Towards *Fast* and *Flexible* Datacenters

Muhammad Shahbaz
Stanford University
• The continuous growth in chips’ processing capacity has...
  • kept CPUs general-purpose and fast
  • and switches fixed-function and dumb

• However, recent trends are forcing us to rethink how we build CPUs and switches.
  • The **decline of Moore’s Law**
  • The **rise of Big Data** (e.g., AI, Social Media, IoT)
Concern ...

Must now **repurpose on-chip resources**, for general-purpose processing, to a given **application domain** (e.g., Machine Learning).

Must now **take on more responsibility**, and can no longer be fixed-function, **line-rate** devices.
Research Theme

CPUs

Switches

Networks

Flexibility -
Performance +

Flexibility +
Performance -

Systems

Domain-Specific Languages (DSLs) & Architectures (DSAs)

ARGUS

L-NIC

eRSS

λ-NIC

PISCES

NetASM

Elmo

Taurus

...
What's the cost of flexibility on performance?
And, demonstrate using PISCES and Elmo that flexibility comes at negligible cost.
PISCES: A Programmable, Protocol-Independent Software Switch

Muhammad Shahbaz\(^1\)

Sean Choi\(^2\), Ben Pfaff\(^3\), Changhoon Kim\(^4\), Nick Feamster\(^1\), Nick McKeown\(^2\), and Jen Rexford\(^1\)

Programmable switches
Programmable switches

Language

Still fixed-function switches

vSwitch
Software Switches

- vSwitch
- Fast Packet Forwarding
- OVS
- vSwitch
Software Switches

- vSwitch
- DPDK
- Kernel
- NetDev
- OVS
Software Switches

Packet Processing Logic

- DPDK
- Kernel
- NetDev
- ...

vSwitch

OVS
Software Switches
Software Switches

Requires domain expertise in:

- Network protocol design
- Software development
  - Develop
  - Test
  - Deploy

...large, complex codebases.

- Can take 3-6 months to get a new feature in.

- Maintaining changes across releases
Software Switches

- vSwitch
- DPDK
- Kernel
- NetDev
- Match-Action Pipeline
- Complex APIs
- OVS

Parser
Software Switches
Software Switches
PISCES: A Programmable Software Switch
PISCES: A Programmable Software Switch
P4 is an open-source language.\footnote{http://www.p4.org}

Describes different aspects of a packet processor:
- Packet headers and fields
- Metadata
- Parser
- Actions
- Match-Action Tables (MATs)
- Control Flow

\footnote{http://www.p4.org}
PISCES: A Programmable Software Switch
PISCES: A Programmable Software Switch

Parser | Match-Action Pipeline

PISCES

Compiler

OVS

DPDK | Kernel | NetDev | ...

P4

Executable
Research Goals

1. **Quantify reduction in complexity**, *i.e.*, does expressing customizations in P4 easier than direct modifications to the OVS source code?

2. **Performance optimizations**, *i.e.*, is there any overhead in compiling P4 programs to OVS? If so, can we mitigate these overheads via compiler optimizations.
Quantifying Reduction in Complexity

We evaluate two categories of complexity:

1. Development complexity of developing baseline packet processing logic for a software switch

2. Change complexity of making changes and maintaining an existing software switch
Quantifying Reduction in Complexity

1. Development complexity

- **Lines of Code**
  - OVS: 14,535
  - PISCES: 341
  - Reduction: 40x

- **Method Count**
  - OVS: 106
  - PISCES: 40
  - Reduction: 40x

- **Method Size (Avg.)**
  - OVS: 137.1
  - PISCES: 8.5
  - Reduction: 20x
Quantifying Reduction in Complexity

2. Change complexity

<table>
<thead>
<tr>
<th>Connection Label</th>
<th>Tunnel OAM Flag</th>
<th>TCP Flag</th>
<th>Files Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVS</td>
<td>28</td>
<td>1</td>
<td>411</td>
</tr>
<tr>
<td>PISCES</td>
<td>18</td>
<td>1</td>
<td>170</td>
</tr>
<tr>
<td>OVS</td>
<td>20</td>
<td>1</td>
<td>370</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Connection Label</th>
<th>Tunnel OAM Flag</th>
<th>TCP Flag</th>
<th>Lines Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVS</td>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>PISCES</td>
<td>170</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>
Performance Optimizations
Performance Optimizations

Parser | Match-Action Pipeline

PISCES

P4

Compiler

OVS

Executable

Performance overhead?
Naïve Compilation from P4 to OVS (L2L3-ACL)

Performance overhead of 40%
P4 Forwarding Model

Ingress → Packet Parser → Checksum Verify → Match-Action Tables → Checksum Update → Packet Deparser → Egress

