Stateful Multi-Pipelined Programmable Switches

Vishal Shrivastav
Purdue University
Motivating Example

Consider a packet processing program:

- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
  - Switch drops all subsequent packets destined to d
Motivating Example

Consider a packet processing program:

- Switch maintains packet counters for each destination IP
- If the counter value for destination $d$ exceeds a threshold
  - Switch drops all subsequent packets destined to $d$

Code

P4
Motivating Example

Consider a packet processing program:

- Switch maintains packet counters for each destination IP.
- If the counter value for destination d exceeds a threshold,
  - Switch drops all subsequent packets destined to d.

```
0
If C > threshold
Mark packet “to drop”
0
```

Diagram:

```
Port 0
    hash(dst)
    Port 1
```

Switch Pipeline:

```
0: 0
1: 0
2: 0
3: 0
```

If $C >$ threshold
Mark packet “to drop”
Motivating Example

Consider a packet processing program:

- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
  - Switch drops all subsequent packets destined to d

```
0: hash(dst)
1: 0
2: 0
3: 0
```

If C > threshold
Mark packet “to drop”
Motivating Example

Consider a packet processing program:

- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
  - Switch drops all subsequent packets destined to d

```
0: 1
D: X

hash(D: X) = 1
```

Switch Pipeline:

- Port 0
  - D: X

- Port 1

- Code

- Compile

- If C > threshold
  - Mark packet “to drop”
Motivating Example

Consider a packet processing program:

- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
  - Switch drops all subsequent packets destined to d

\[
\text{hash(D: X)} = 1
\]

Switch Pipeline

- If C > threshold
- Mark packet “to drop”
Motivating Example

Consider a packet processing program:

- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
  - Switch drops all subsequent packets destined to d

```
if C > threshold
    Mark packet "to drop"
```

---

**Switch Pipeline**

1. Port 0: D: X
2. hash(D: X) = 1
3. Port 1: D: X
4. Code
5. Compile
6. If C > threshold
   - Mark packet "to drop"
Reality of Today’s Switch Hardware

- Clock speed of a single pipeline has saturated
  - Limits the line rate

- Employ multiple **parallel pipelines** to sustain multi-tbps line rate
  - Each pipeline processes packets *independently* — No co-ordination
Goal

Rate: R

Logical single large pipeline

Code

Map

Rate: R/4

Rate: R
Goal

Logical single large pipeline → Code

Rate: R

Functional Equivalence
Runtime behavior of program same as on a single large pipeline

Performance Equivalence
Program runs as close to rate of a single large pipeline, i.e., R w/o violating functional equivalence

Rate: R/4
Our Contribution

We present a new switch design **MP5** that extends current programmable switch’s **architecture**, **compiler**, and **runtime** to guarantee **functional equivalence** with **high performance**.
Naive Approaches

Consider a stateless packet processing program:

- Switch increments the ttl value in packet header by 1
- If ttl value exceeds a threshold
  - Switch drops the packet
Consider a *stateless* packet processing program:

- Switch increments the ttl value in packet header by 1
- If ttl value exceeds a threshold
  - Switch drops the packet

```
if p.ttl > threshold
    Mark packet “to drop”
```
Consider a *stateless* packet processing program:

- Switch increments the ttl value in packet header by 1
- If ttl value exceeds a threshold
  - Switch drops the packet

**Try 1: Replicate stateless processing on all pipelines**
# Goals and Techniques

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Functional Equivalence</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateless</td>
<td>Replicate stateless processing</td>
<td>Stateless</td>
</tr>
<tr>
<td>Stateful</td>
<td></td>
<td>Stateful</td>
</tr>
</tbody>
</table>

- Replicate stateless processing

- ✔️ ✔️
Consider a *stateful* packet processing program:

- Switch maintains packet counters for each destination IP
- If the counter value for destination $d$ exceeds a threshold
  - Switch drops all subsequent packets destined to $d$

![Diagram](attachment:image.png)
Consider a *stateful* packet processing program:

- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
  - Switch drops all subsequent packets destined to d

**Try 1: Replicate stateful processing on all pipelines**
Naive Approaches

Consider a stateful packet processing program:
- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
  - Switch drops all subsequent packets destined to d

Try 1: Replicate stateful processing on all pipelines

Violates functional equivalence!
Naive Approaches

Consider a *stateful* packet processing program:
- Switch maintains packet counters for each destination IP
- If the counter value for destination $d$ exceeds a threshold
  - Switch drops all subsequent packets destined to $d$

**Try 2: Limit stateful processing to a single “shared” pipeline**

Port 0

hash(dst)

Port 1

If $C >$ threshold

Mark packet “to drop”
Naive Approaches

Consider a *stateful* packet processing program:

- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
  - Switch drops all subsequent packets destined to d

**Try 2: Limit stateful processing to a single “shared” pipeline**
Steer all packets to the “shared” pipeline

![Diagram of packet processing](image)
Consider a *stateful* packet processing program:

- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
  - Switch drops all subsequent packets destined to d

**Try 2: Limit stateful processing to a single “shared” pipeline**
Steer all packets to the “shared” pipeline

Limits speed of stateful processing!

