenant application
Processing a Multicast Policy in Elmo

Switch looks for:
- Matching bitmap
- Table entry
- Default bitmap

Implemented using P4 on a Barefoot Tofino Switch
Applications Run Without Performance Overhead

![Graph showing Subscriber Throughput and CPU Utilization for Elmo and Unicast](image)

- **Subscriber Throughput (rps)**
  - **Elmo**
  - **Unicast**

- **Publisher CPU Utilization (%)**
  - **Elmo**
  - **Unicast**

**Number of subscribers**

- 1
- 2
- 4
- 8
- 16
- 32
- 64
- 128
- 256

**Throughput (rps)**

- 0K
- 50K
- 100K
- 150K
- 200K
Conclusion

Elmo
Source Routed Multicast for Public Clouds

• Designed for multi-tenant data centers
• Compactly encodes multicast policy inside packets
• Operates at hardware speed using programmable data planes

Learn more here:
https://elmo-mcast.github.io
Is this a good usecase of programmable dataplanes?
What are the limitations?
Your opinions

• Pros
  • Scalable multicast – support hundreds of thousands of tenants
    • Devises different mechanisms for this.
  • Exploits datacenter topology structure
  • Compact headers
  • Low overhead
  • Good application of P4
  • Adhere to constraints of programmable switches.
Your opinions

• Cons
  • Too specific to fat-tree topology
  • What happens under link/port failures?
  • Is the controller churn actually smaller than conventional approach?
Your opinions

- Ideas
  - More general topology
  - Inter-datacenter multicast
Other networking use cases

• Load balancing:
  • HULA: Scalable Load Balancing Using Programmable Data Planes, SOSR’16

• Congestion control:
  • Evaluating the Power of Flexible Packet Processing for Network Resource Allocation, NSDI’17
    • Support RCP and XCP on programmable switches
  • HPCC: High Precision Congestion Control, SIGCOMM’19
    • Obtain precise link information for congestion control

• A new protocols for more efficient L2 switching
  • The Deforestation of L2, SIGCOMM’16

• ....