SIGCOMM 2022
Amsterdam

Artifacts Available V1.1
Artifacts Evaluated Functional V1.1
Results Reproduced V1.1

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:

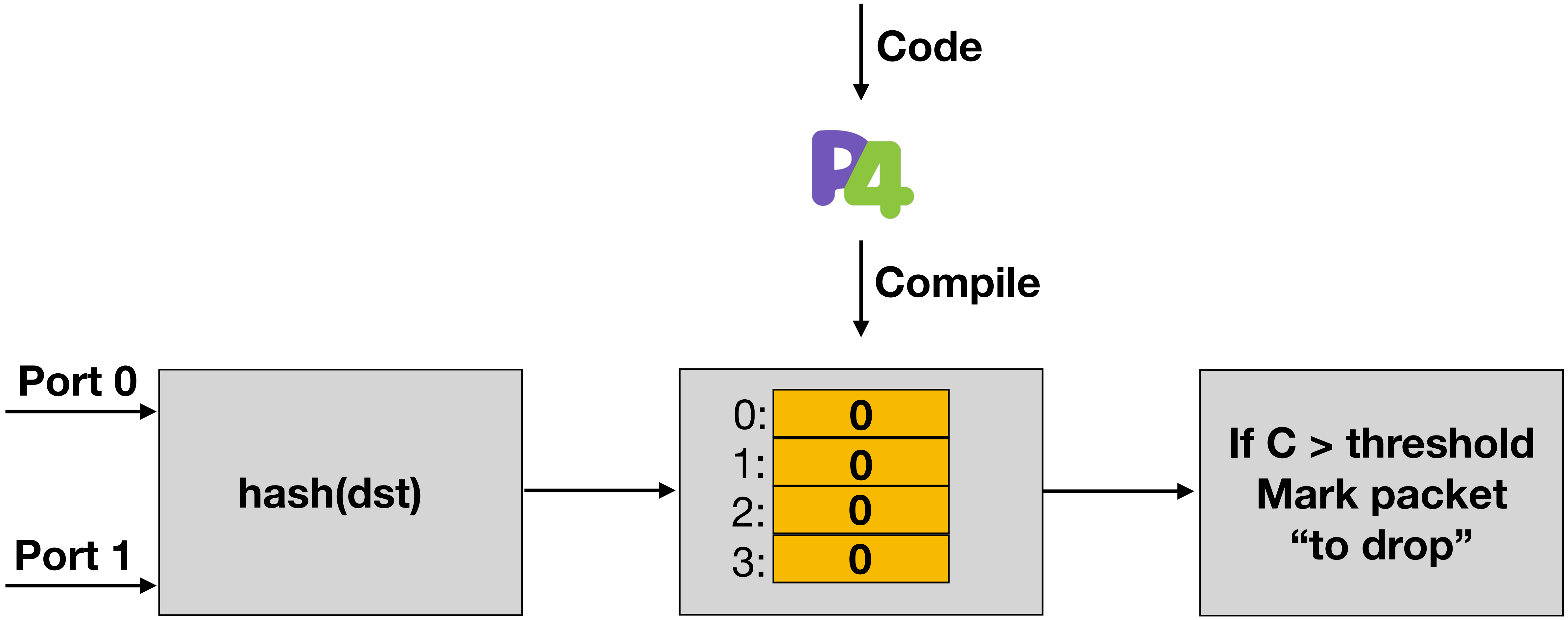
- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
    - Switch drops all subsequent packets destined to d

$\Big\downarrow$ **Code**

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

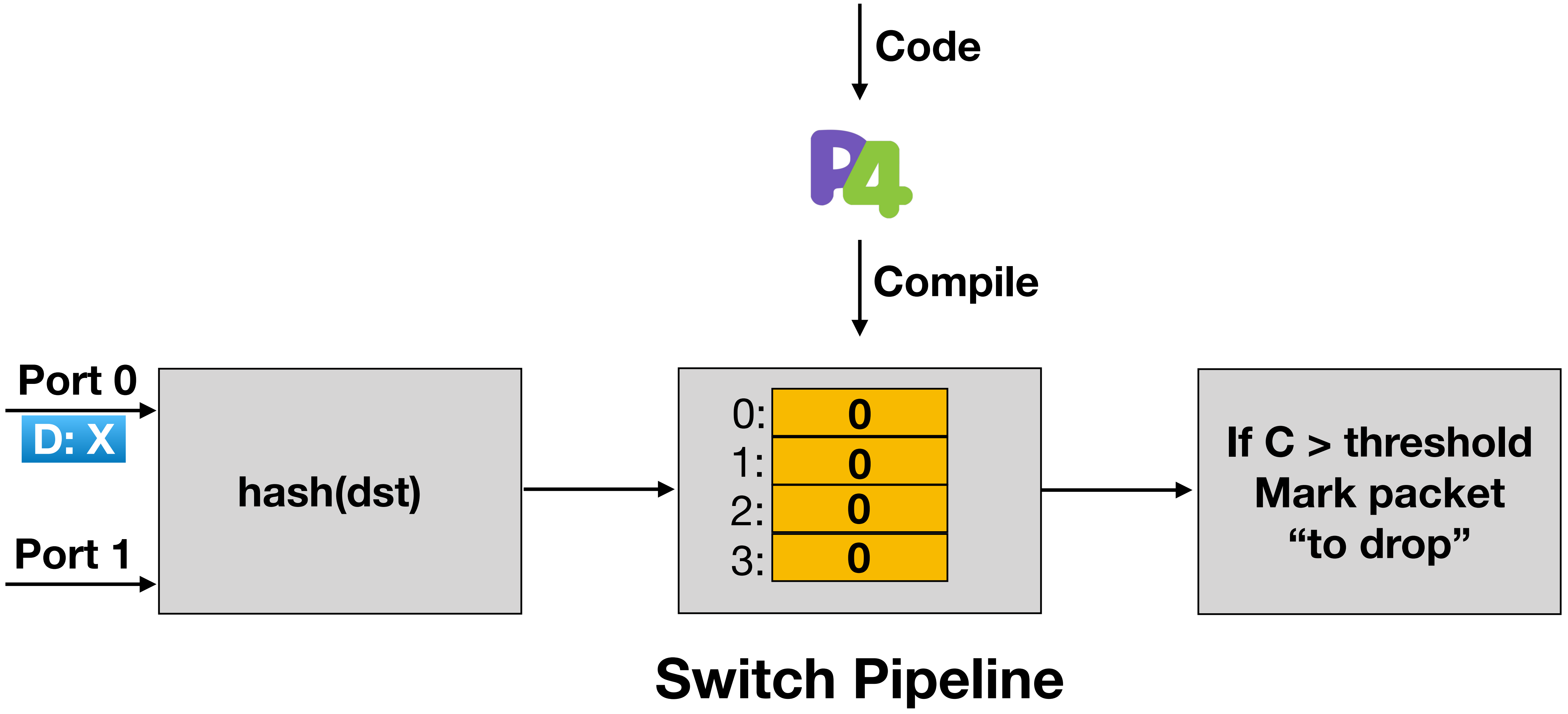


Switch Pipeline

Motivating Example

Consider a packet processing program:

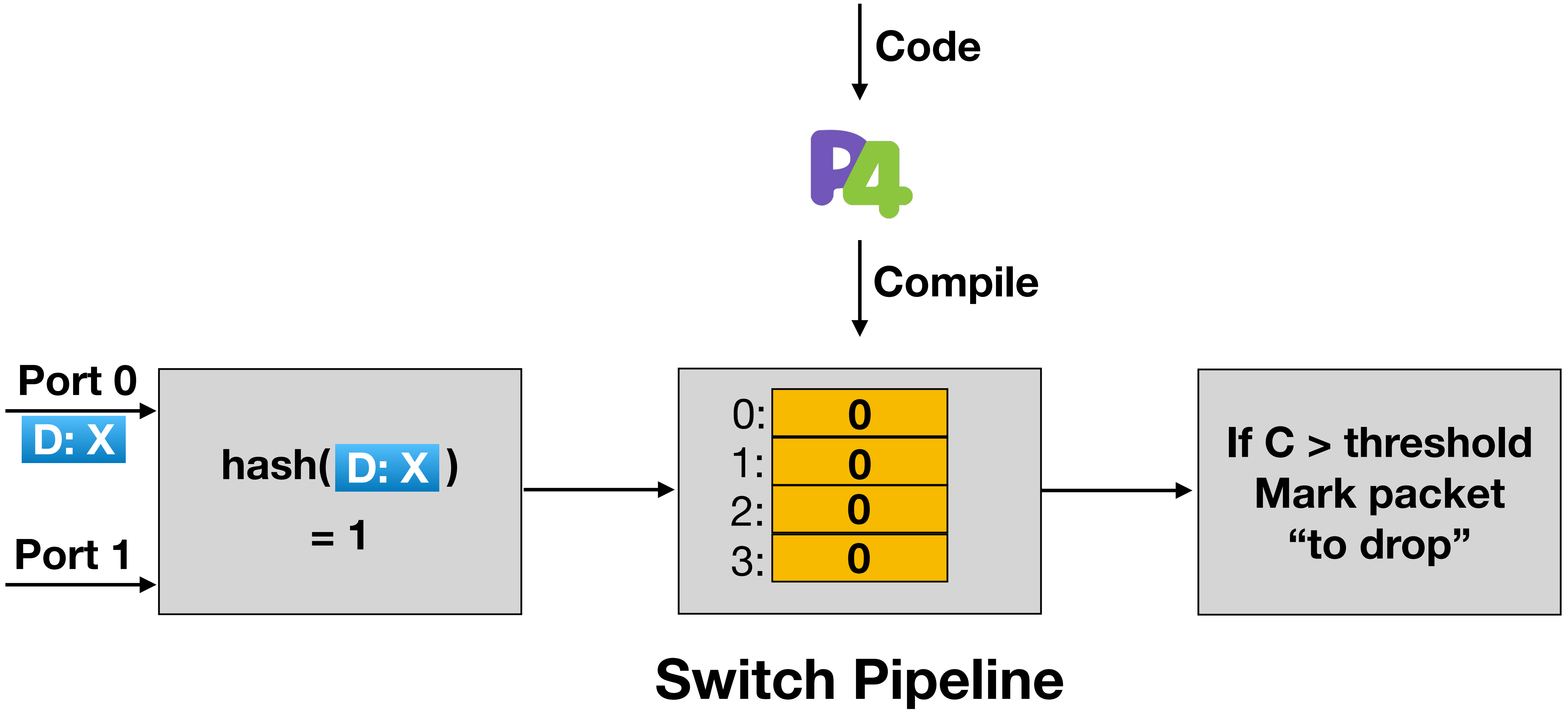
- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
    - Switch drops all subsequent packets destined to d



Motivating Example

Consider a packet processing program:

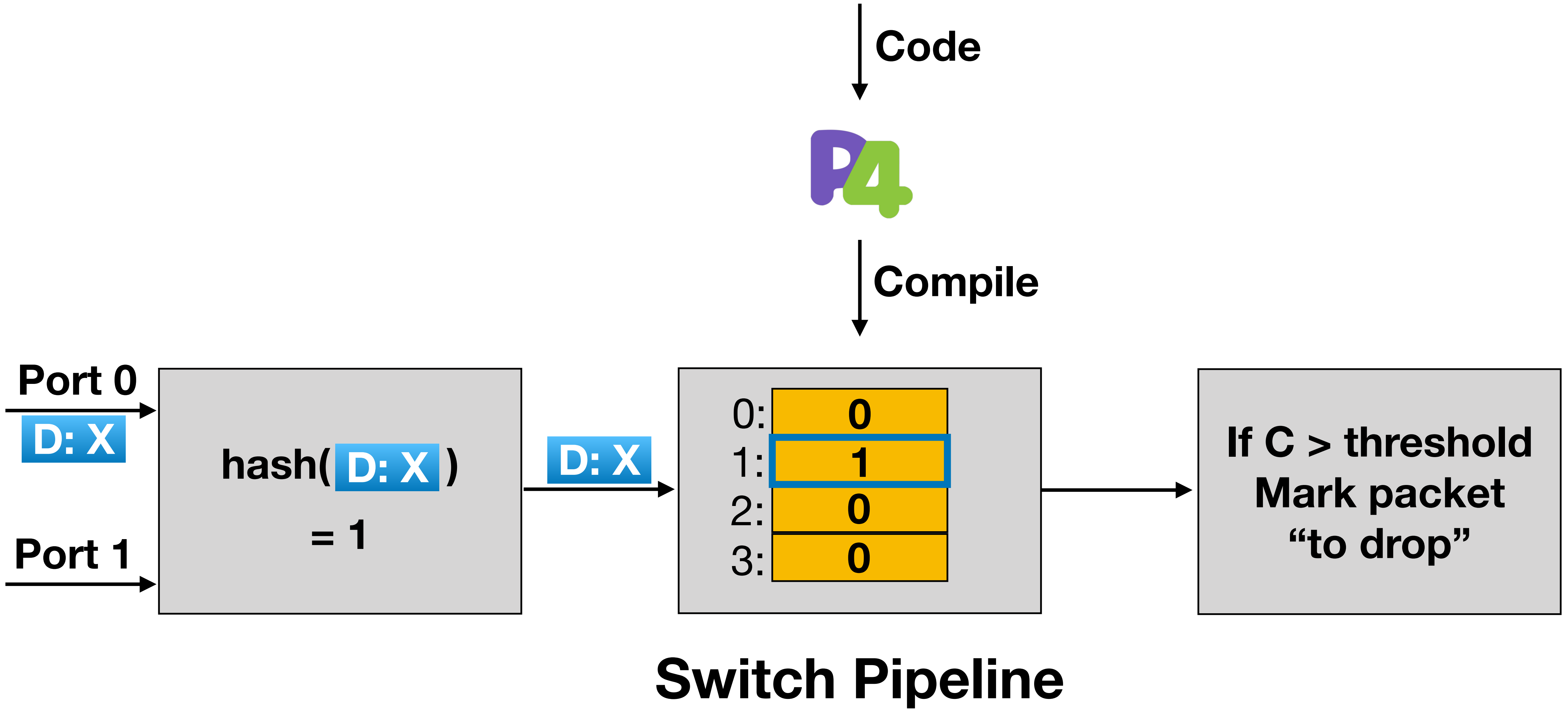
- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
  - Switch drops all subsequent packets destined to d



Motivating Example

Consider a packet processing program:

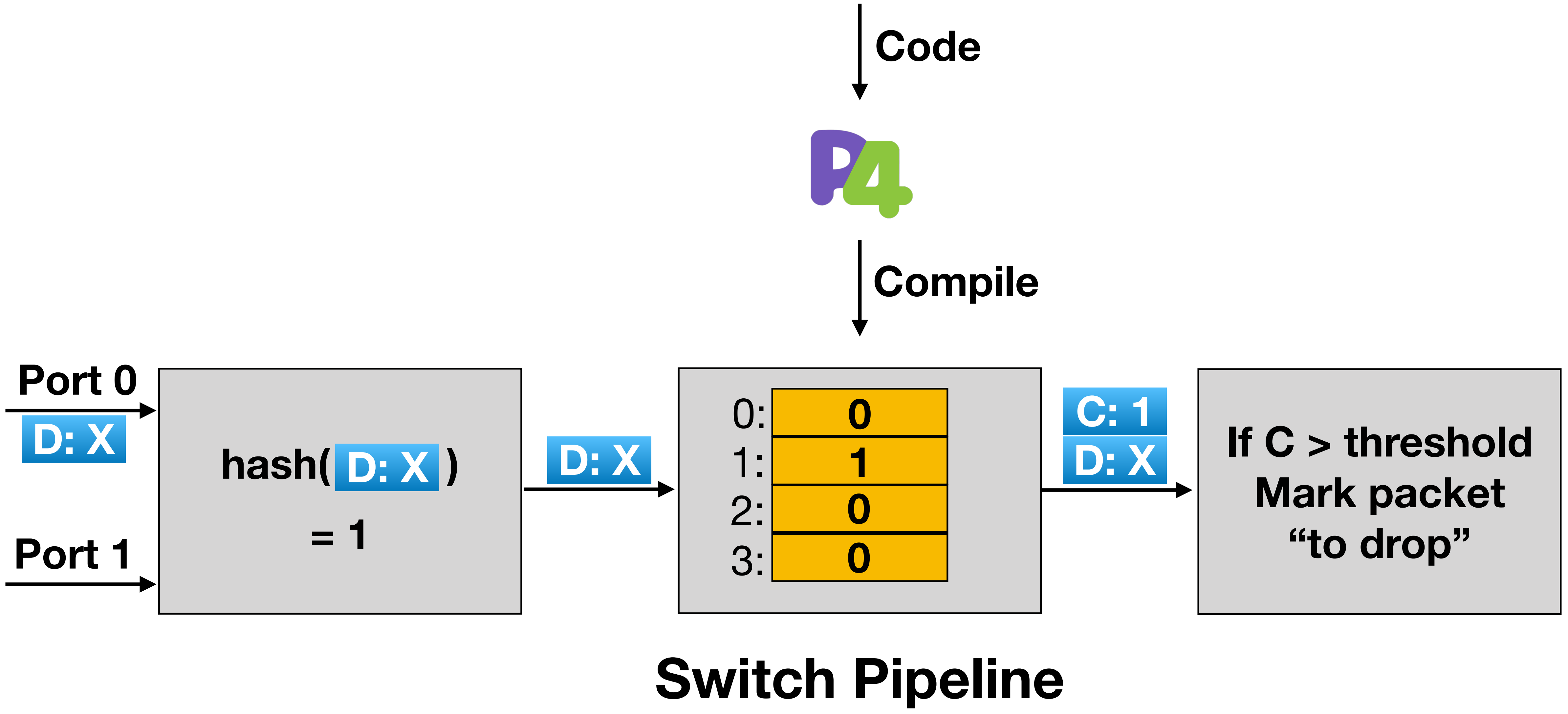
- Switch maintains packet counters for each destination IP
- If the counter value for destination d exceeds a threshold
    - Switch drops all subsequent packets destined to d



Motivating Example

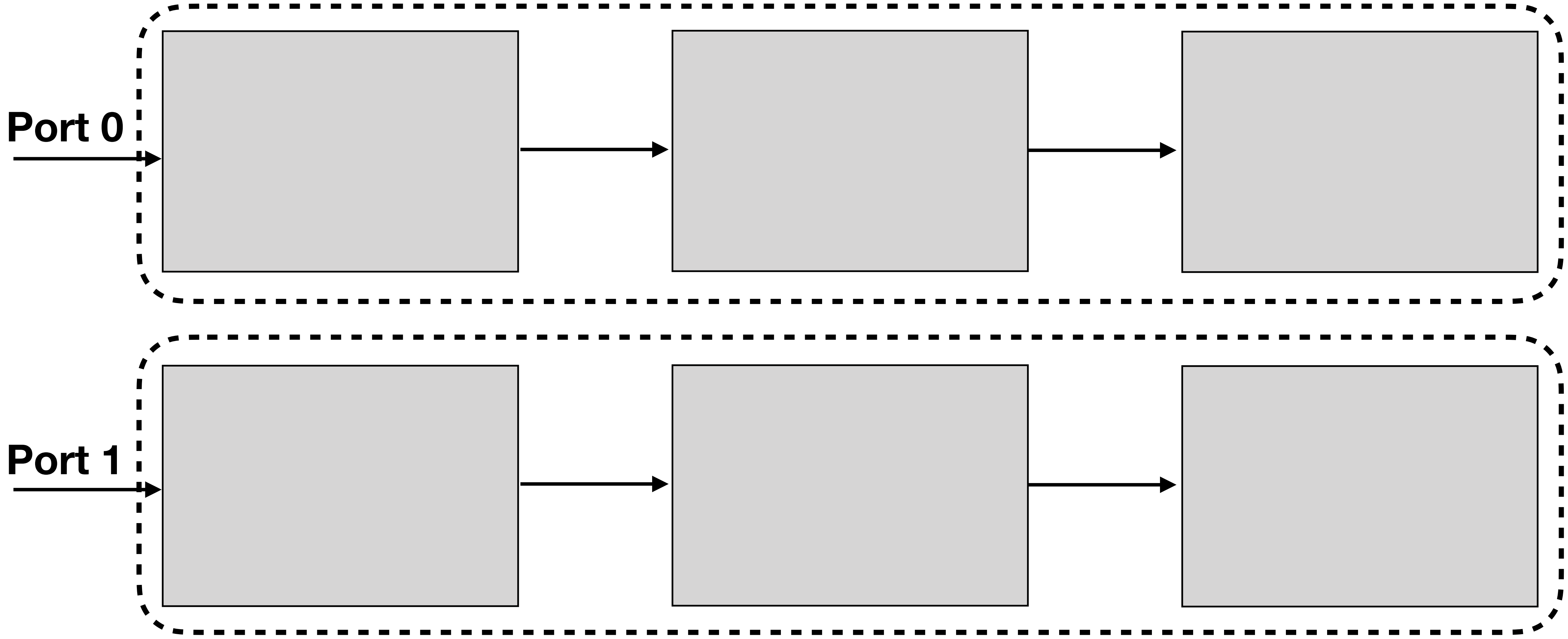
Consider a packet processing program:

- 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

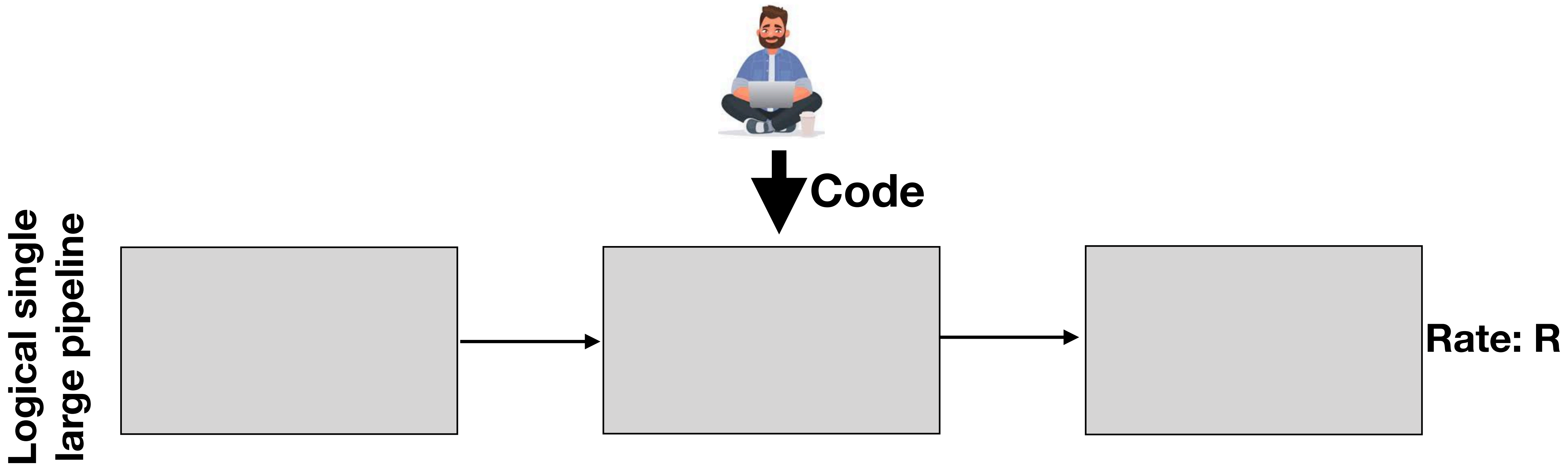


Reality of Today's Switch Hardware

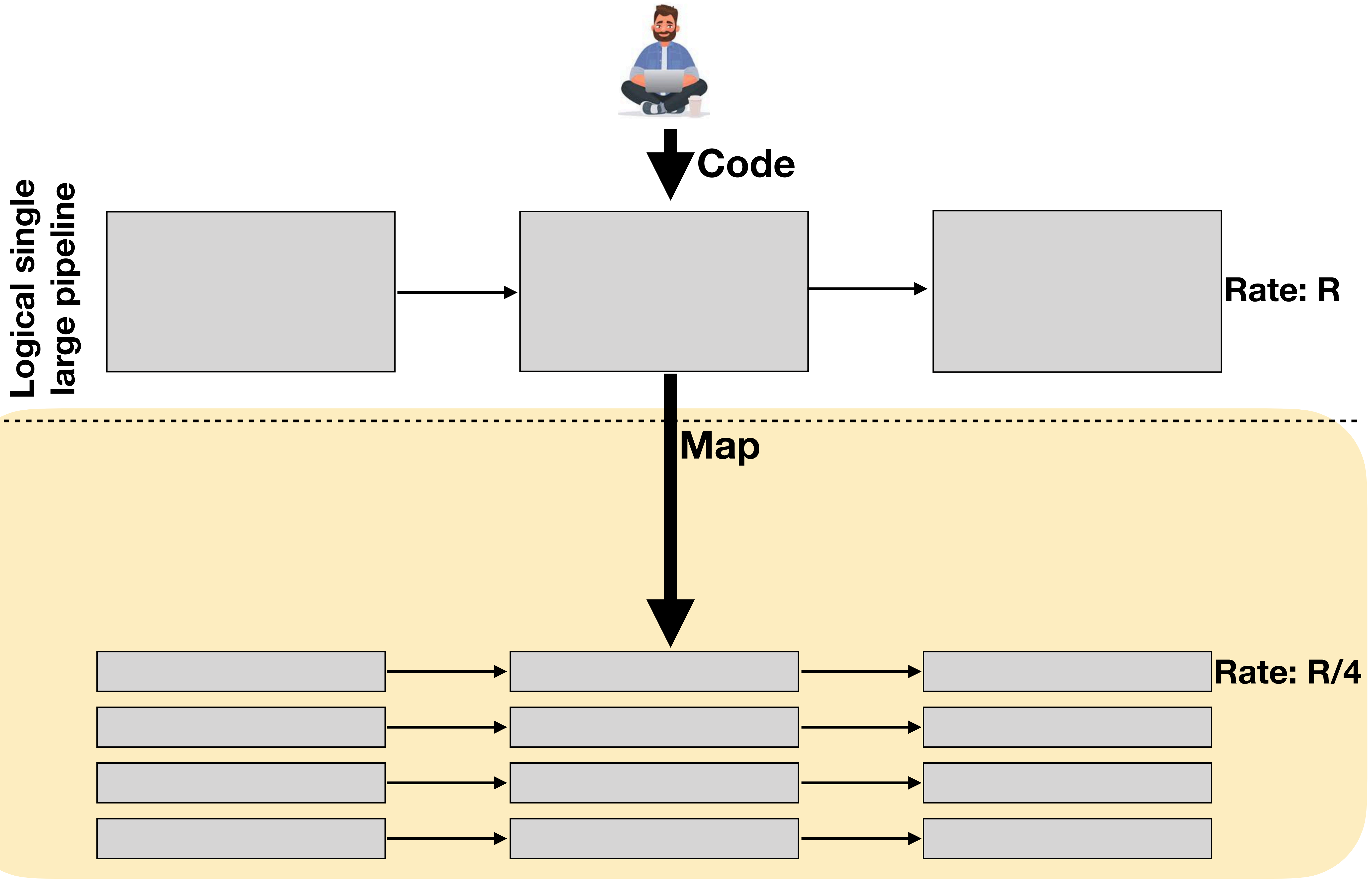
- Clock speed of a single pipeline has saturated
  - Limits the line rate
- Employ multiple **parallel pipelines** to sustain multi-tbps line rate
  - Each pipeline processes packets **independently** — No co-ordination



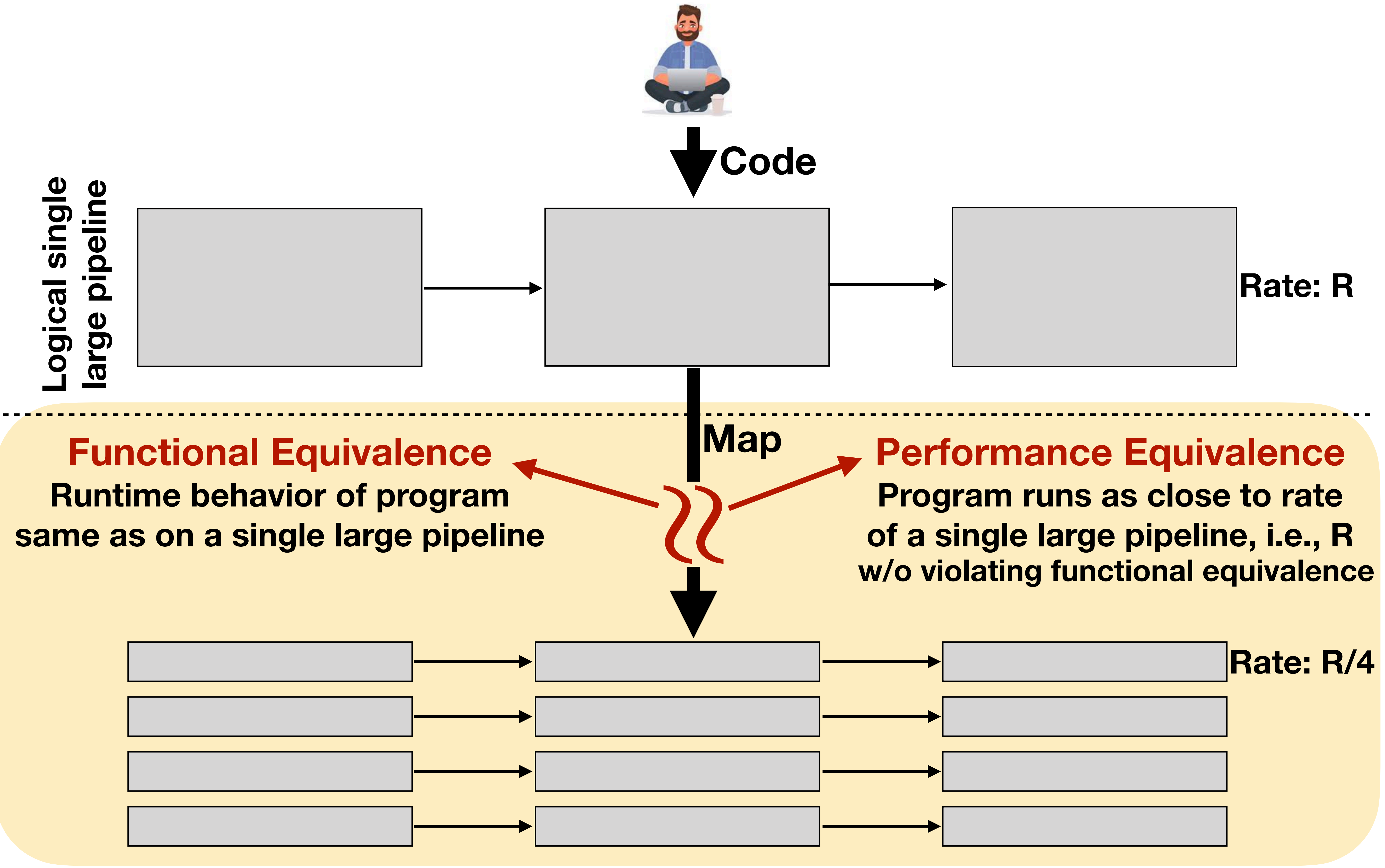
Goal



Goal



Goal


Code
Logical single large pipeline
Rate: R
Map
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

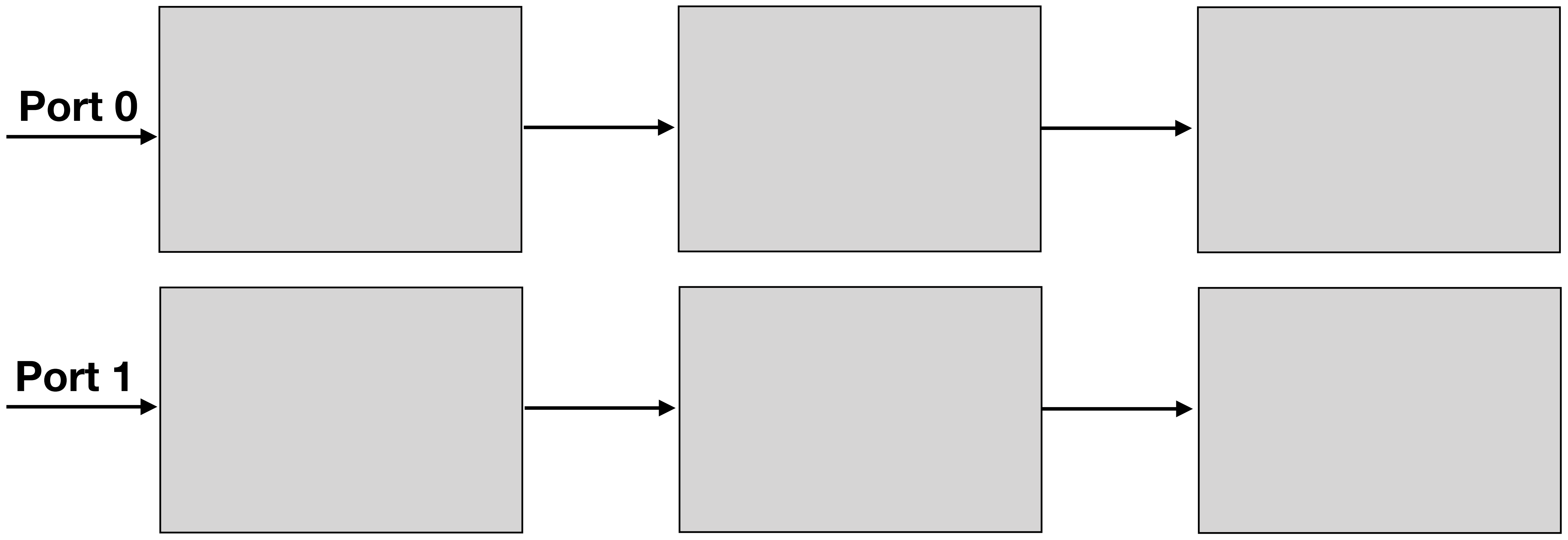
Our Contribution

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:

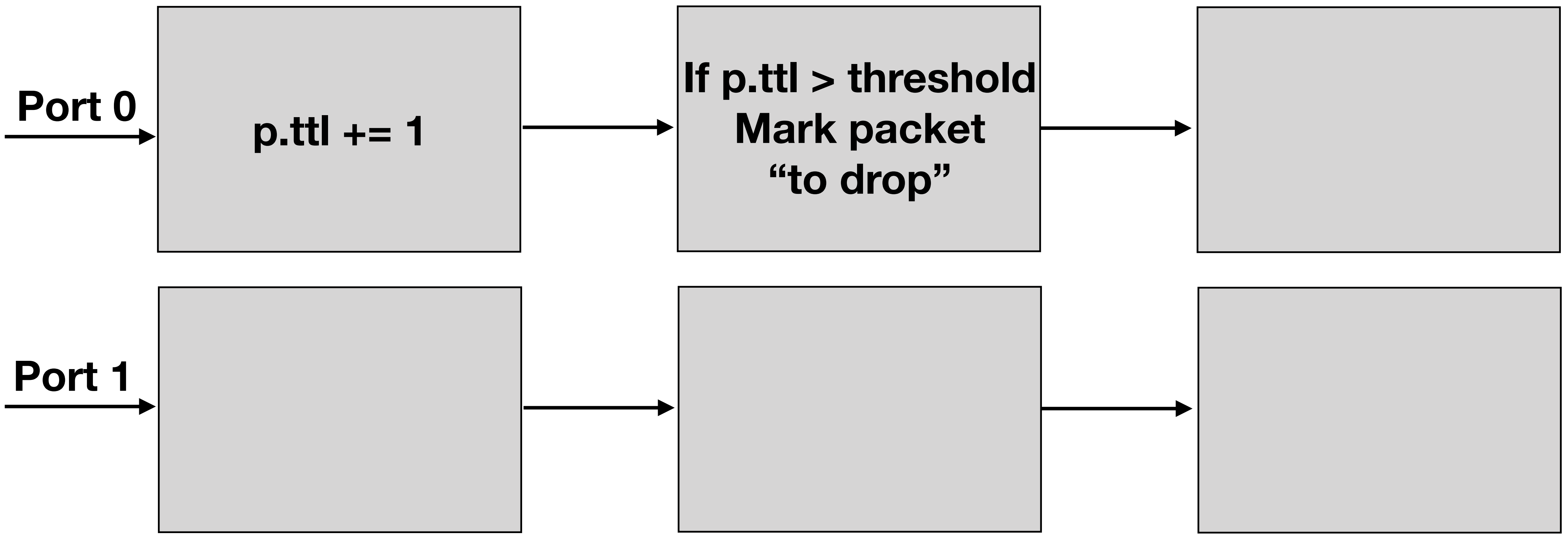
- Switch increments the ttl value in packet header by 1
- If ttl value exceeds a threshold
    - Switch drops the packet



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

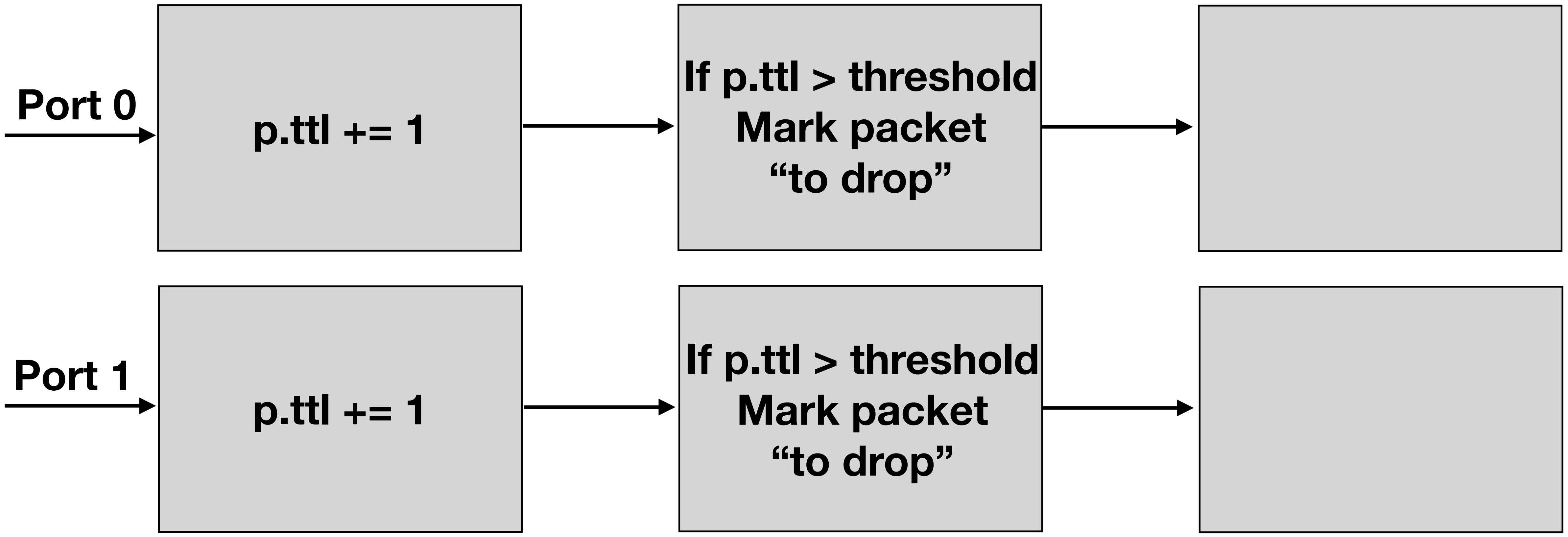


Naive Approaches

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

Try 1: Replicate stateless processing on all pipelines


Port 0
p.ttl += 1
If p.ttl > threshold Mark packet "to drop"
Port 1
p.ttl += 1
If p.ttl > threshold Mark packet "to drop"

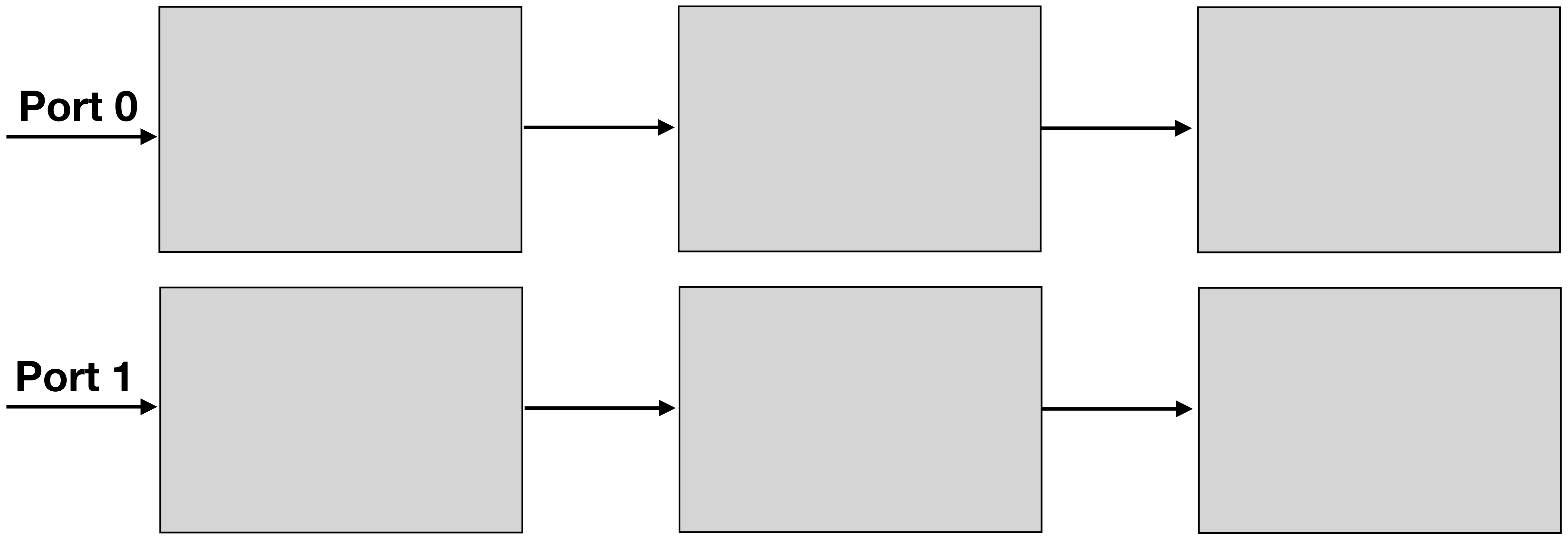
Goals and Techniques

Techniques	Functional Equivalence		Performance	
	Stateless	Stateful	Stateless	Stateful
Replicate stateless processing	✓		✓	