---

**Naive Approaches**

- hash(D: X) = 1
- If C > threshold
  - Mark packet “to drop”
## Goals and Techniques

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Functional Equivalence</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stateless</td>
<td>Stateful</td>
</tr>
<tr>
<td>Replicate stateless processing + Limit stateful processing to single pipeline</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Question

How to improve performance? (without violating functional equivalence)
Problem

How to store shared state that enables high packet processing throughput?

If $C >$ threshold
Mark packet “to drop”
Solution

How to store shared state that enables high packet processing throughput?

Shard the shared state across pipelines
How to store shared state that enables high packet processing throughput?

**Solution**

Hash(D: X) = 0

Port 0

Hash(D: X) = 0

D: X

0: 1
1: 0

C: 1
D: X

If C > threshold
Mark packet “to drop”

Hash(D: Y) = 3

Port 1

Hash(D: Y) = 3

D: Y

2: 0
3: 1

C: 1
D: Y

If C > threshold
Mark packet “to drop”

*Shard* the shared state across pipelines
Solution

How to store shared state that enables high packet processing throughput?

**Shard** the shared state across pipelines

...but what is the optimal sharding strategy?
Solution

How to store shared state that enables high packet processing throughput?

**Shard** the shared state across pipelines

...but what is the optimal sharding strategy?

Port 0

\[ \text{hash}(D: X) = 0 \]

If \( C > \) threshold

Mark packet “to drop”

Port 1

\[ \text{hash}(D: Y) = 3 \]

If \( C > \) threshold

Mark packet “to drop”
Solution

How to store shared state that enables high packet processing throughput?

**Shard** the shared state across pipelines

...but what is the optimal sharding strategy?

If $C > \text{threshold}$

Mark packet “to drop”

Optimal

Port 0

$\text{hash}(D: X) = 0$

1:

2:

3:

Optimal

Port 1

$\text{hash}(D: Y) = 3$

If $C > \text{threshold}$

Mark packet “to drop”
Solution

How to store shared state that enables high packet processing throughput?

**Shard** the shared state across pipelines

...but what is the optimal sharding strategy?

Port 0

```
hash(D: Z) = 2
```

Port 1

```
hash(D: Y) = 3
```

If C > threshold

Mark packet “to drop”
How to store shared state that enables high packet processing throughput?

**Shard** the shared state across pipelines

...but what is the optimal sharding strategy?

<table>
<thead>
<tr>
<th>Port 0</th>
<th>Port 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash(D: Z) = 2</td>
<td>hash(D: Y) = 3</td>
</tr>
</tbody>
</table>

Not Optimal

If C > threshold
Mark packet “to drop”
Solution

How to store shared state that enables high packet processing throughput?

**Shard** the shared state across pipelines

...but what is the optimal sharding strategy?

Port 0  
hash(D: Z)  
= 2  

If C > threshold  
Mark packet “to drop”

Port 1  
hash(D: Y)  
= 3  

If C > threshold  
Mark packet “to drop”
Solution

How to store shared state that enables high packet processing throughput?

**Shard** the shared state across pipelines

…but what is the optimal sharding strategy?

Port 0

hash\(_{\text{D: Z}}\) = 2

Port 1

hash\(_{\text{D: Y}}\) = 3

If C > threshold
Mark packet “to drop”

Optimal

If C > threshold
Mark packet “to drop”
Solution

How to store shared state that enables high packet processing throughput?

**Shard** the shared state across pipelines

...but what is the optimal sharding strategy?

*Ensure state accesses are uniformly distributed across pipelines*
Solution

How to store shared state that enables high packet processing throughput?

**Shard** the shared state across pipelines

...but what is the optimal sharding strategy?

*Ensure state accesses are uniformly distributed across pipelines*

...depends upon the packet arrival pattern (hard to predict)
Solution

How to store shared state that enables high packet processing throughput?

**Shard** the shared state across pipelines

...but what is the optimal sharding strategy?

*Ensure state accesses are uniformly distributed across pipelines*

...depends upon the packet arrival pattern (hard to predict)

**Dynamically shard** the shared state across pipelines by **monitoring** the state access patterns at runtime
Solution

How to store shared state that enables high packet processing throughput?

Dynamically shard the shared state across pipelines by monitoring the state access patterns at runtime

Reduces to a variant of bin packing problem (NP-Hard!)
Solution

How to store shared state that enables high packet processing throughput?

