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: \(^1\)

Answering many network traffic queries, one memory update at a time!

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

\(^1\)[bo'ku] Adv. many, a lot.
Network traffic query

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

Select $\text{DstIP}$ where $\text{distinct}(\text{SrcIP}) > 1000$
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>SrcIP</td>
<td>1000</td>
</tr>
<tr>
<td>Worm</td>
<td>SrcIP</td>
<td>DstIP</td>
<td>300</td>
</tr>
<tr>
<td>PortScan</td>
<td>SrcIP, DstIP</td>
<td>DstPort</td>
<td>100</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
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 on average
  2. Combine many different queries, on average uses $O(1)$
  3. Coordinate, at most $O(1)$ per packet
BeauCoup’s Approach

• Challenge:
  many queries, few memory updates

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

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

Query: Select $\text{DstIP}$ where distinct($\text{SrcIP}$) > 130

• Map each $\text{SrcIP}$ 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

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

Mapping

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

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

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

- 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 SrclP, each coupon is drawn with probability 1%
Stacking queries: same attribute

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

$q_2: f(SrcIP) \rightarrow \text{Coupon}$
$m_2 = 3, p_2 = 1/16$

Hash function $h_1(SrcIP) \rightarrow [0, 1)$

4 coupons for $q_1$
3 coupons for $q_2$
One hash function for each attribute

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

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

Randomly break ties if a coupon needs to be collected for two different attributes
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
\[ \gamma_q = \frac{\Gamma}{|Q|} \] (fair allocation)

Compiler

Goal:
I. Stop near Threshold
II. Update limit \( m \cdot p \leq \gamma_q \)
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
Installing queries into switches

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

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

Key, Attribute, Threshold

Header field tuples

Static program

Code Generator

P4 code

P4 Compiler

Dynamic rules

Query Compiler

$(m,p,n)$

Rules Generator

Rules

Data plane program

Table rules

Programmable Switch

Packets

Alerts
TCAM for selecting a coupon

Packet
SrcPort: 25012
DstPort: 443
SrcIP: 10.0.1.15
DstIP: 162.249.4.107

\( h_A(SrcPort) \) = 11010
\( h_B(DstPort) \) = 11010
\( h_C(SrcIP) \) = 1010111...
\( h_D(DstIP) \) = 0101011...

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>

No coupon
No coupon
No coupon
Collect coupon \( (q_6, \#3) \)

Random tiebreak if >1 coupons
**Coupon collector table in SRAM**

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

*SrcIP 10.0.1.33* is sending to >1000 distinct *DstIPs.*
Evaluation highlights

• How efficient is BeauCoup?
Uses 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?
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
- Publish-Subscribe Systems
- State Replication
- Streaming Telemetry
- Infrastructure Applications
- 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**

Controller

Processing overhead

R

R

R
Restricted to **Unicast-based Alternatives**

Controller

Traffic overhead

Processing overhead
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

Diagram showing the connection between S, R, and Controller.
Proposal: Source Routed Multicast

Controller

- 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

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

<table>
<thead>
<tr>
<th>Switch 1: [Ports ]</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>
Encoding a Multicast Policy in Elmo

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

1. Encode switch ports as a **bitmap**

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.
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 (Switch, Ports) pairs

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

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 (Switch, 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 same ports share a bitmap
Encoding a Multicast Policy in Elmo

### Sender H_s
- **Outer header(s)**: VXLAN
- **Sender-specific leaf, spine, and core p-rules**
  - **u-leaf**: 01|M
  - **u-spine**: 00|M
  - **d-core**: 0011

### Sender H_k
- **Outer header(s)**: VXLAN
- **Sender-specific leaf, spine, and core p-rules**
  - At L_s: multipath to P_2
  - P_2: multipath to C
  - C: forward to P_2, P_3
- **Common downstream spine and leaf p-rules**
  - **d-spine**: 10:[P_0], 11:[P_3] Default
  - **d-leaf**: 11:[L_0,L_3], 01:[L_7] Default

---

**Packet body**

---

**Packet body**
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>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 2,3: [.. ... ..]</td>
</tr>
<tr>
<td>Switch 4: [.. ... .x ..]</td>
</tr>
<tr>
<td>Switch 5: [..x .. ... ..]</td>
</tr>
</tbody>
</table>

3 Switches within a layer with same 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**: [... ... ...]
- **Switch 5**: [.. . . . . .]

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

Modern commodity switches can parse packet headers of 512 bytes.

3. Switches within a layer with **same ports share a bitmap**

![Bar chart showing number of groups covered using non-default rules](image)

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

  Supports **890,000 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>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 2,3: [.........]</td>
</tr>
<tr>
<td>Switch 4: [.....x..]</td>
</tr>
<tr>
<td>Switch 5: [.x.....]</td>
</tr>
</tbody>
</table>

### Notes
- Swiiches within a layer with same ports **share a bitmap**

### Layers
- **Core**
- **Spine**
- **Leaf**
Encoding a Multicast Policy in Elmo

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

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

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 $H_a$

| Outer header(s) | VXLAN | $u$ | 01|M | 00|M | 0011 |
|-----------------|-------|-----|-----|-----|-------|

Sender $H_k$

| Outer header(s) | VXLAN | $u$ | 00|M | 00|M | 1001 |
|-----------------|-------|-----|-----|-----|-------|

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

At $L_0$: forward to $H_0$ and multipath to $P_0$
At $L_5$: multipath to $C$

$P_0$: multipath to $P_2$, $P_3$
$P_2$: multipath to $C$
$P_3$: forward to $P_0$, $P_3$
$C$: forward to $P_2$, $P_3$

Common downstream spine and leaf $p$-rules

<table>
<thead>
<tr>
<th>$d$-spine</th>
<th>$d$-leaf</th>
</tr>
</thead>
<tbody>
<tr>
<td>01:[$P_2$]</td>
<td>Default</td>
</tr>
<tr>
<td>10:[$L_8$]</td>
<td></td>
</tr>
</tbody>
</table>

Packet body

Packet body
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

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!

Table:

<table>
<thead>
<tr>
<th>No. of groups</th>
<th>Difference in ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1M</td>
</tr>
</tbody>
</table>
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 ..]

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 ..]

Default Bitmap
Switch Table Entries

Use switch entries and a default bitmap for larger groups
Encoding a Multicast Policy in Elmo

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

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

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

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

Default Bitmap
Switch Table Entries

5.1.2 Data-Plane Scalability.

Switch Table entries

Traffic overhead

Difference in ports
Encoding a Multicast Policy in **Elmo**

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

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

**Core**

**Spine**

**Leaf**

1. Encode switch ports as a **bitmap**
2. Group switches into **layers**
3. Switches within a layer with:
   - **same** ports **share a bitmap**
   - **N different** ports **share a bitmap**
4. 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

![Graphs showing throughput and CPU utilization for Elmo and Unicast](image)

- **Substrate Throughput (rps):**
  - **Elmo** vs **Unicast**
  - Number of subscribers range from 1 to 256

- **Publisher CPU Utilization (%):**
  - **Elmo** vs **Unicast**
  - Number of subscribers range from 1 to 256
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
Is this a good use case of programmable dataplanes?
What are the limitations?
Other networking usecases

- **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

- And others…