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

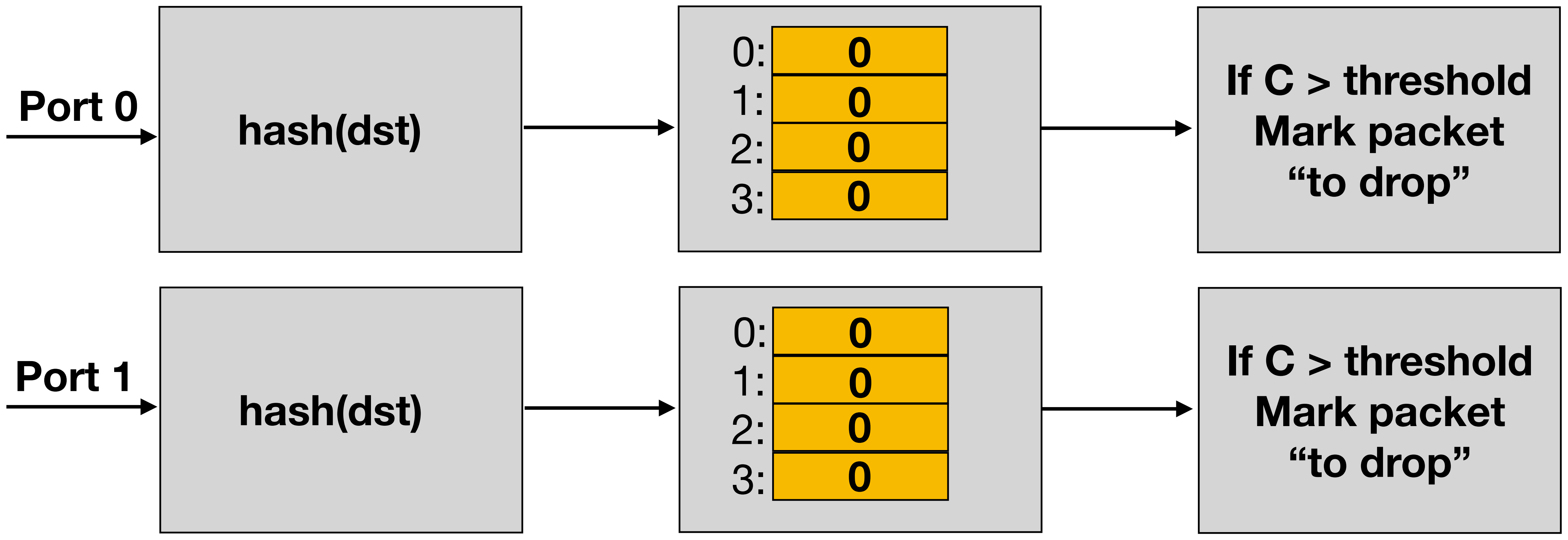


Naive Approaches

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



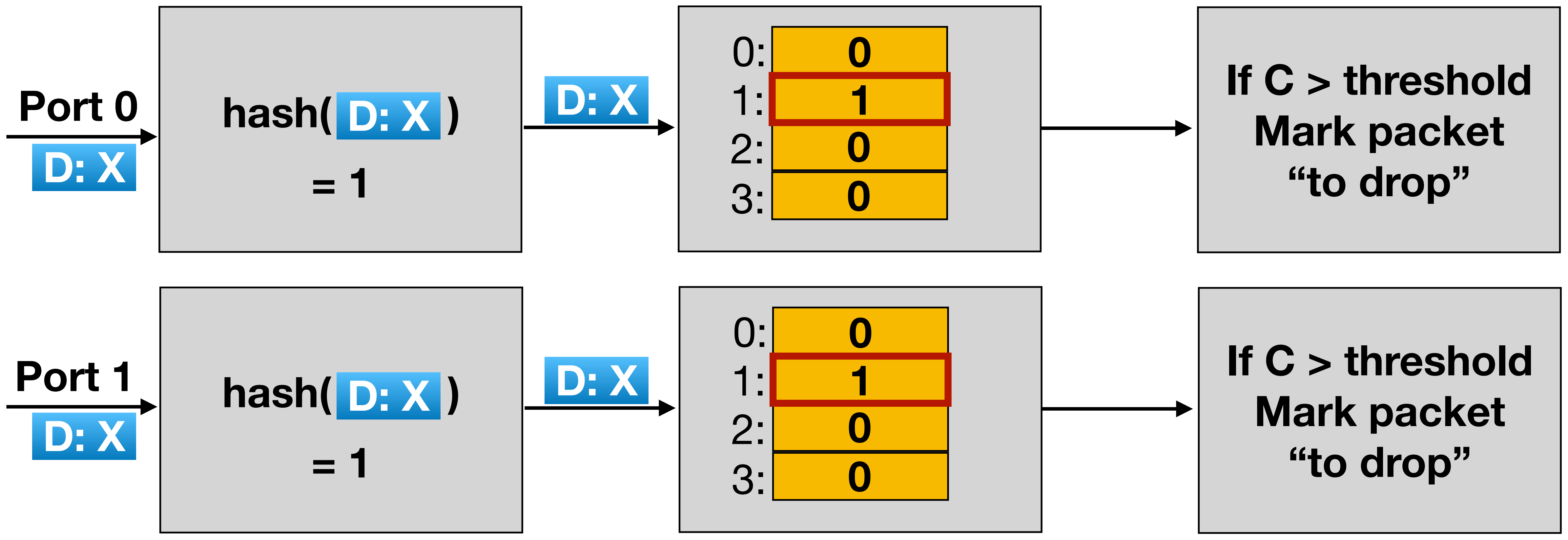
Naive Approaches

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!

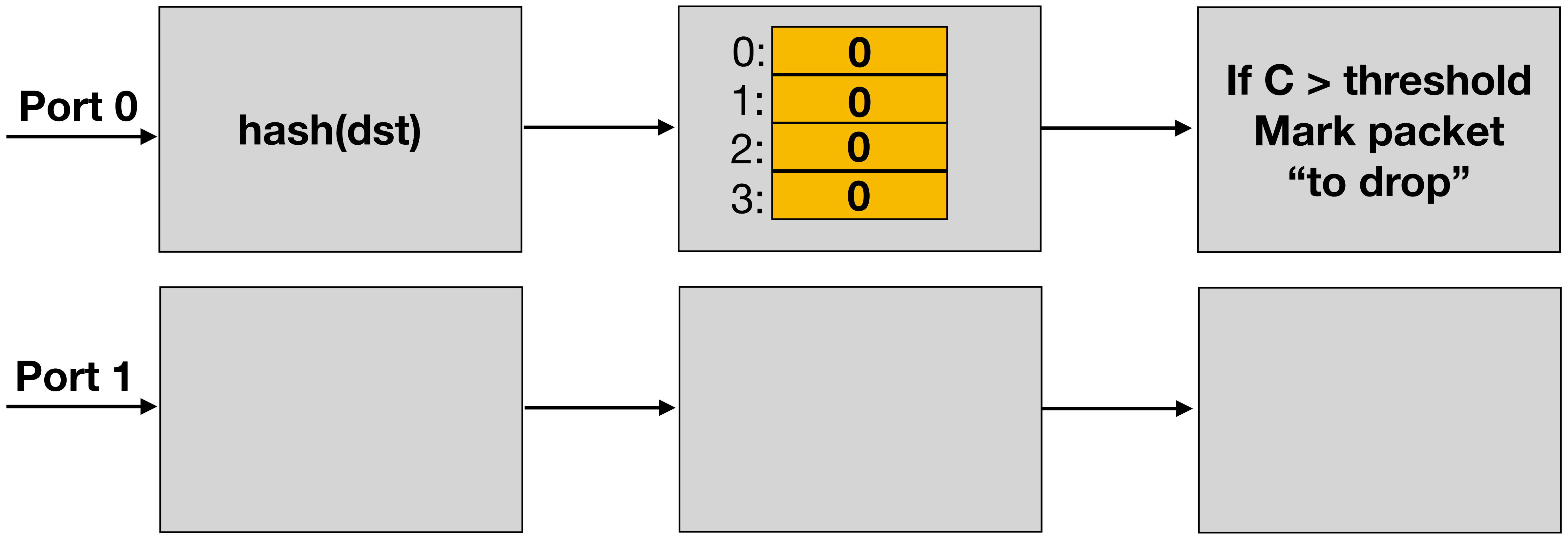


Naive Approaches

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



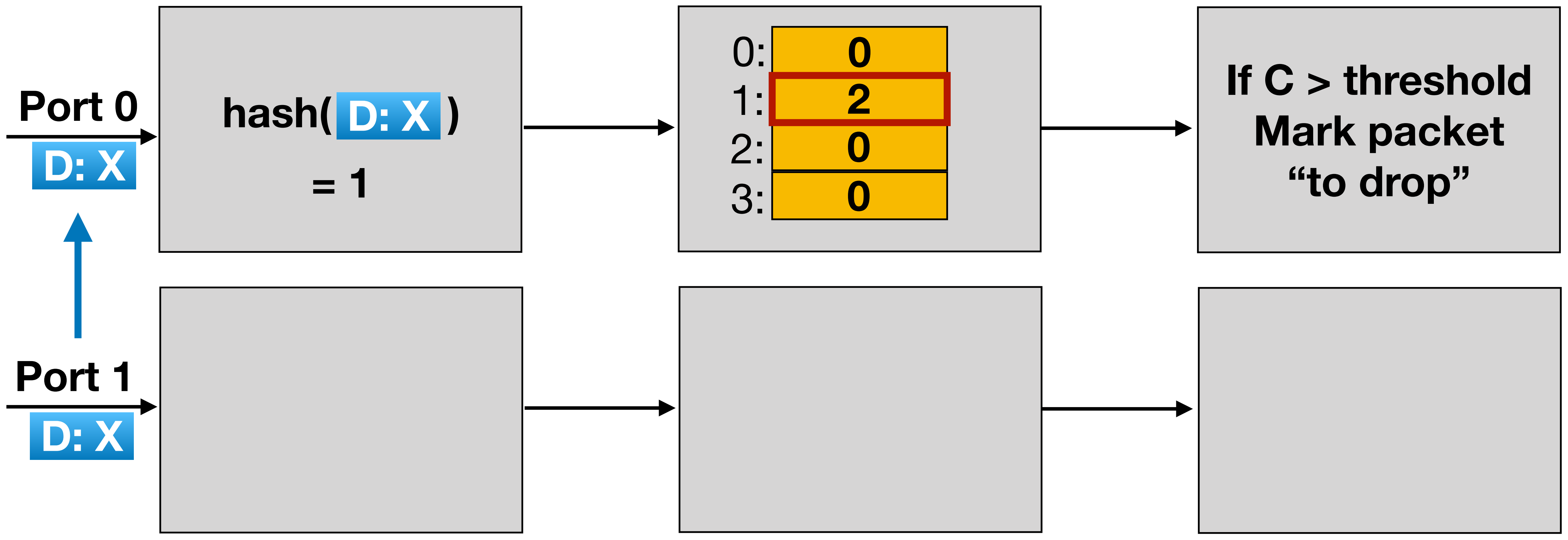
Naive Approaches

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



Naive Approaches

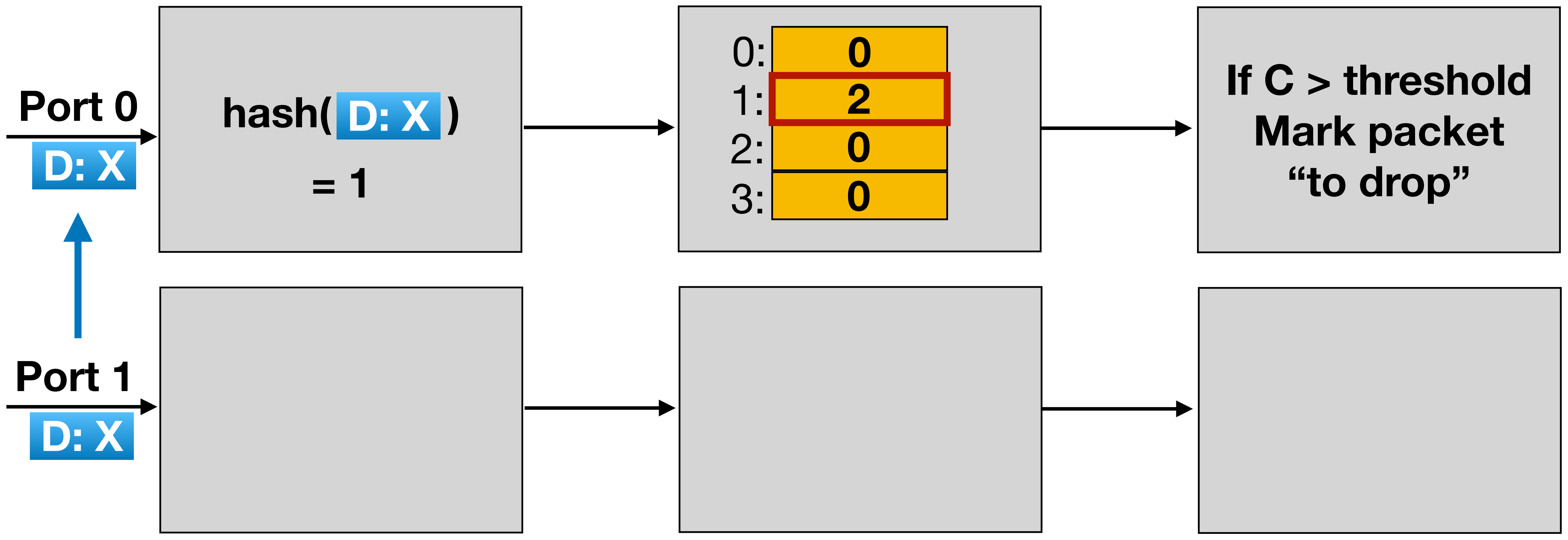
Consider a *stateful* packet processing program:

- 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

Limits speed of stateful processing!



Goals and Techniques

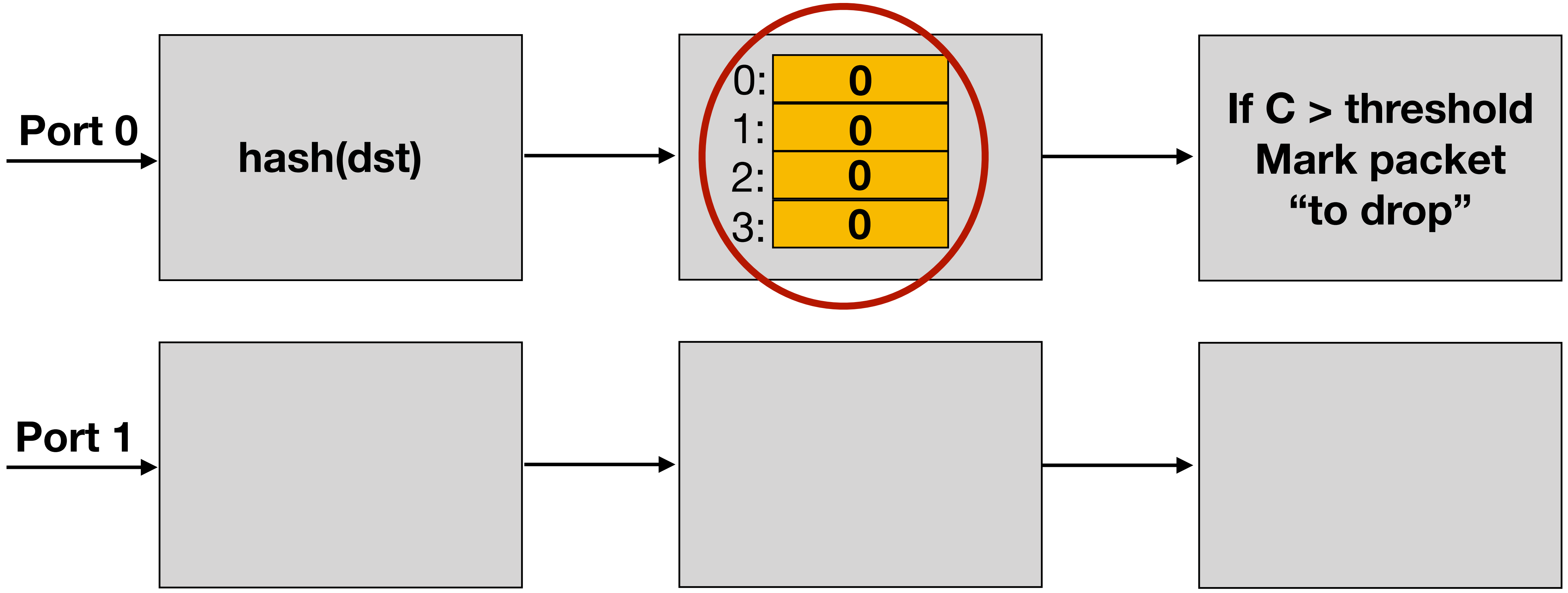
Techniques	Functional Equivalence		Performance	
	Stateless	Stateful	Stateless	Stateful
Replicate stateless processing	✓		✓	
+ Limit stateful processing to single pipeline	✓	✓	✓	✗

Question

How to improve performance?
(without violating functional equivalence)