**Dynamically shard** the shared state across pipelines by **monitoring** the state access patterns at runtime.

Reduces to a variant of **bin packing** problem (NP-Hard!)

MP5 uses a heuristic to approximates bin packing that is amenable to fast hardware implementation.
One Missing Detail

If $C > \text{threshold}$
Mark packet “to drop”

Port 0

hash(D: Z) = 2

Port 1

hash(D: Y) = 3

If $C > \text{threshold}$
Mark packet “to drop”
If $C > \text{threshold}$
Mark packet
"to drop"

Packet and the corresponding shared state may be on different pipelines!

Port 0
hash(D: Z) = 2

Port 1
hash(D: Y) = 3

If $C > \text{threshold}$
Mark packet “to drop”
Packet may need to go back and forth between pipelines to access the shared states!
One Missing Detail

How to steer packets to a shared state in a remote pipeline?
Existing Solution

How to steer packets to a shared state in a remote pipeline?

Packet Re-circulation
Existing Solution

How to steer packets to a shared state in a remote pipeline?

Packet Re-circulation
How to steer packets to a shared state in a remote pipeline?

**Packet Re-circulation**
How to steer packets to a shared state in a remote pipeline?

Packet Re-circulation
Existing Solution

How to steer packets to a shared state in a remote pipeline?

Packet Re-circulation results in **throughput penalty** and **increased latency** …because packets re-visit same stages multiple times!
How to steer packets to a shared state in a remote pipeline?

Packet Re-circulation results in **throughput penalty** and **increased latency**

...because packets re-visit same stages multiple times!

Need a **feed-forward-only** packet steering design
How to steer packets to a shared state in a remote pipeline?

Current switch design

A packet in stage $i$ of pipeline $j$ could move to stage $i+1$ of only pipeline $j$. 

- Red arrow from stage 0 to stage 1.
- Green arrow from stage 3 to stage 0.
Our Solution

How to steer packets to a shared state in a remote pipeline?

**Feed-forward-only packet steering design**

A packet in stage $i$ of pipeline $j$ could move to stage $i+1$ of only pipeline $j$ any pipeline.
Our Solution

How to steer packets to a shared state in a remote pipeline?

**Feed-forward-only packet steering design**

A packet in stage $i$ of pipeline $j$ could move to stage $i+1$ of only pipeline $j$ *any* pipeline
Question Re-visited

How to improve performance?
(without violating functional equivalence)
# Goals and Techniques

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Functional Equivalence</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replicate stateless processing</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Limit stateful processing to single pipeline</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Dynamic state sharding</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Feed-forward pkt steering</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

- Replicate stateless processing
- Limit stateful processing to single pipeline
- Dynamic state sharding
- Feed-forward pkt steering

- ✔: Satisfied
- X: Not satisfied
## Goals and Techniques

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Functional Equivalence</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stateless</td>
<td>Stateful</td>
</tr>
<tr>
<td>Replicate stateless processing</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Limit stateful processing to single pipeline</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Dynamic state sharding &amp;</td>
<td>✔️</td>
<td>?</td>
</tr>
<tr>
<td>Feed-forward pkt steering</td>
<td>✔️</td>
<td></td>
</tr>
</tbody>
</table>
### Goals and Techniques

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Functional Equivalence</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stateless</td>
<td>Stateful</td>
</tr>
<tr>
<td>Replicate stateless</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>processing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Limit stateful processing</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>to single pipeline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic state sharding</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feed-forward pkt steering</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Problem

Each pipeline can process 1 packet per time unit
Problem

Each pipeline can process 1 packet per time unit.

On a single-pipelined switch, D will always access register index 1 in stage 2 before E.
Problem

Each pipeline can process 1 packet per time unit
Problem

Each pipeline can process 1 packet per time unit.
Each pipeline can process 1 packet per time unit
Each pipeline can process 1 packet per time unit.

E will access index 1 in stage 2 before D! (may violate functional equivalence)
Problem

Each pipeline can process 1 packet per time unit

E will access index 1 in stage 2 before D!
(may violate functional equivalence)

Packet re-ordering can also impact application performance
e.g., if D and E belong to same TCP flow
Problem

How to avoid packet re-ordering and out-of-order state access?
Problem

How to avoid packet re-ordering and out-of-order state access?

Too late if we try to enforce ordering after a packet visits a stateful stage
...due to non-deterministic waits at a stateful stage
Solution

How to avoid packet re-ordering and out-of-order state access?