Header Fields
OVS Forwarding Model

Ingress → Packet Parser → Slow-Path → Fast-Path
OVS Forwarding Model

- **Ingress**
  - Packet Parser
  - Match-Action Cache

- **Match-Action Tables**

- **Egress**

- **Slow-Path**
- **Fast-Path**
OVS Forwarding Model

Ingress Packet Parser → Match-Action Cache → Egress

Match-Action Tables

Egress

Slow-Path

Fast-Path

Hit
OVS Forwarding Model (Inline Editing)
PISCES Forwarding Model (Modified OVS)

- Supports both editing modes:
  - **Inline Editing**
  - **Post-pipeline Editing**
Compiling P4 to OVS

Naïve compilation performance overhead is 40%
Causes of Performance Overhead

- Match-Action Tables
- Packet Parser
- Checksum Verify
- Match-Action Cache
- Checksum Update
- Packet Deparser

- CPU Cycles per Packet
- Cache Misses
Factors Affecting CPU Cycles

L2L3-ACL (CPU Cycles for a 64 Byte Packet)

Throughput (Gbps)

<table>
<thead>
<tr>
<th></th>
<th>Naïve</th>
<th>OVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parser</td>
<td>76.5</td>
<td>43.6</td>
</tr>
<tr>
<td>Cache: Match</td>
<td>209.5</td>
<td>197.5</td>
</tr>
<tr>
<td>Cache: Actions</td>
<td>379.5</td>
<td>132.5</td>
</tr>
<tr>
<td>Throughput (Gbps)</td>
<td>7.6</td>
<td>13.5</td>
</tr>
</tbody>
</table>
Factors Affecting CPU Cycles

a. Extra copy of headers

b. Fully-specified checksums

c. Parsing unused header fields

and more ...
## Performance Optimizations

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Per-Packet Cost (CPU Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inline vs. post-pipeline editing</td>
<td>✓</td>
</tr>
<tr>
<td>Incremental checksum</td>
<td>✓</td>
</tr>
<tr>
<td>Parser specialization</td>
<td>✓</td>
</tr>
<tr>
<td>Action specialization</td>
<td>✓</td>
</tr>
<tr>
<td>Action coalescing</td>
<td>✓</td>
</tr>
</tbody>
</table>
# Performance Optimizations

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Per-Packet Cost (CPU Cycles)</th>
<th>Cache Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inline vs. post-pipeline editing</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Incremental checksum</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Parser specialization</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Action specialization</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Action coalescing</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Stage assignment</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Cached field modifications</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Performance Optimizations for L2L3-ACL

L2L3-ACL (CPU Cycles for a 64 Byte Packet)

- Parser: Naïve
- Cache: Match: 
  - Naïve
  - Inline
  - Inc. Chksm
  - Parsr Spcl
  - Act Spcl
  - Act Coalcng
  - OVS
- Cache: Actions: 
  - Naïve
  - Inline
  - Inc. Chksm
  - Parsr Spcl
  - Act Spcl
  - Act Coalcng
  - OVS

Performance overhead of < 2%
PISCES: A Programmable Software Switch

• Programs written in PISCES, using P4, are **40 times more concise** than native software code.

• With hardly **any performance overhead**!
Elmo: Source-Routed Multicast for Cloud Services

Muhammad Shahbaz

Lalith Suresh, Jen Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira

1. Princeton   2. VMware   3. Technion
1-to-Many Communication in Cloud
1-to-Many Communication in Cloud

- Distributed Programming Frameworks
- Publish-Subscribe Systems
- State Replication
- Streaming Telemetry
- Infrastructure Applications
- and more …
1-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
1-to-Many Communication in Cloud

- 10,000s of tenants
- 100s of workloads
- Millions of groups

Multicast

Amazon, Google, Microsoft
Limitations of **Native** Multicast
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
Restricted to **Unicast-based Alternatives**

---

Controller

- Traffic overhead

- Processing overhead

---

54
1-to-Many Communication in the Cloud

Controller
1-to-Many Communication in the Cloud

Controller

Traffic overhead

Processing overhead
1-to-Many Communication in the Cloud

- Processing overhead
- Excessive control churn due to membership and topology changes
- Traffic overhead
- Limited group entries
- Processing overhead
1-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
Proposal: Source Routed Multicast

- Little processing overhead
- Minimal control churn
- No traffic overhead
- No group entries needed
- Negligible processing overhead
Enabling Source Routed Multicast in Public Clouds

Key challenges:

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

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

- **Execute** tenants’ applications **without modification**
A Naïve Source Routed Multicast

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

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

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!
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 .. .. ..]

Encode switch ports as a bitmap

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 .. .. .. ]

Group switches into layers
Encoding a Multicast Policy in **Elmo**

A multicast group encoded as a list of \textbf{(Switch, 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>

2 Group switches into \textbf{layers}

\textbf{Core} \hspace{1cm} \textbf{Spine} \hspace{1cm} \textbf{Leaf}

Core \hspace{1cm} Spine \hspace{1cm} 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:</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>

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 .. ...]