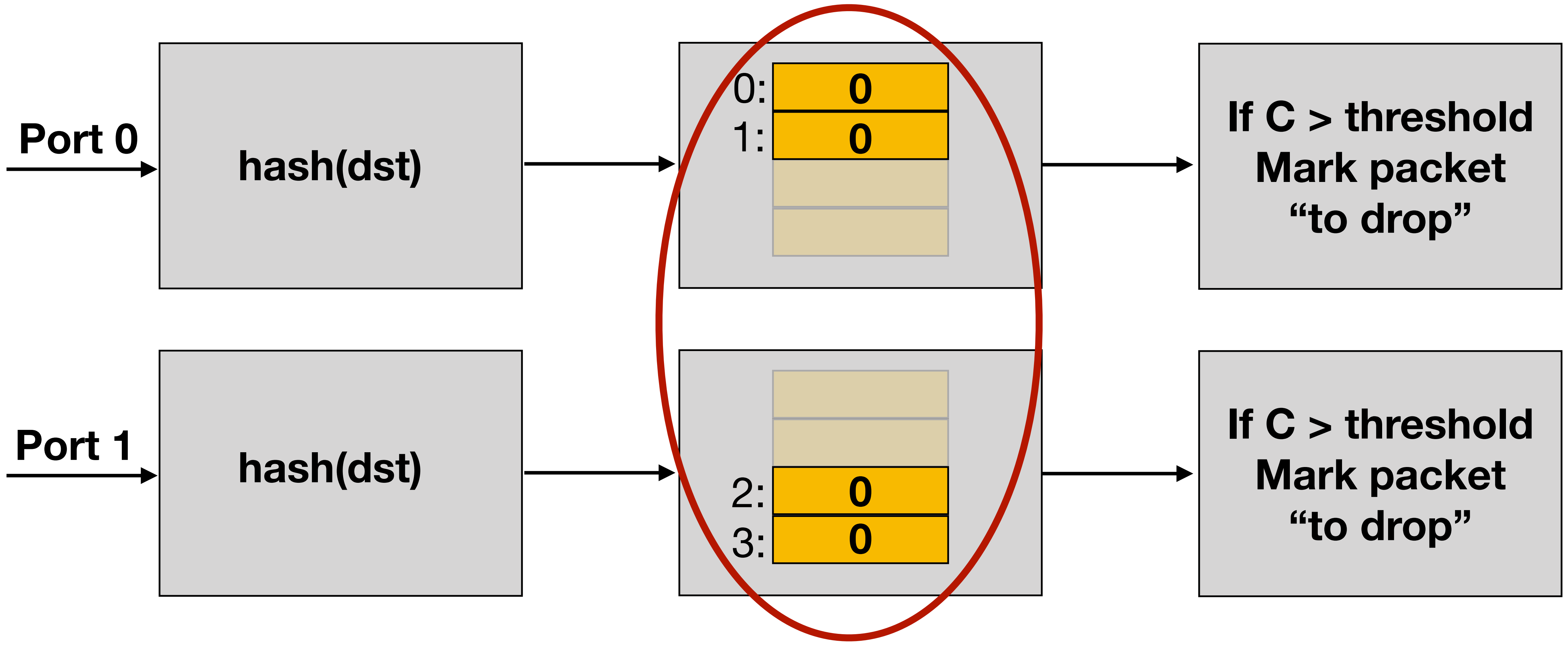
Problem

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



Solution

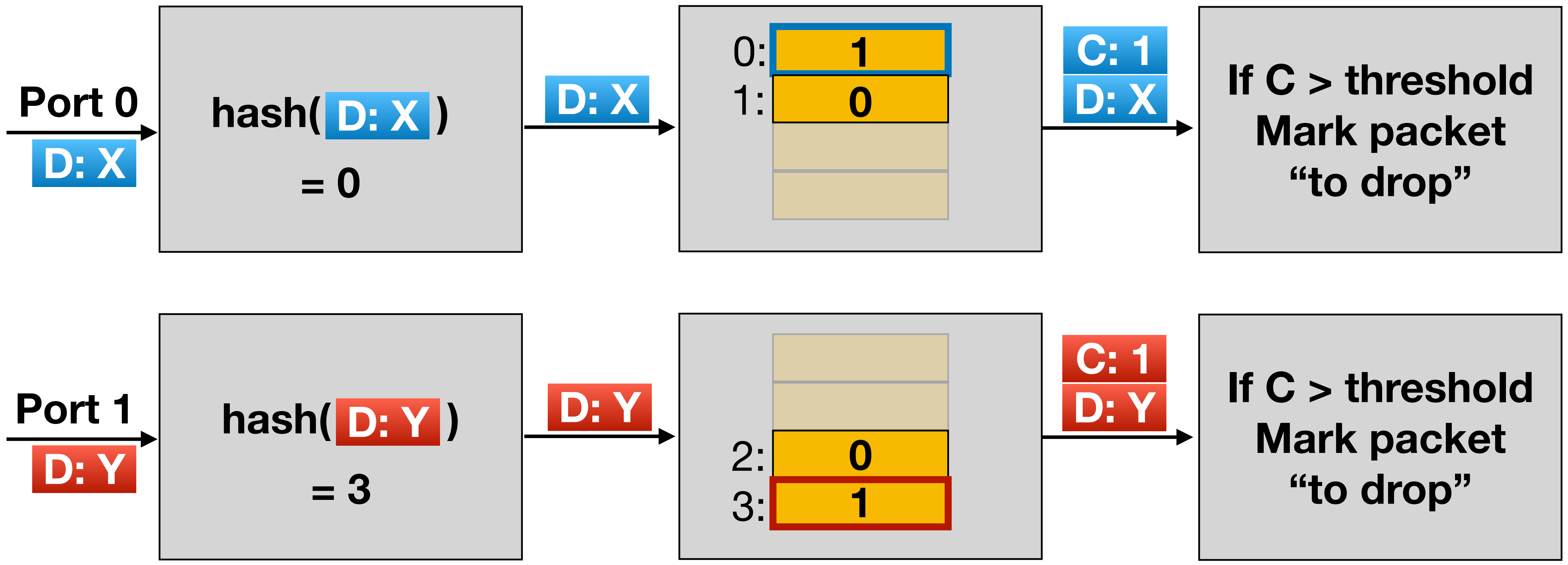
How to store shared state that enables high packet processing throughput?



Shard the shared state across pipelines

Solution

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



Shard the shared state across pipelines

Solution

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

Shard the shared state across pipelines

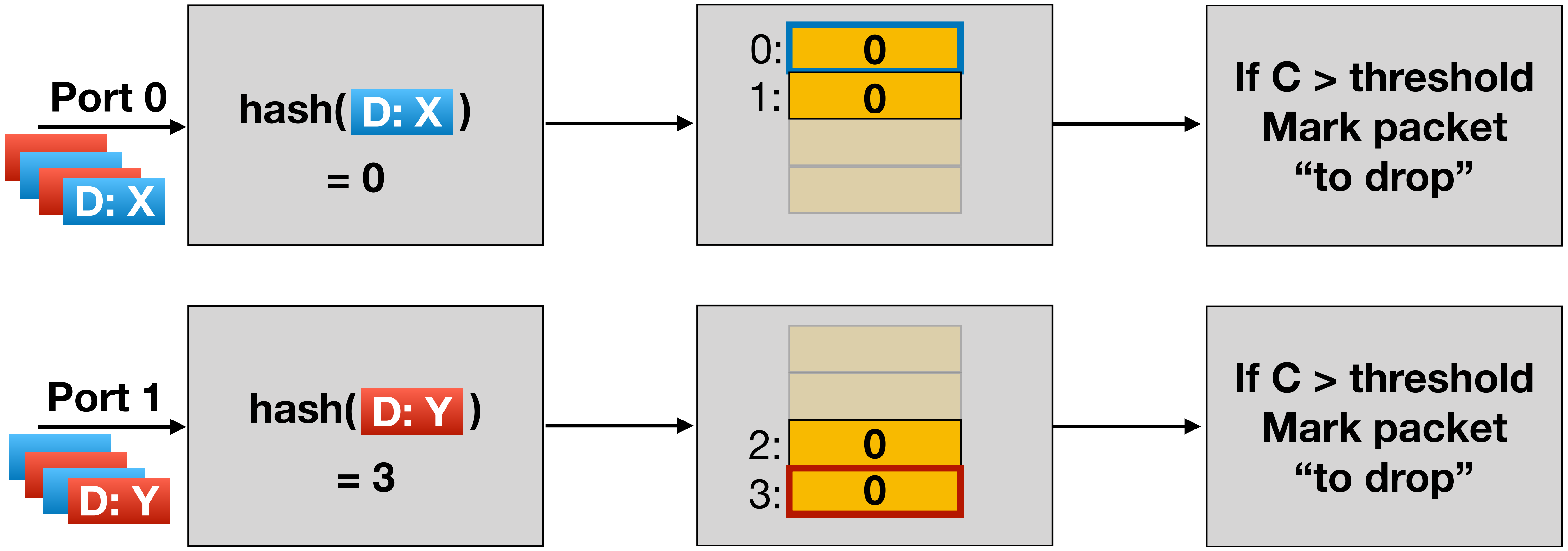
…but what is the optimal sharding strategy?

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?

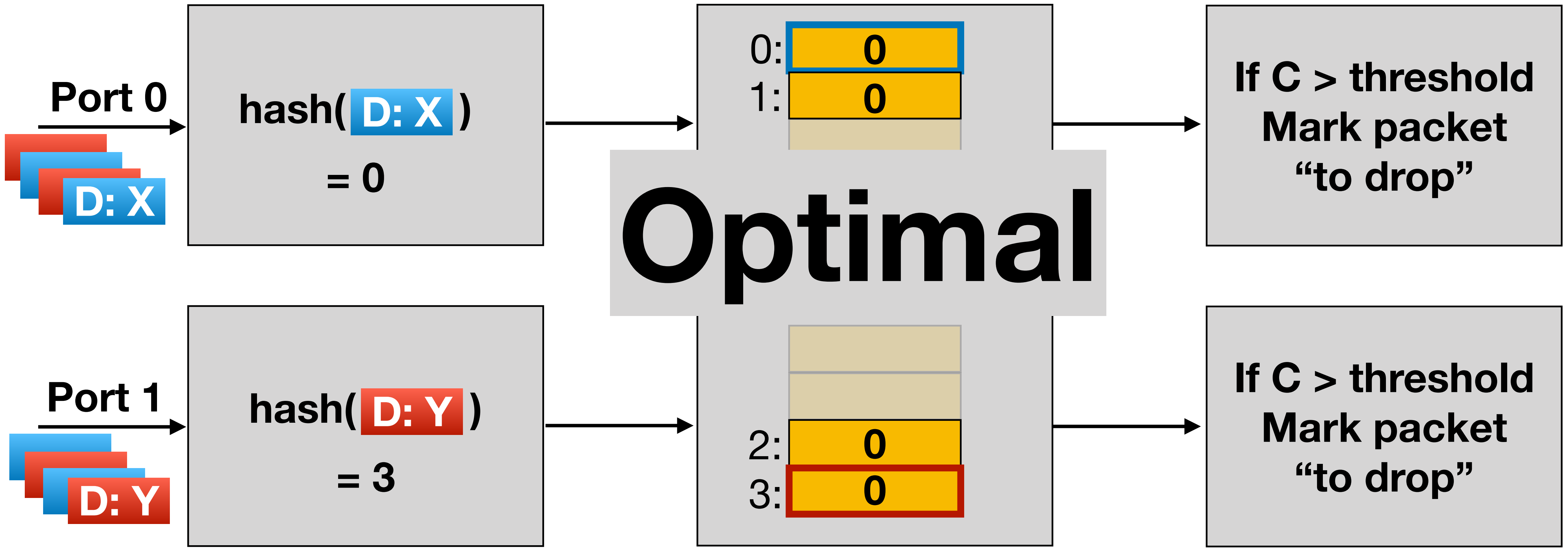


Solution

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

Shard the shared state across pipelines

…but what is the optimal sharding strategy?

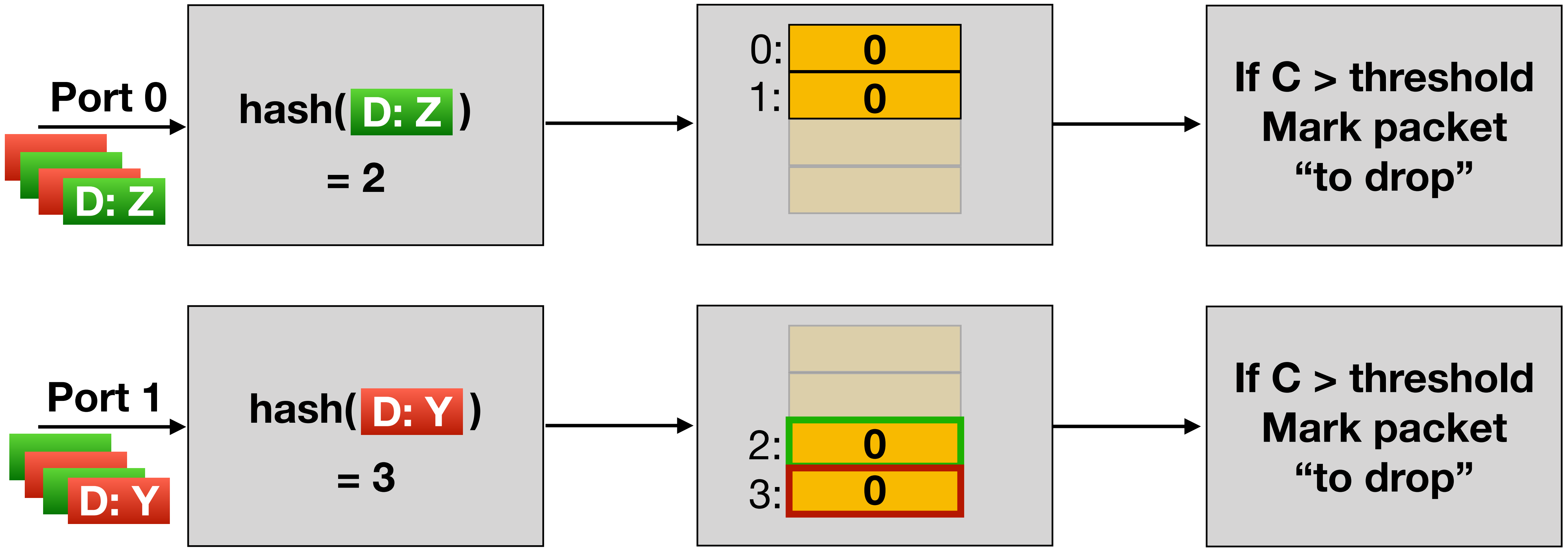


Solution

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

Shard the shared state across pipelines

…but what is the optimal sharding strategy?

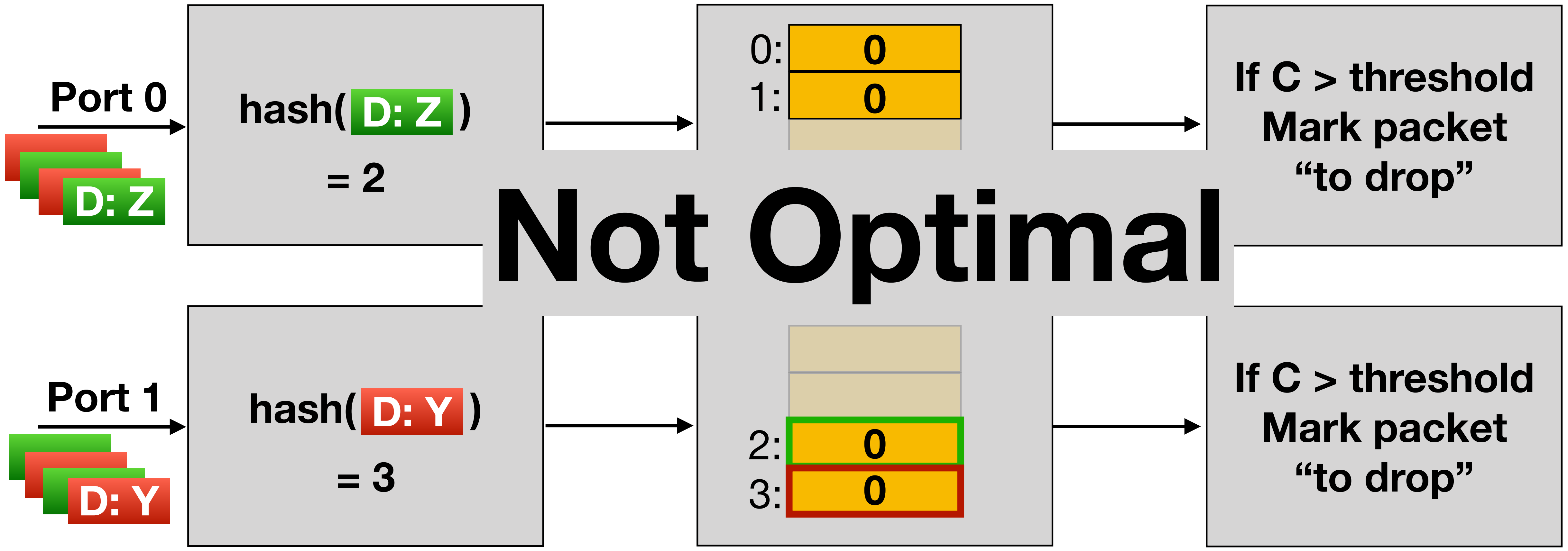


Solution

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

Shard the shared state across pipelines

…but what is the optimal sharding strategy?

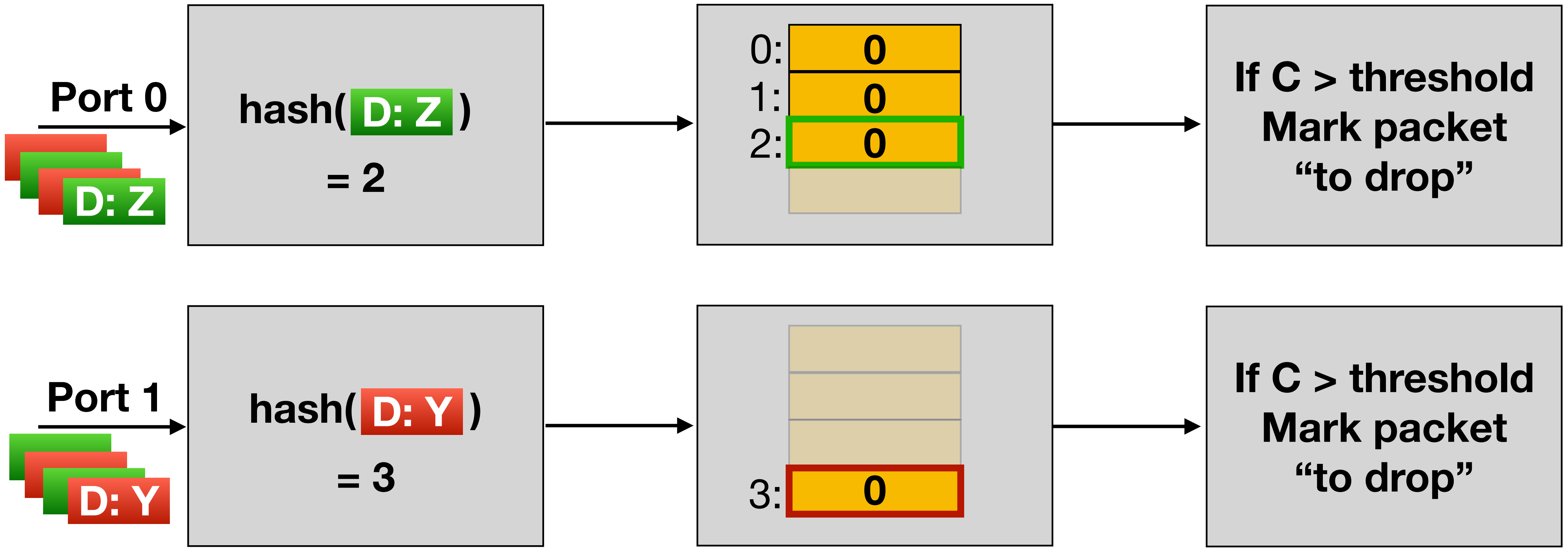


Solution

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

Shard the shared state across pipelines

…but what is the optimal sharding strategy?

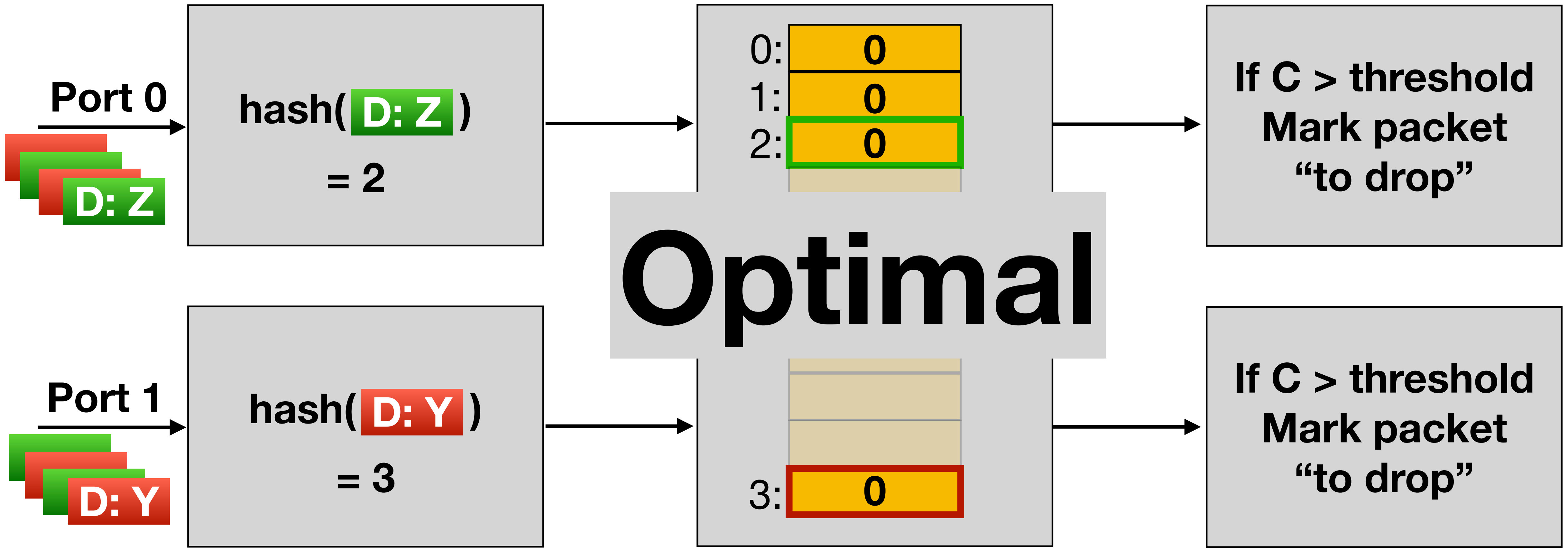


Solution

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?