Too late if we try to enforce ordering after a packet visits a stateful stage
...due to non-deterministic waits at a stateful stage

Enforce ordering preemptively (i.e., before a packet reaches a stateful stage)
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 1: Preemptively figure out all states a packet would access
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 1: Preemptively figure out all states a packet would access

Hard in general (even impossible in some cases)
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 1: Preemptively figure out all states a packet would access

Hard in general (even impossible in some cases)

*Insight*: Most packet processing programs access register index based on hash of a subset of packet header fields
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 1: Preemptively figure out all states a packet would access

Hard in general (even impossible in some cases)

Insight: Most packet processing programs access register index based on hash of a subset of packet header fields

...can be known as soon as a packet arrives at the switch
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 1: Preemptively figure out all states a packet would access

Compiler adds a new stage before any stateful stage

Port 0

state index = hash(p.hdr)

Port 1

state index = hash(p.hdr)
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 2: Enforce ordering in the stateful stages

Compiler adds a new stage before any stateful stage

Port 0

state index = hash(p.hdr)

Port 1

state index = hash(p.hdr)
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 2: Enforce ordering in the stateful stages

Compiler adds a new stage before any stateful stage

Timestamp Packets?

Port 0

state index = hash(p.hdr) + Timestamp pkts

Port 1

state index = hash(p.hdr) + Timestamp pkts
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 2: Enforce ordering in the stateful stages

Compiler adds a new stage before any stateful stage

Stateful operation

Timestamp Packets? - won't work!

Port 0

Port 1
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 2: Enforce ordering in the stateful stages

Compiler adds a new stage before any stateful stage

Generate “placeholders” for data packets

Port 0

state index = hash(p.hdr) + Gen Phantom pkt

Port 1

state index = hash(p.hdr) + Gen Phantom pkt
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 2: Enforce ordering in the stateful stages

Compiler adds a new stage before any stateful stage

Generate “placeholders” for data packets

Port 0

state index = hash(p.hdr)

Gen Phantom pkt

Port 1

state index = hash(p.hdr)

Gen Phantom pkt
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 2: Enforce ordering in the stateful stages

Compiler adds a new stage before any stateful stage

Generate “placeholders” for data packets

Port 0

state index = hash(p.hdr)

Gen Phantom pkt

Port 1

state index = hash(p.hdr)

+ Gen Phantom pkt
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 2: Enforce ordering in the stateful stages

Compiler adds a new stage before any stateful stage

Generate “placeholders” for data packets

Port 0

state index = hash(p.hdr) + Gen Phantom pkt

Port 1

state index = hash(p.hdr) Gen Phantom pkt

Generate "placeholders" for data packets
Solution

How to avoid packet re-ordering and out-of-order state access?

Step 2: Enforce ordering in the stateful stages

Compiler adds a new stage before any stateful stage

Generate "placeholders" for data packets

state index = hash(p.hdr) + Gen Phantom pkt

state index = hash(p.hdr)

Order enforced
<table>
<thead>
<tr>
<th>Techniques</th>
<th>Functional Equivalence</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stateless</td>
<td>Stateful</td>
</tr>
<tr>
<td>Replicate stateless processing</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>+ Limit stateful processing to single pipeline</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>+ Dynamic state sharding</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>&amp; Feed-forward pkt steering</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>+ Preemptive state access order enforcement</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Performance Evaluation
Sensitivity Analysis

- Normalized Throughput vs. Number of Pipelines
- Normalized Throughput vs. Number of Stateful Pipeline Stages
- Normalized Throughput vs. Register Size (Number of entries)
- Normalized Throughput vs. Packet Size (Bytes)

Legend:
- Ideal (uniform)
- Ideal (skewed)
- MP5 (uniform)
- MP5 (skewed)
Realistic Workloads & Applications

- Flowlet Switching
- CONGA load balancing
- Priority calculation for WFQ
- Network Sequencer
Summary

Functional Equivalence
Runtime behavior of program same as on a single large pipeline

Performance Equivalence
Program runs as close to rate of a single large pipeline, i.e., R/4 w/o violating functional equivalence
Summary

Functional Equivalence
Runtime behavior of program same as on a single large pipeline

Performance Equivalence
Program runs as close to rate of a single large pipeline, i.e., R
w/o violating functional equivalence

Map

MP5

Rate: R

Rate: R/4
Summary

**Functional Equivalence**
Runtime behavior of program same as on a single large pipeline.

**Performance Equivalence**
Program runs as close to rate of a single large pipeline, i.e., R/4, w/o violating functional equivalence.

Logical single large pipeline ➔ Code ➔ Rate: R

---

Rate: R/4

---

[Diagram with flow and annotations]
Summary

Functional Equivalence
Runtime behavior of program same as on a single large pipeline

Performance Equivalence
Program runs as close to rate of a single large pipeline, i.e., R

Dynamically shard shared state based on runtime state access pattern

Rate: R/4
Summary

Functional Equivalence
Runtime behavior of program same as on a single large pipeline

Performance Equivalence
Program runs as close to rate of a single large pipeline, i.e., $R$ w/o violating functional equivalence

Dynamically shard shared state based on runtime state access pattern

Preemptively enforce state access order
Thank you!