Switches within a layer with same ports share a bitmap
A multicast group encoded as a list of (Switch, Ports) pairs

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

Modern commodity switches can parse packet headers of **512 bytes**

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

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 (Switch, Ports) pairs

<table>
<thead>
<tr>
<th>Switch 1:</th>
<th>[Bitmap]</th>
</tr>
</thead>
<tbody>
<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>

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 .. .. ..]

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

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

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 \( \text{(Switch, Ports)} \) pairs

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

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 \((\text{Switch, Ports})\) pairs

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

---

**Fixed Header Size**

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

Fixed Header Size

Core

Spine

Leaf

5. 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

Difference in ports

Traffic Overhead

Switch entries

5.0K

2.5K

0.0

0 6 12

0 6 12

5.1.2 Data-Plane Scalability.

Groups covered with p

250K

500K

20K

3

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

Difference in ports
Encoding a Multicast Policy in Elmo

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

Switch 1: \([\text{Bitmap}]\)
Switch 2,3: \([.. .. ..]\)
Switch 4,5: \([.x .. .x ..]\)

Core
Spine
Leaf

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

Controller
We measure that today's hypervisor switches are capable of computing Algorithm 1 for routing traffic. When a leaf switch fails, all hosts connected to it lose connectivity to the network, and must wait for the switch to get back online. The PISCES [97] switch with support for the Intel(R) Xeon(R) CPUs running at 2.00 GHz and with 32 GB of memory loads. We reduce CPU and bandwidth utilization for multicast workloads to the network, and for the network state to converge [89]. In our simulations, up to 12.25% of groups are impacted when a single spine switch fails and up to 25.81% of groups are impacted when a core switch fails. Hypervisor switches incur average 1.45 ms (SD), on average, for all group sizes with a header size limit of 325 bytes. Existing studies report that it takes less than a millisecond. Elmo gracefully handles spine and core failures.

Our controller consistently executes $p$- and $s$-rules in less than a millisecond. Across our simulations, our Python implementation of Elmo, the throughput remains the same regardless of the number of subscribers.5 A drawback of native multicast is that we cannot use TCP. However, protocols (like PGM [100] and SRM [53]) can support applications that require reliable delivery using native multicast.

5.3 Evaluating End-to-end Applications

5.3.1 Publish-subscribe using ZeroMQ.

Figure 9: Requests-per-second and CPU utilization

<table>
<thead>
<tr>
<th>Number of subscribers</th>
<th>Elmo Unicast</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Subscriber Throughput (rps)</td>
</tr>
<tr>
<td></td>
<td>Publisher CPU Utilization (%)</td>
</tr>
</tbody>
</table>

As our second application, we evaluate applications running without performance overhead. Applications Run Without Performance Overhead.
Control Plane Scalability

For a multi-rooted Clos topology with 27K hosts and p-rule header of 325 bytes:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Elmo</th>
<th>Li et al.*</th>
</tr>
</thead>
<tbody>
<tr>
<td>hypervisor</td>
<td>21 (46)</td>
<td>NE (NE)</td>
</tr>
<tr>
<td>leaf</td>
<td>5 (13)</td>
<td>42 (42)</td>
</tr>
<tr>
<td>spine</td>
<td>4 (7)</td>
<td>78 (81)</td>
</tr>
<tr>
<td>core</td>
<td>0 (0)</td>
<td>133 (203)</td>
</tr>
</tbody>
</table>

* https://conferences.sigcomm.org/co-next/2013/program/p61.pdf
Elmo Operates within the Header Size Limit of Switch ASICs

For a 256-port, 200 mm$^2$ baseline switching ASIC that can parse a 512-byte packet header:

- Header size limit for RMT (512B)

190 bytes for other protocols (e.g., datacenter protocols take about 90 bytes)
Elmo’s Primitives are Inexpensive to Implement in Switch ASICs

For a 256-port, 200 mm² baseline switching ASIC that can parse a 512-byte packet header:

As a comparison, Conga consumes 2% of area and Banzai consumes 12% of area.
Elmo: Source-Routed Multicast for Cloud Services

Elmo
A Scalable Multicast Service

- Designed for multi-tenant data centers
- Compactly encodes multicast policy inside packets
- Operates at line rate using programmable data planes
Future Directions

Data-driven computing and networking systems
- Software-Defined Hardware
- Self-Driving Networks

Ware-house scale computers

Domain-Specific Languages (DSLs) & Architectures (DSAs)
a. Data-driven computing and networking systems
   - Software-Defined Hardware
   - Self-Driving Networks

b. Warehouse scale computers