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

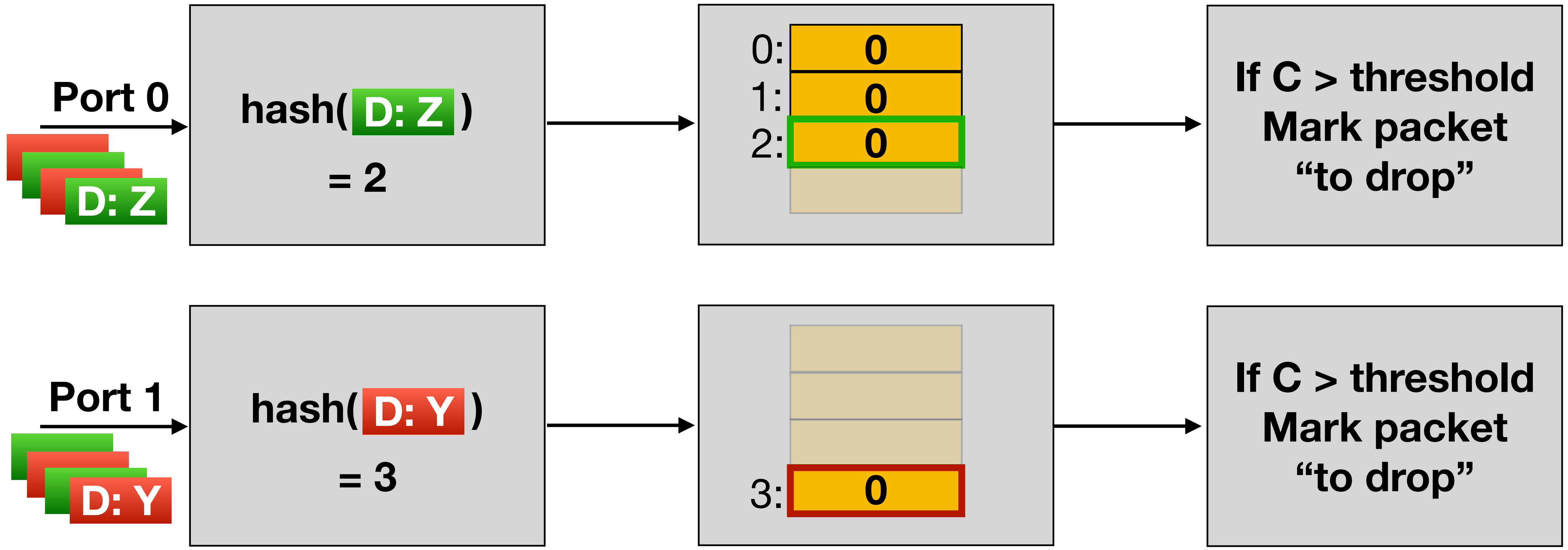
How to store shared state that enables high packet processing throughput?

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!)

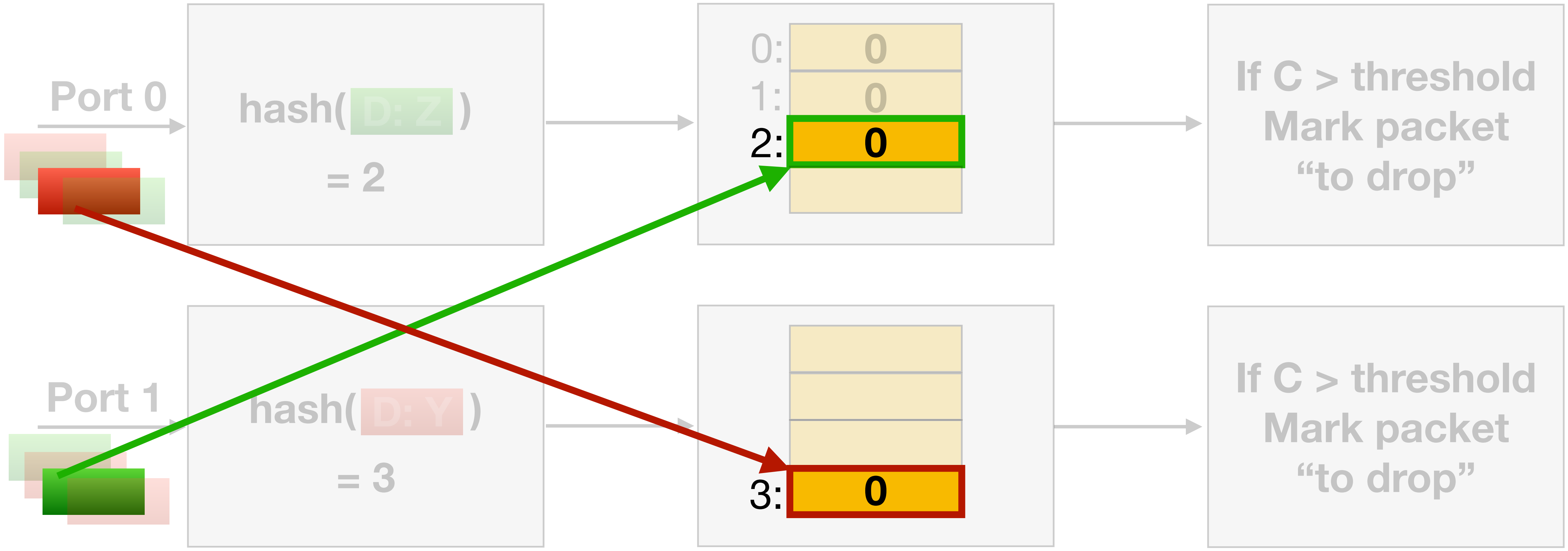
MP5 uses a heuristic to approximates bin packing that is amenable to fast hardware implementation

One Missing Detail



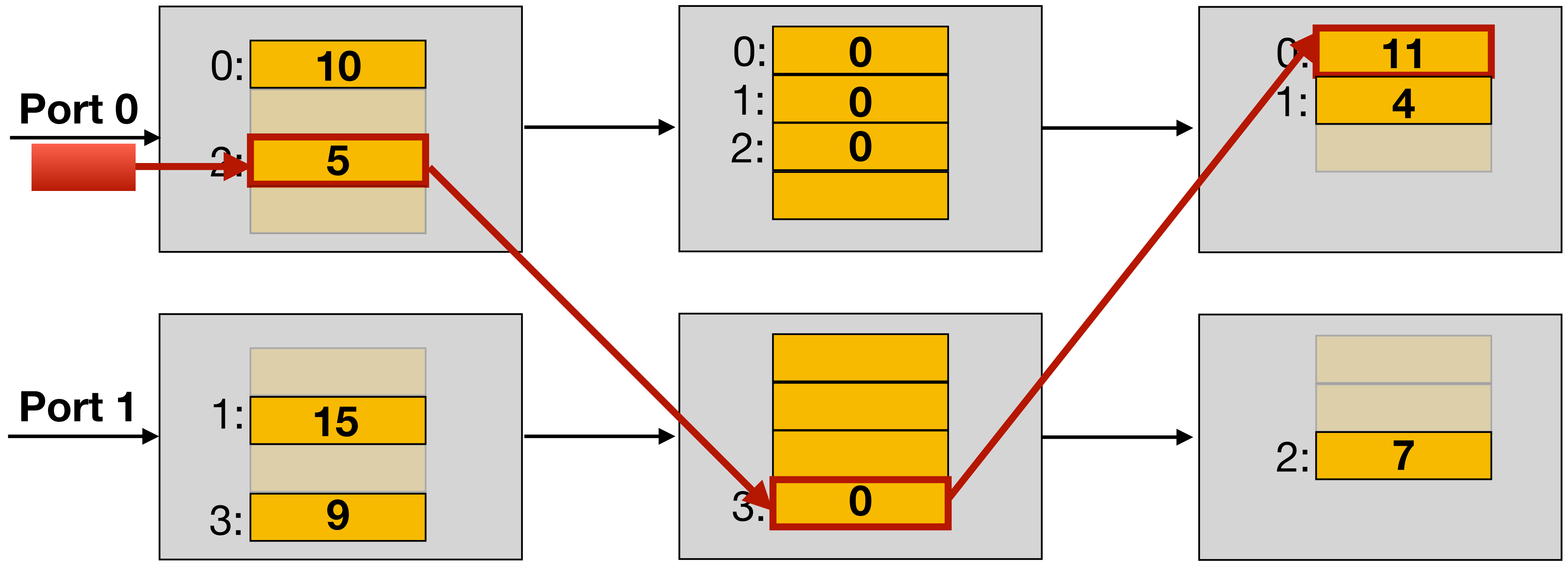
One Missing Detail

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



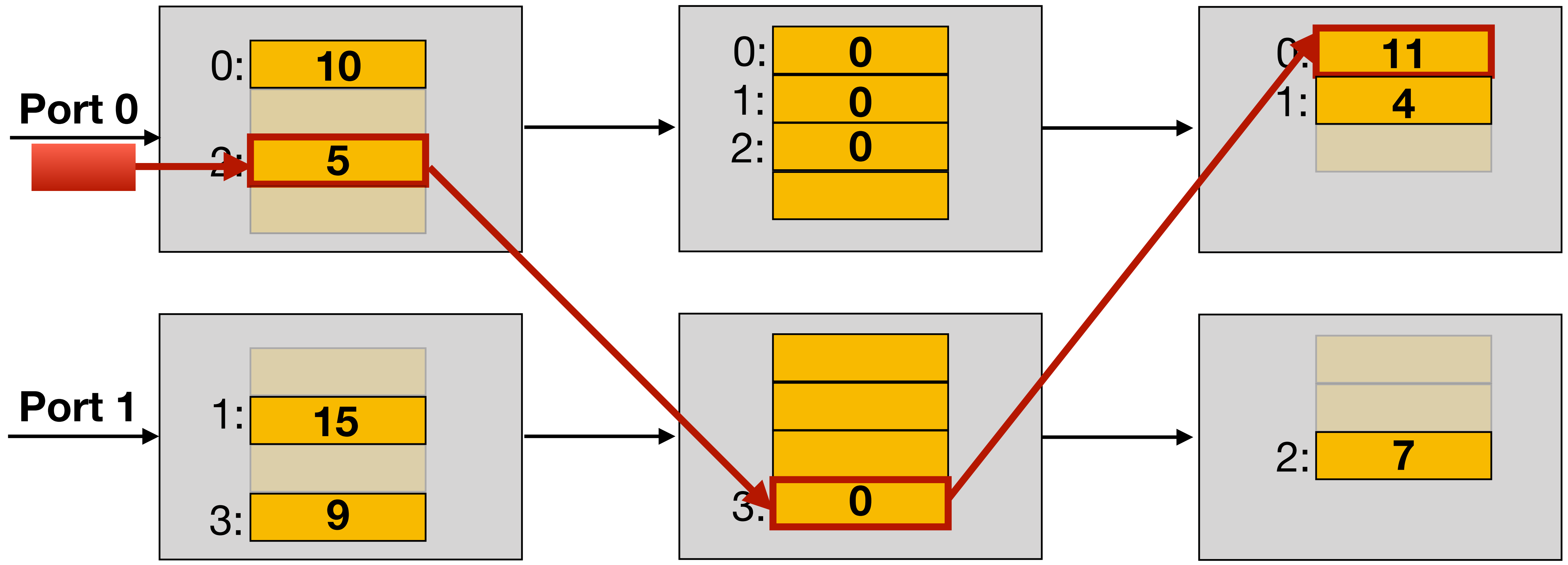
One Missing Detail

Packet may need to go back and forth between pipelines to access the shared states!



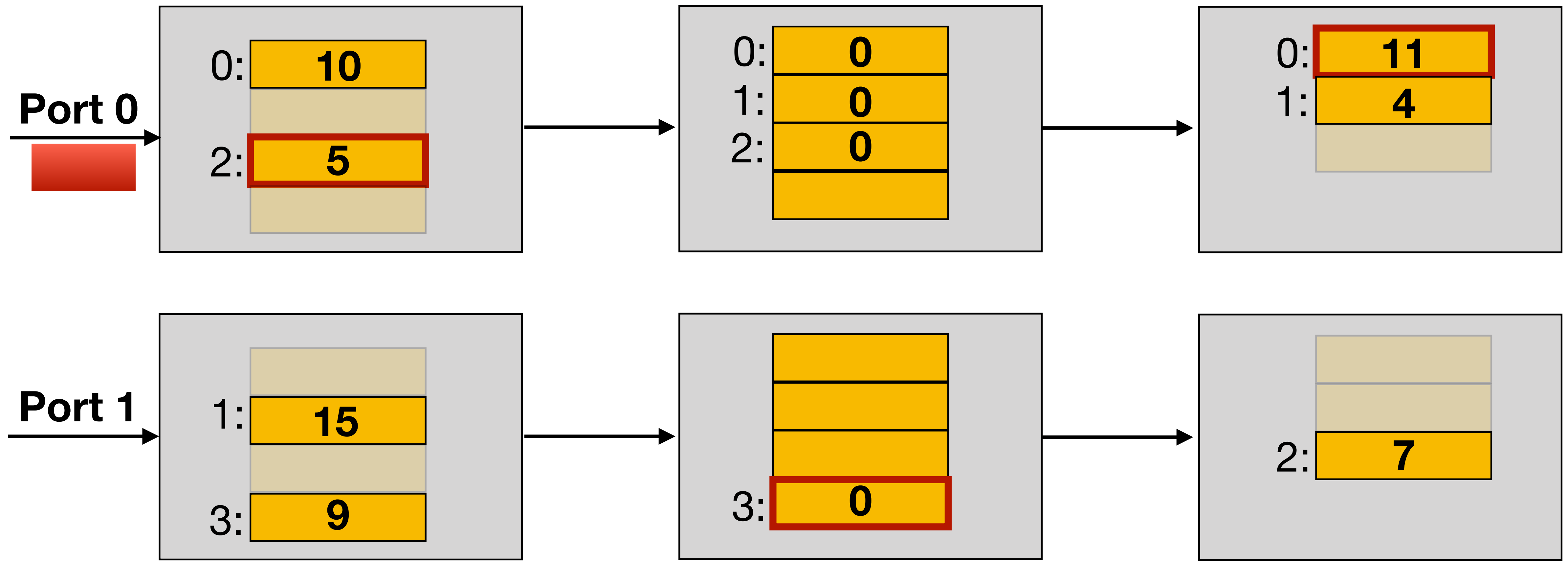
One Missing Detail

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



Existing Solution

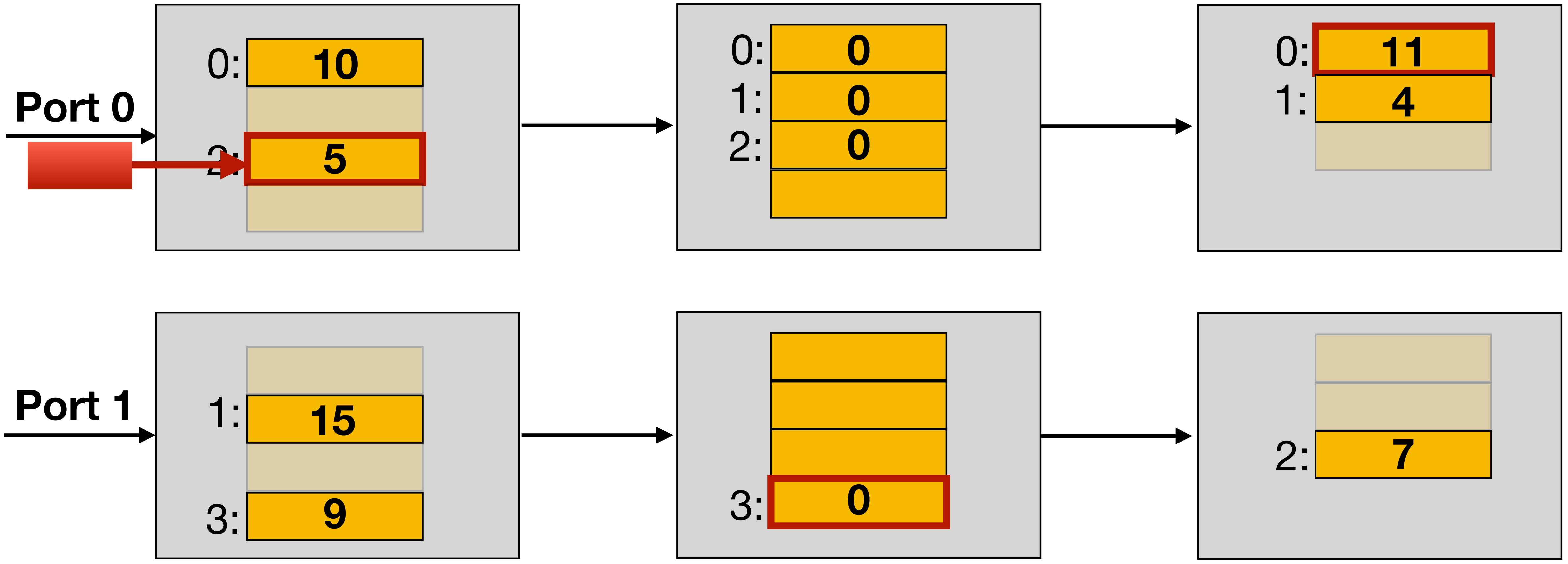
How to steer packets to a shared state in a remote pipeline?



Packet Re-circulation

Existing Solution

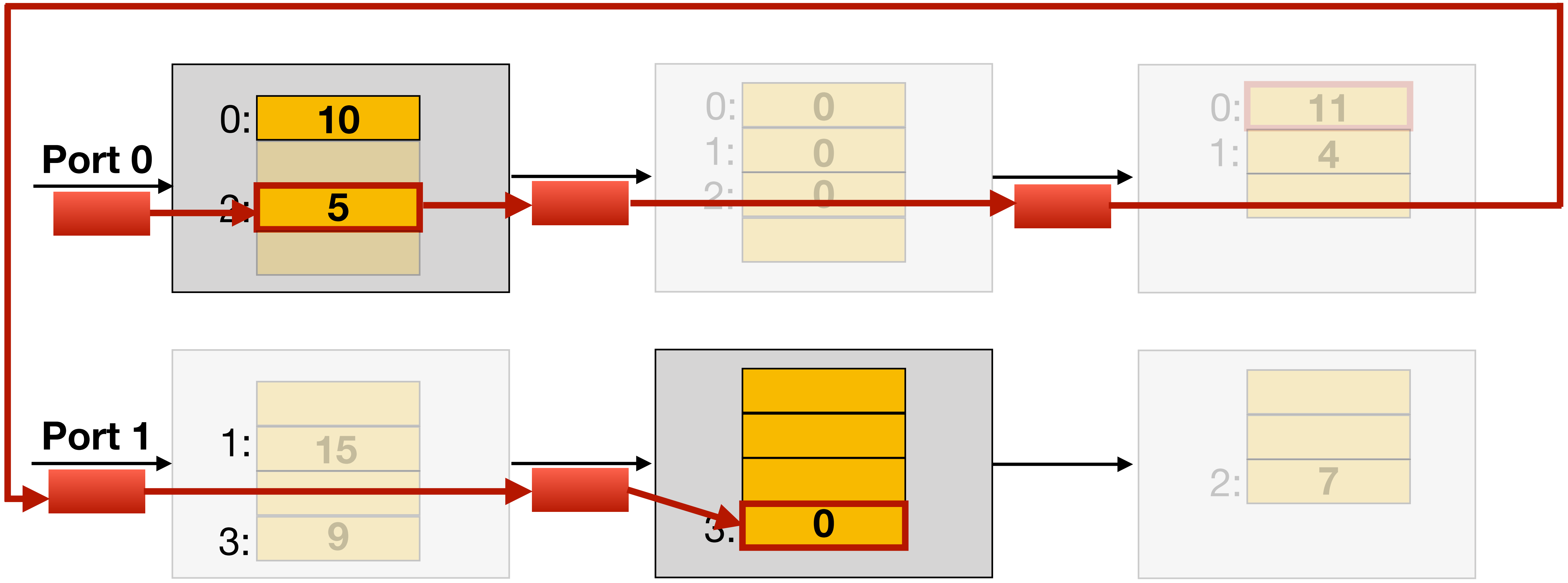
How to steer packets to a shared state in a remote pipeline?



Packet Re-circulation

Existing Solution

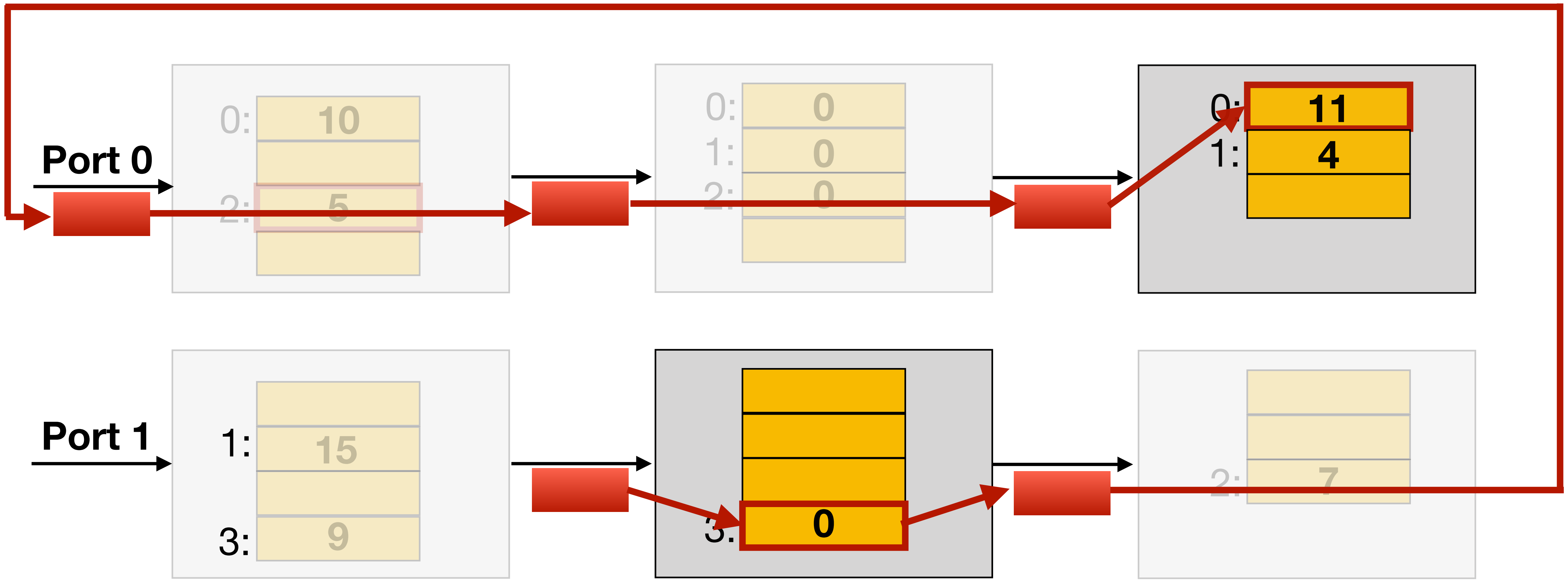
How to steer packets to a shared state in a remote pipeline?



Packet Re-circulation

Existing Solution

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



Packet Re-circulation

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!

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!

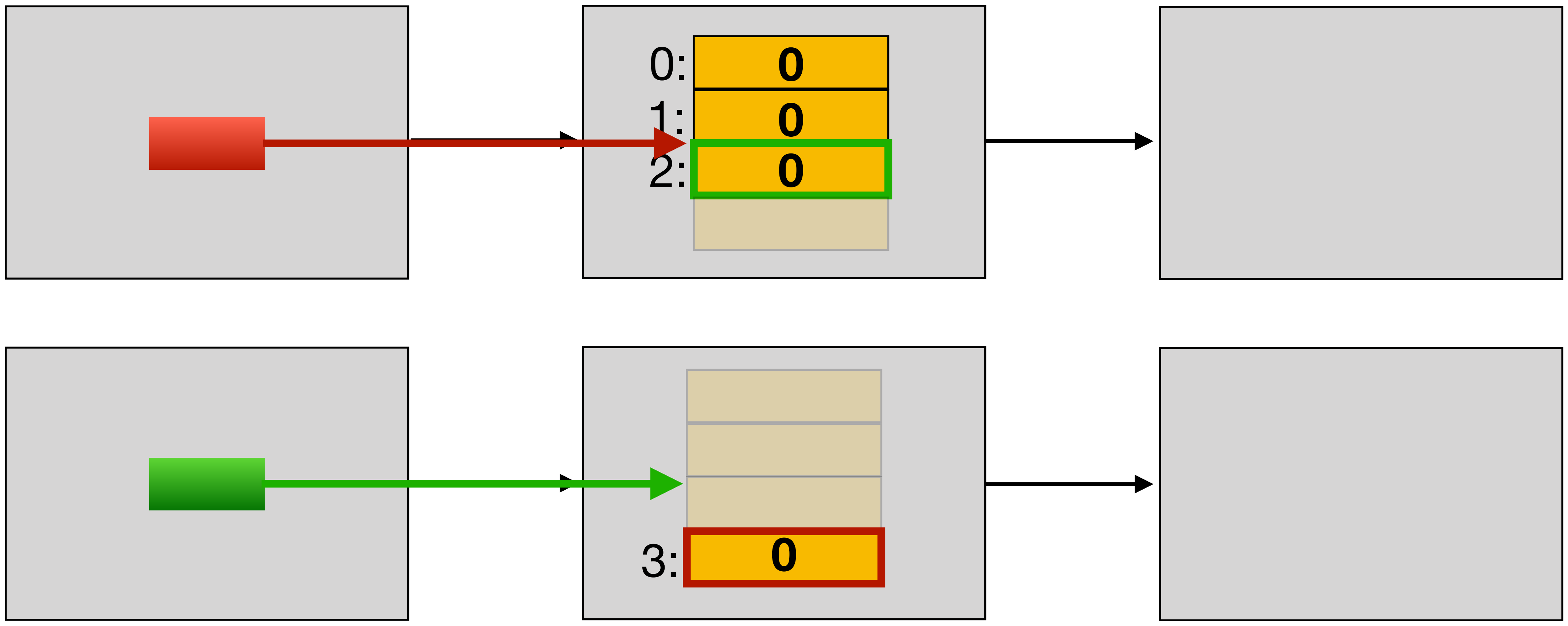
Need a **feed-forward-only** packet steering design

Existing Solution

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***

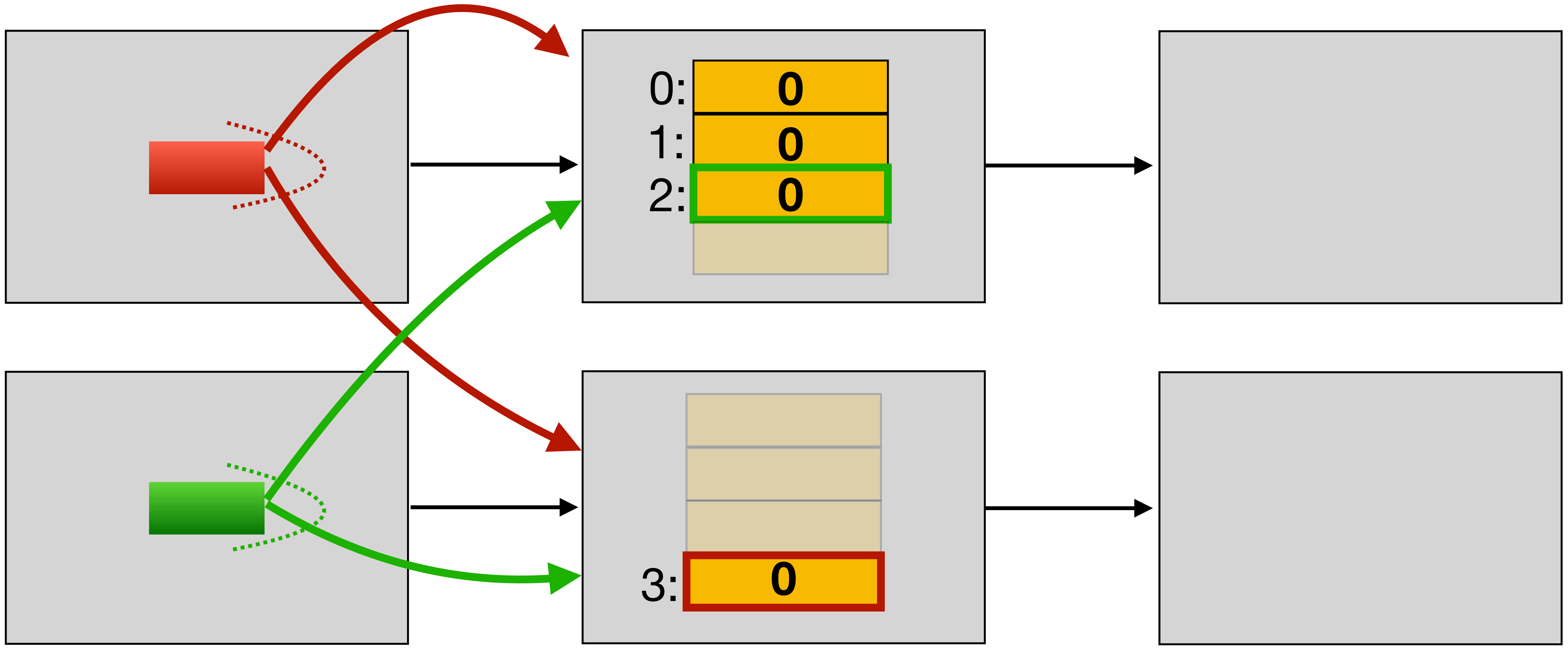


Our Solution

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

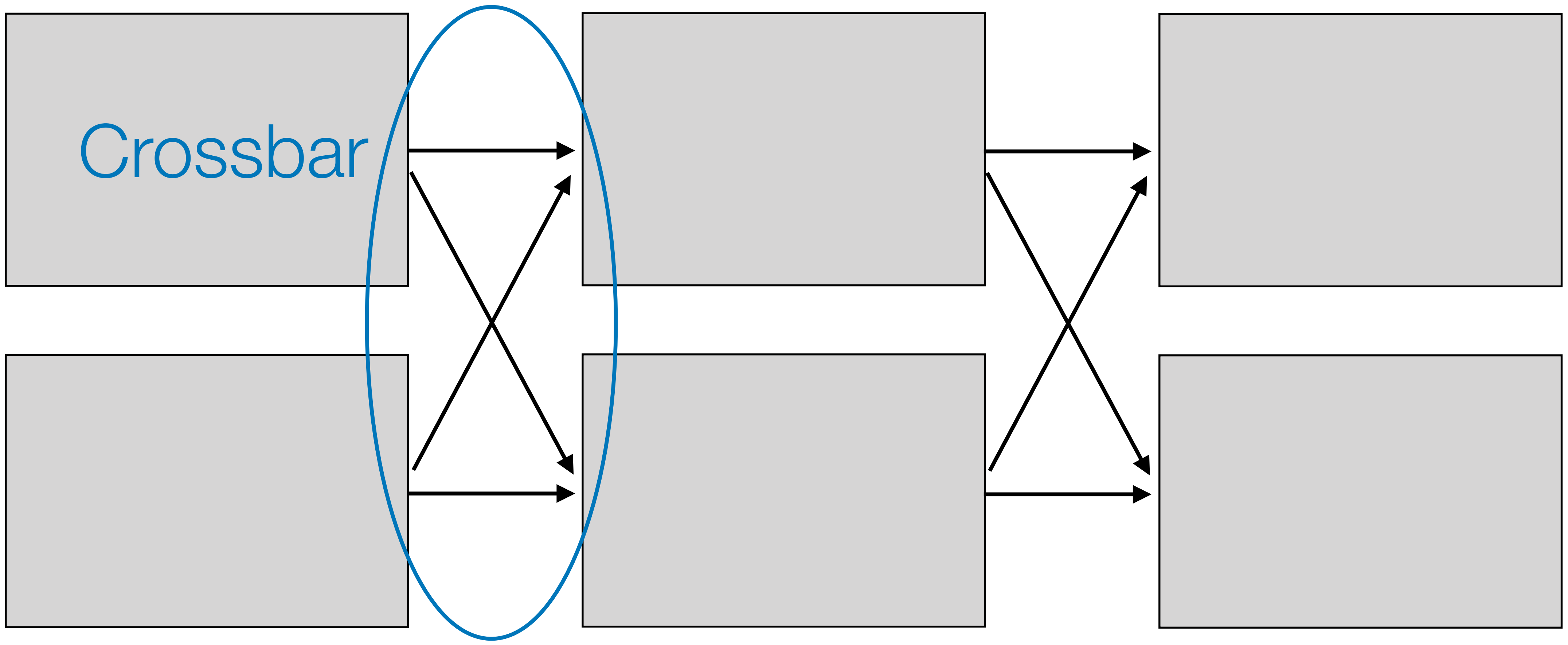


Our Solution

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

Techniques	Functional Equivalence		Performance	
	Stateless	Stateful	Stateless	Stateful
Replicate stateless processing	✓		✓	
+				
~~Limit stateful processing to single pipeline~~	~~✓~~	~~✓~~	~~✓~~	~~✗~~
+				
Dynamic state sharding & Feed-forward pkt steering	✓		✓	✓

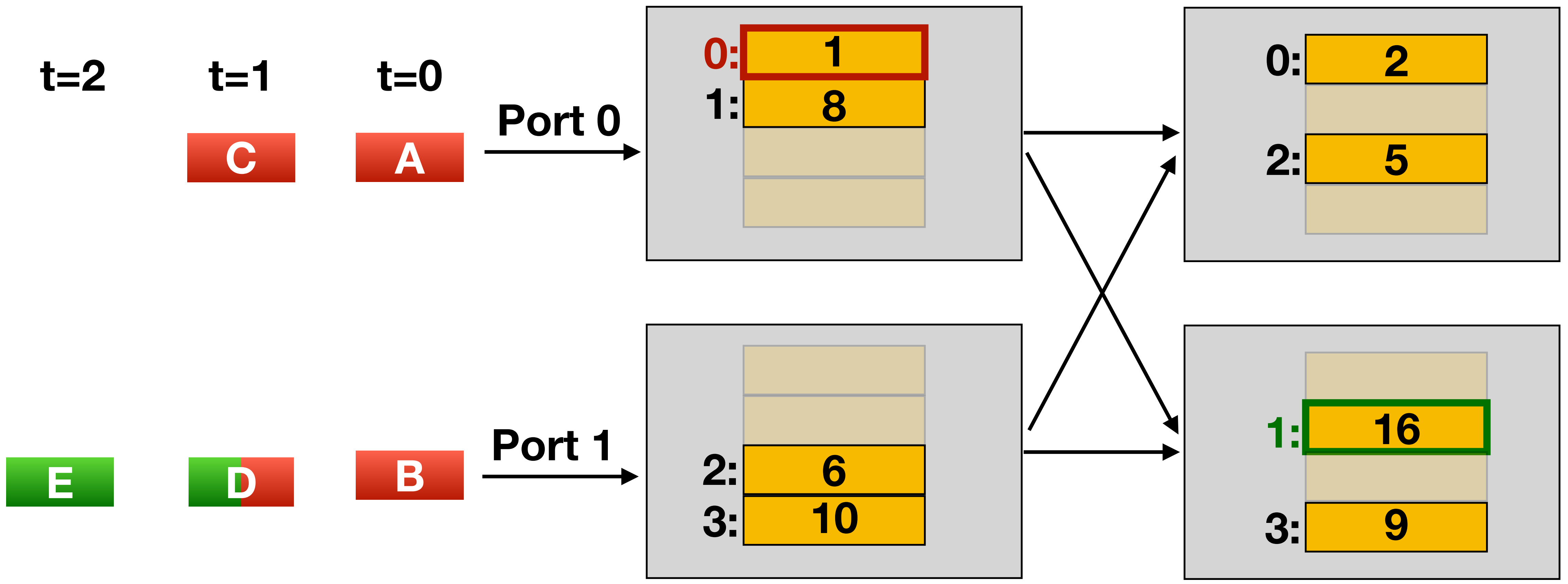
Goals and Techniques

Techniques	Functional Equivalence		Performance	
	Stateless	Stateful	Stateless	Stateful
Replicate stateless processing	✓		✓	
+				
~~**Limit stateful processing to single pipeline**~~	~~✓~~	~~✓~~	~~✓~~	~~✗~~
+				
Dynamic state sharding & Feed-forward pkt steering	✓	**?**	✓	✓

Goals and Techniques

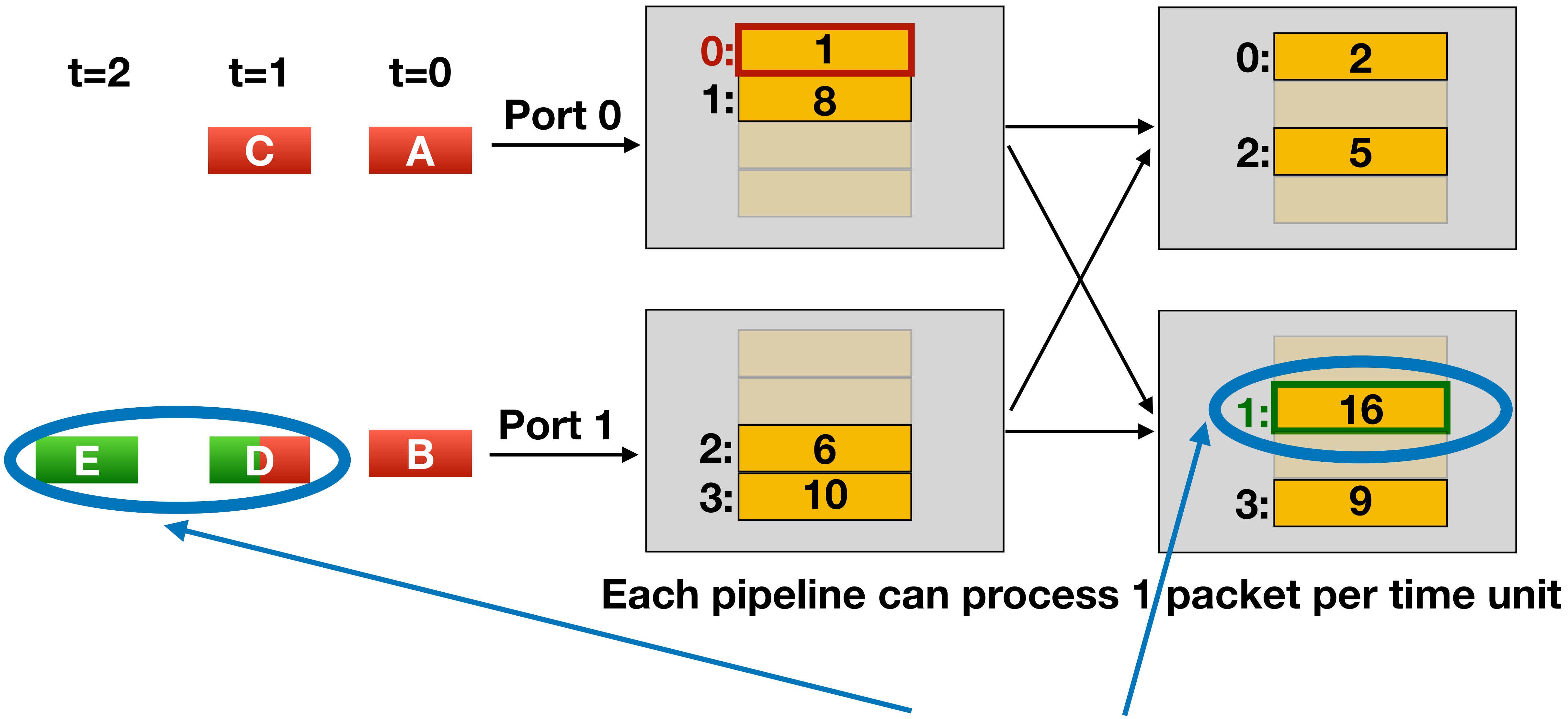
Techniques	Functional Equivalence		Performance	
	Stateless	Stateful	Stateless	Stateful
Replicate stateless processing	✓		✓	
+ ~~Limit stateful processing to single pipeline~~	~~✓~~	~~✓~~	~~✓~~	~~✗~~
+ Dynamic state sharding & Feed-forward pkt steering	✓	✗	✓	✓

Problem



Each pipeline can process 1 packet per time unit

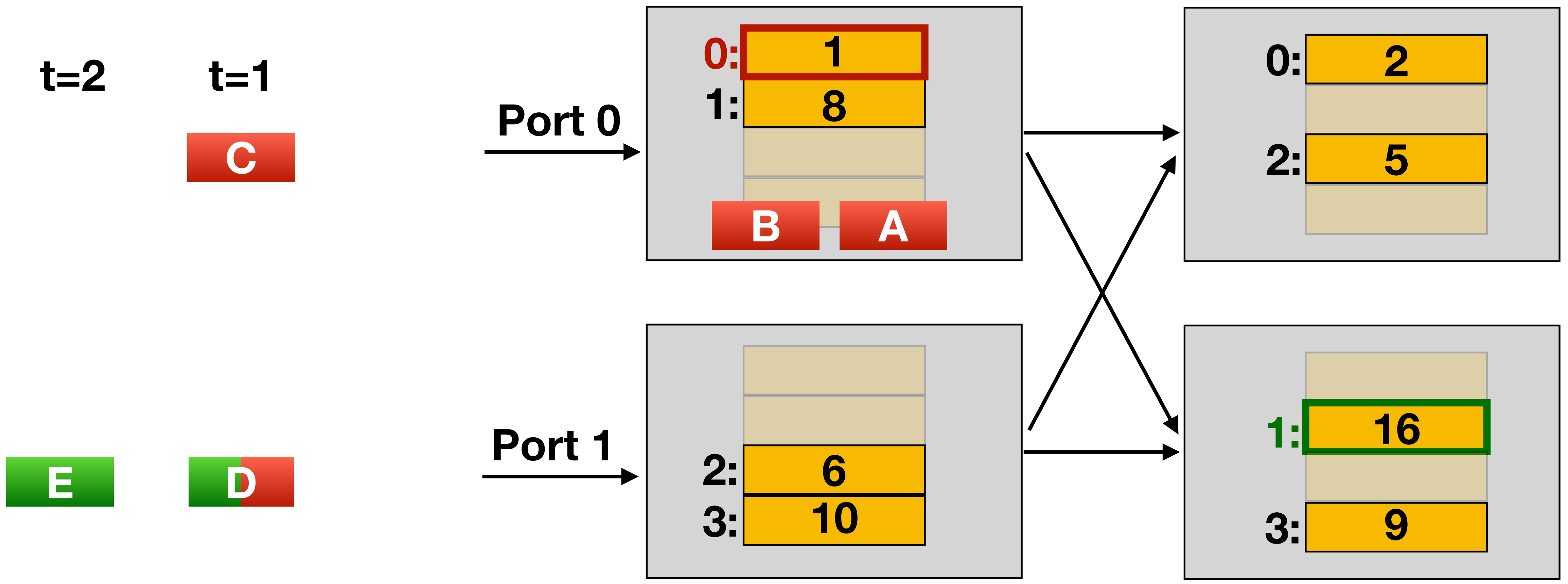
Problem



Each pipeline can process 1 packet per time unit

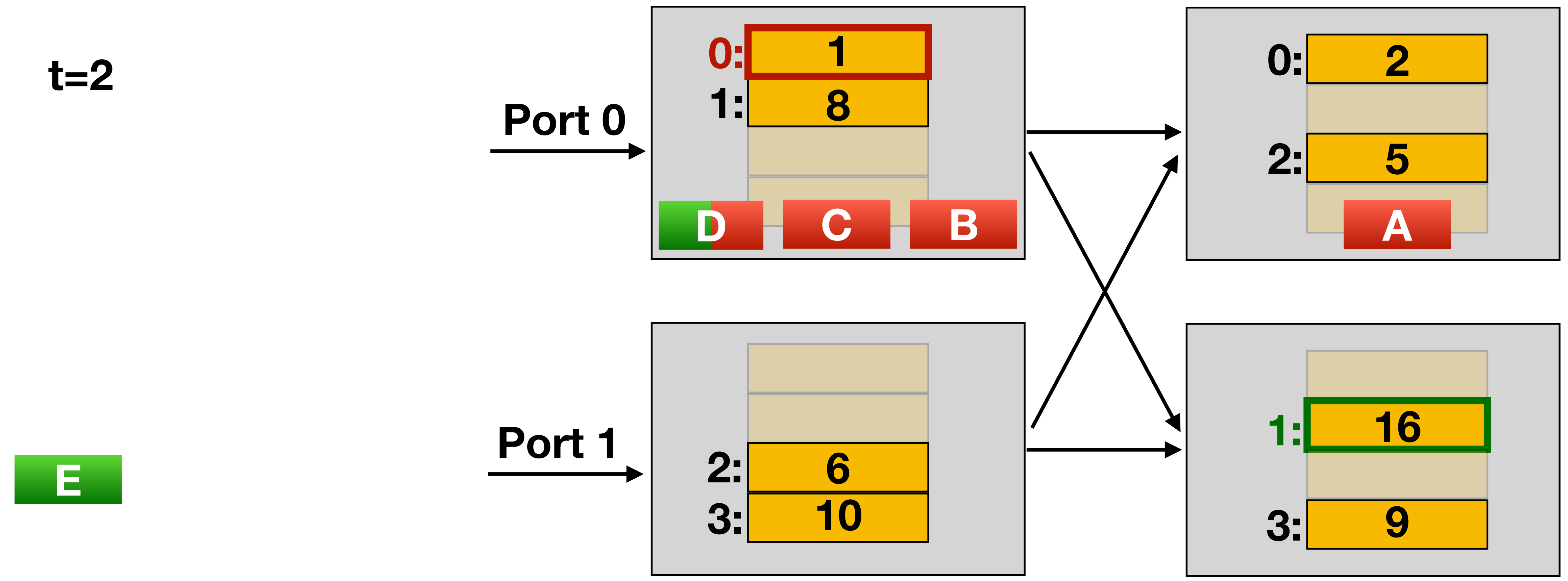
On a single-pipelined switch, D will always access register index 1 in stage 2 before E

Problem



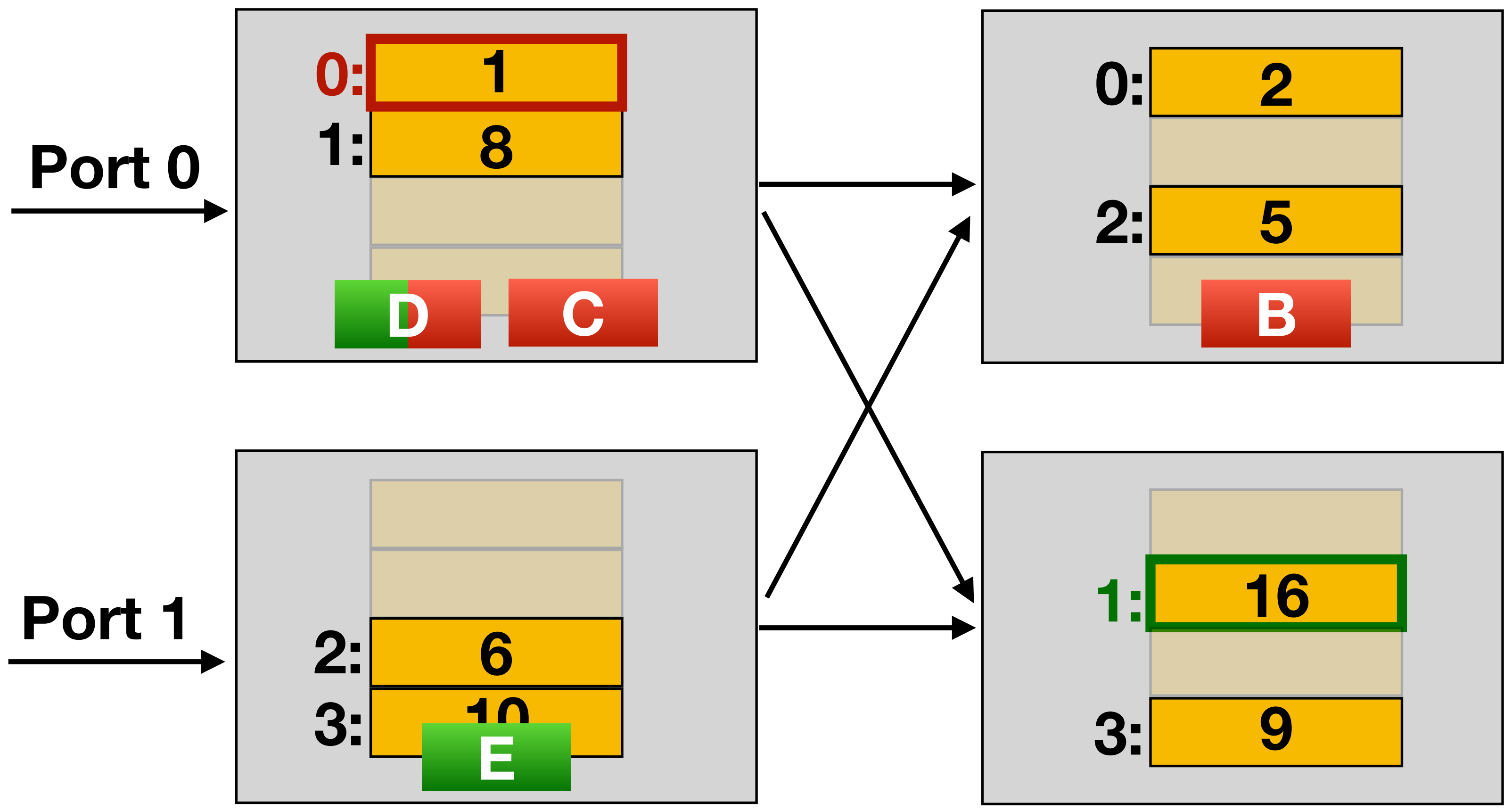
Each pipeline can process 1 packet per time unit

Problem



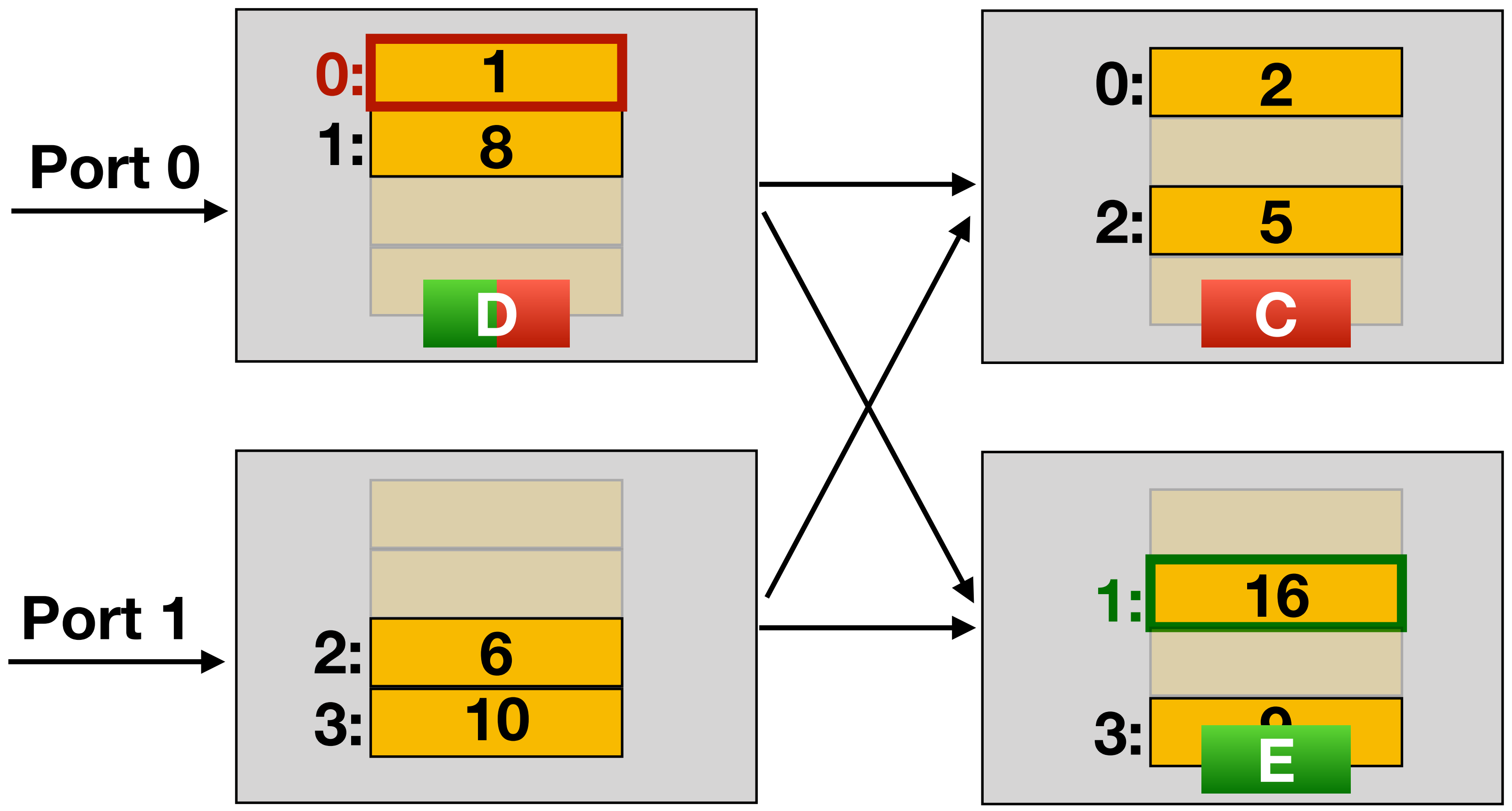
Each pipeline can process 1 packet per time unit

Problem



Each pipeline can process 1 packet per time unit

Problem

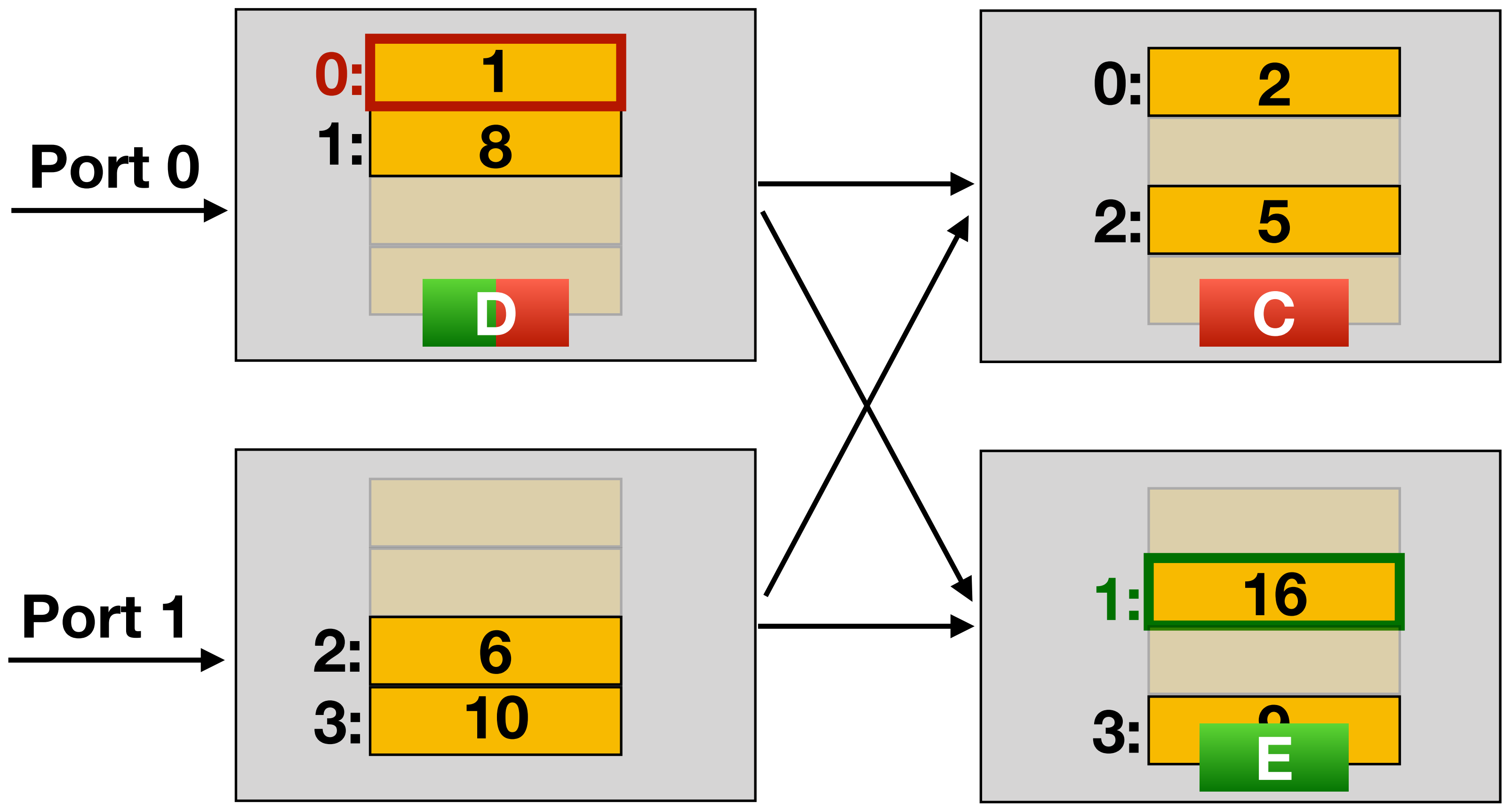


Each pipeline can process 1 packet per time unit

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

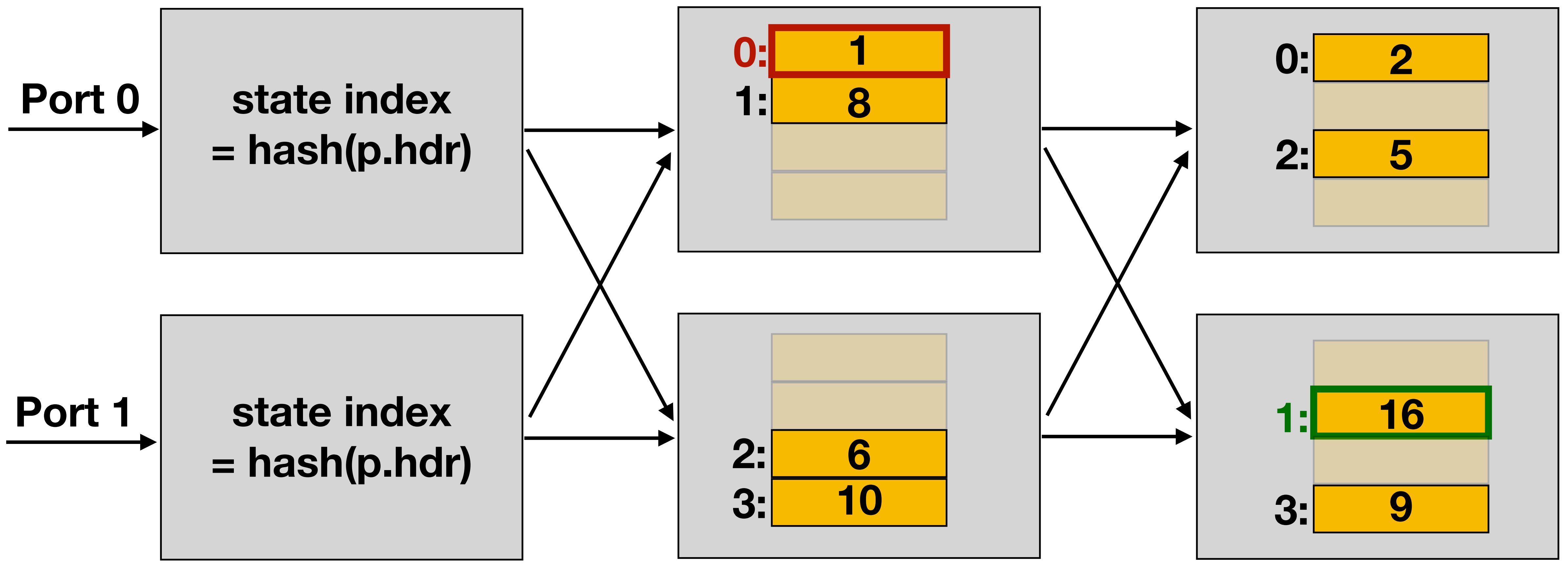
…can be known as soon as a packet arrives at the switch

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

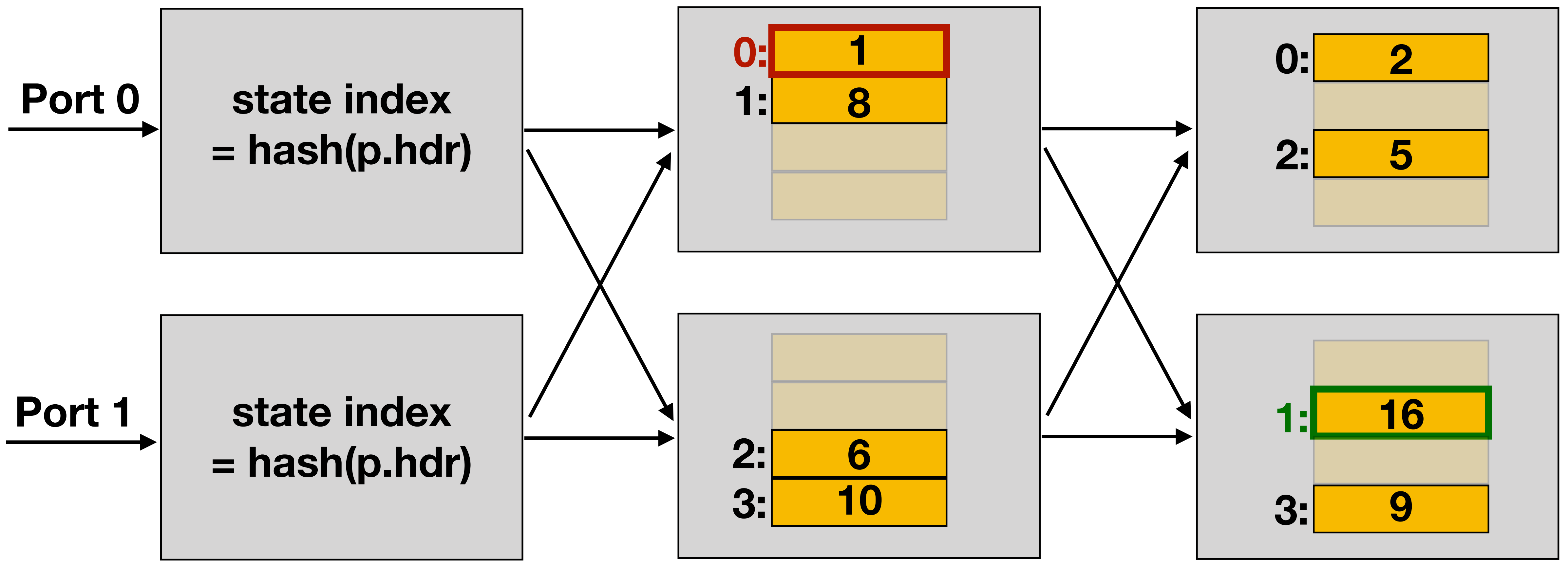


Solution

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



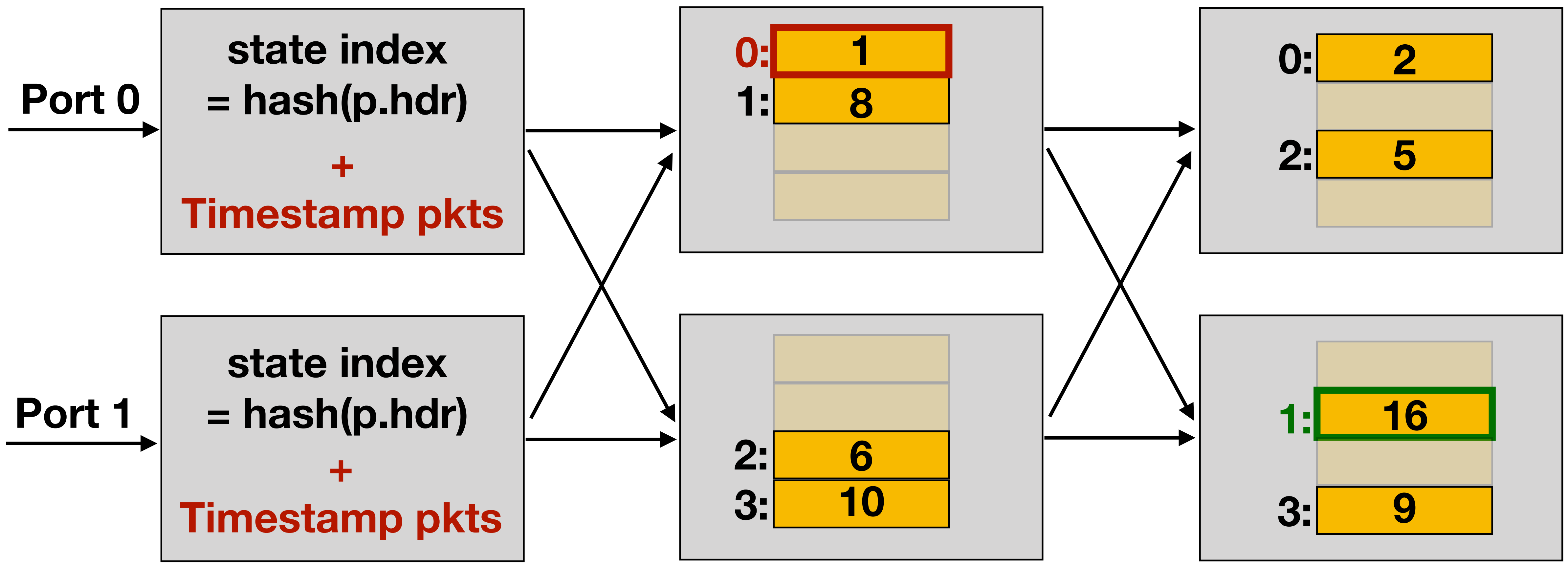
Solution

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?



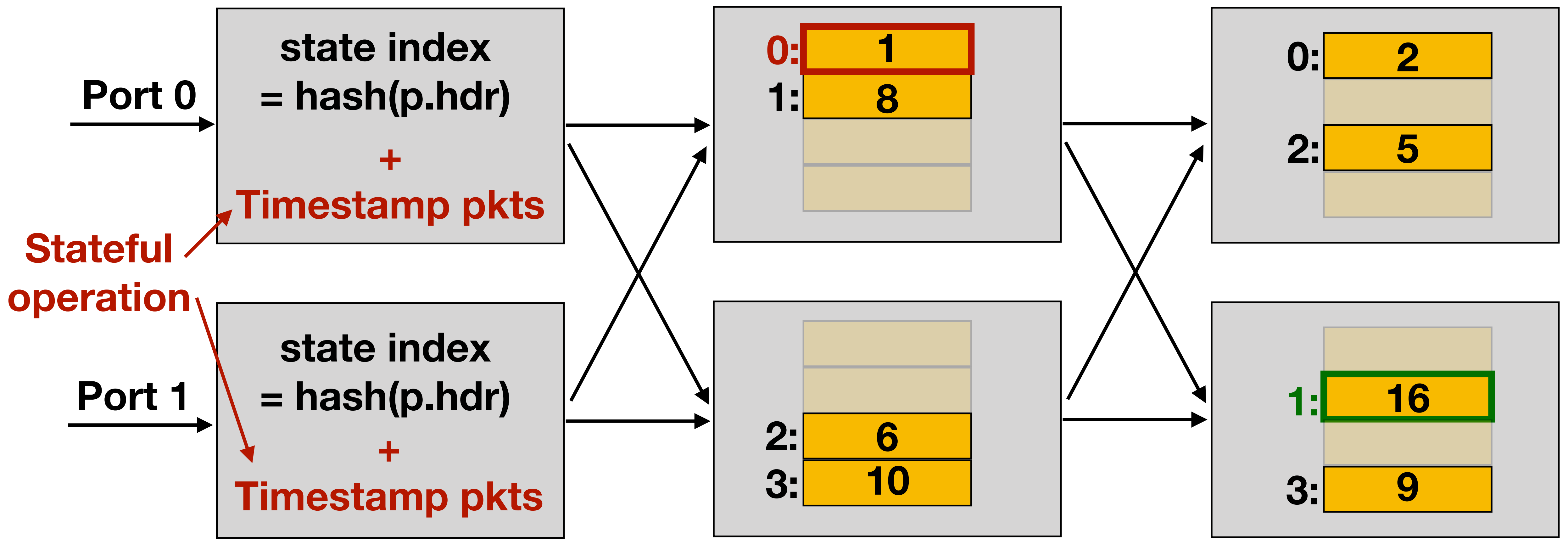
Solution

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? - won't work!



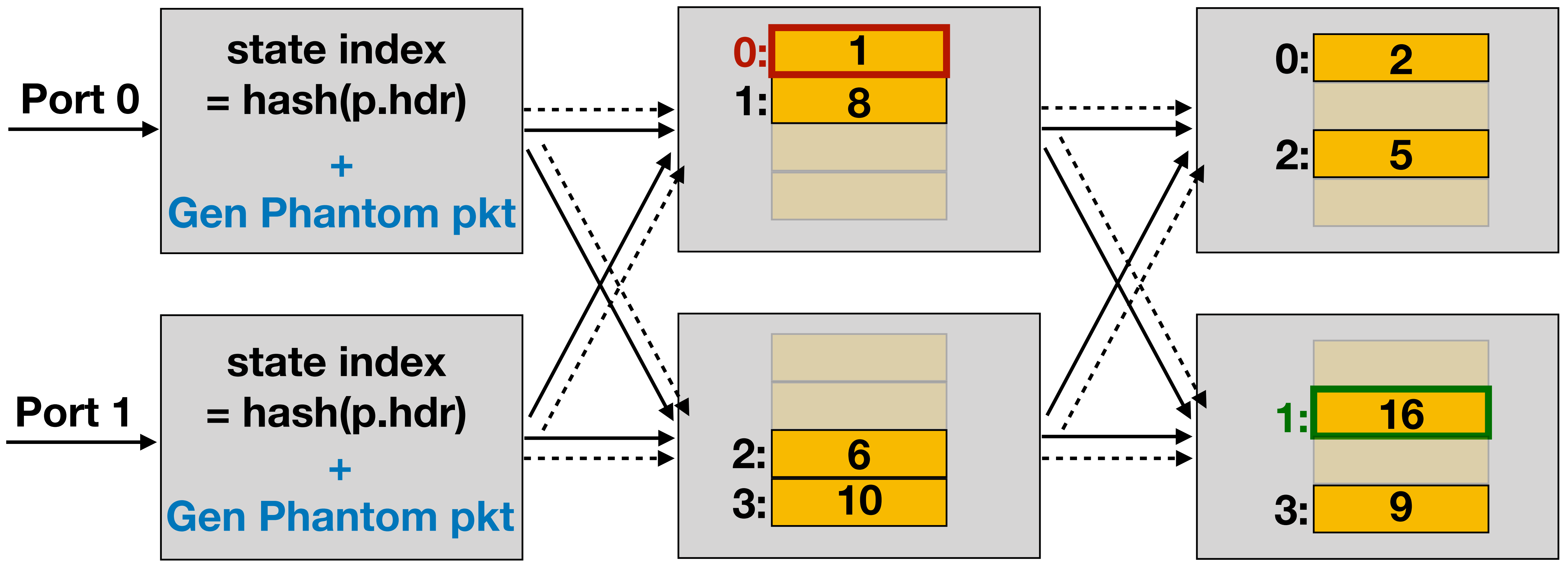
Solution

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



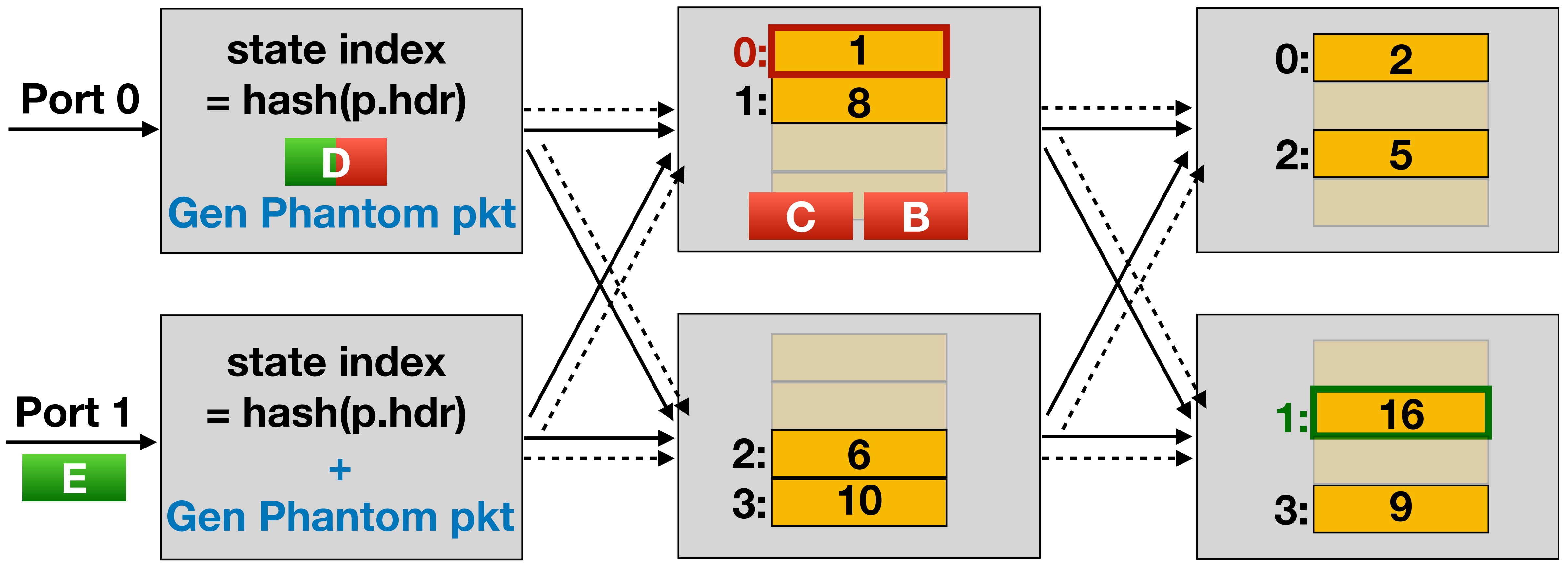
Solution

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



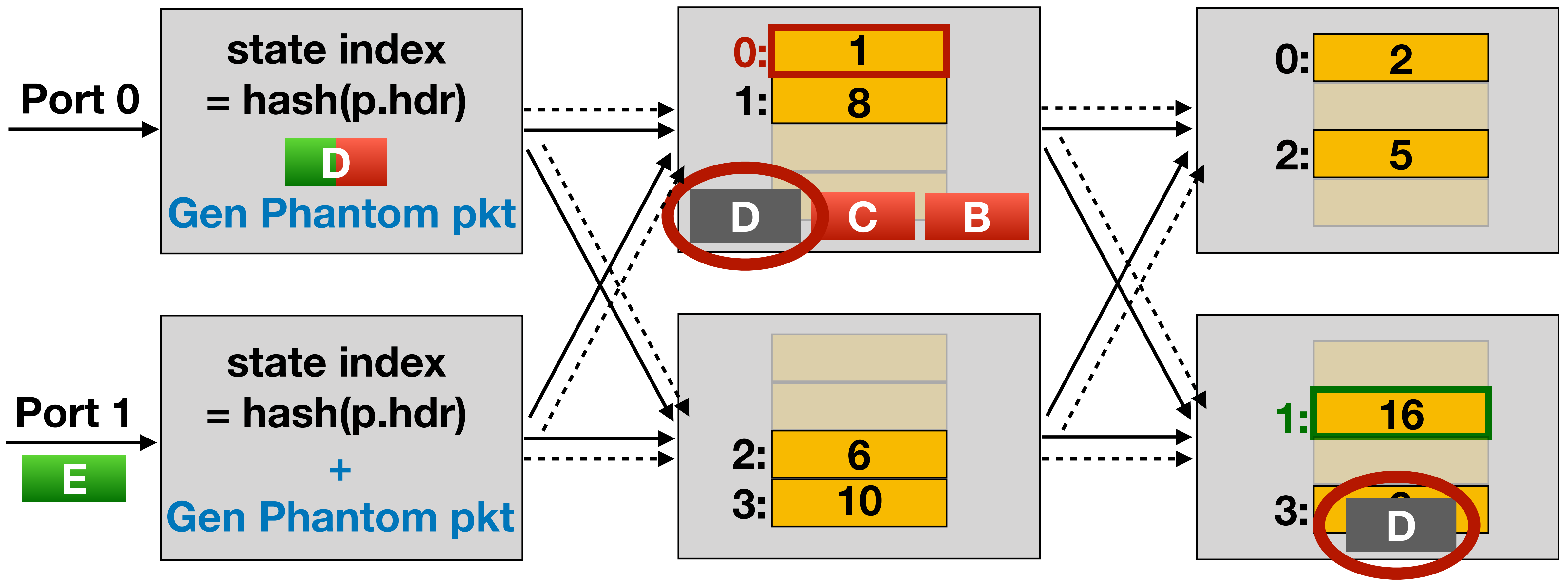
Solution

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



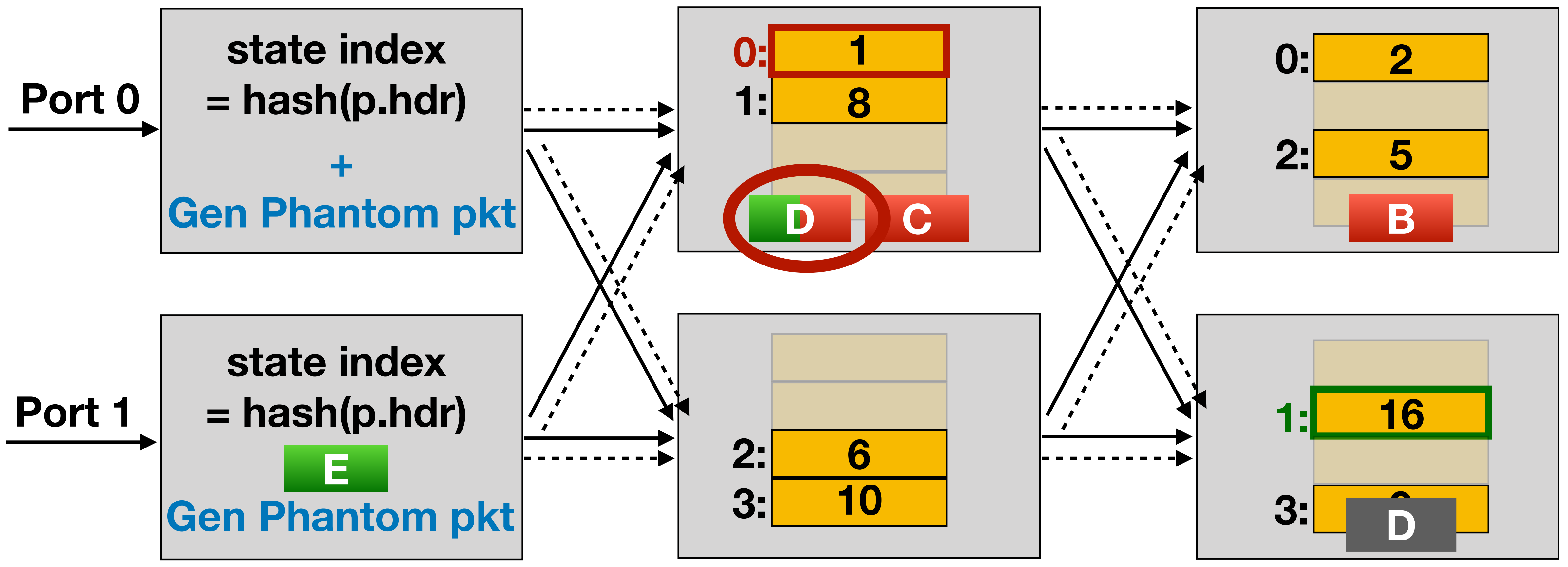
Solution

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



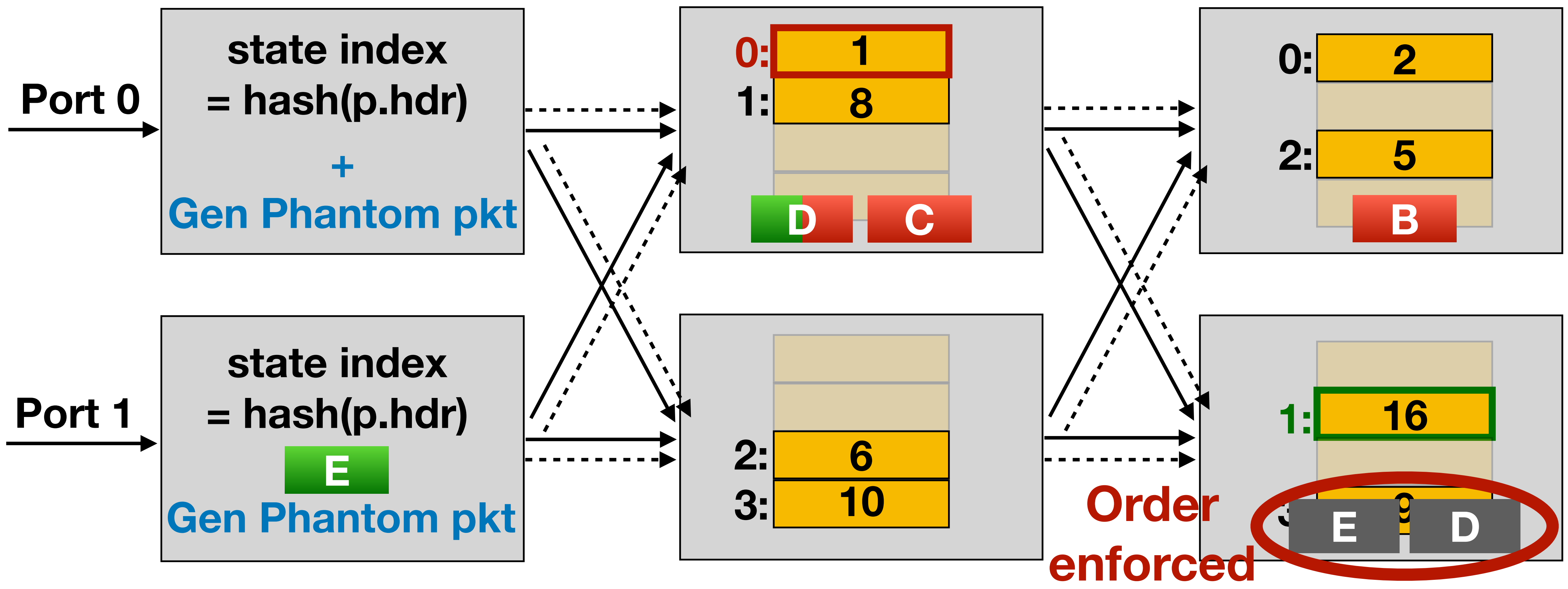
Solution

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

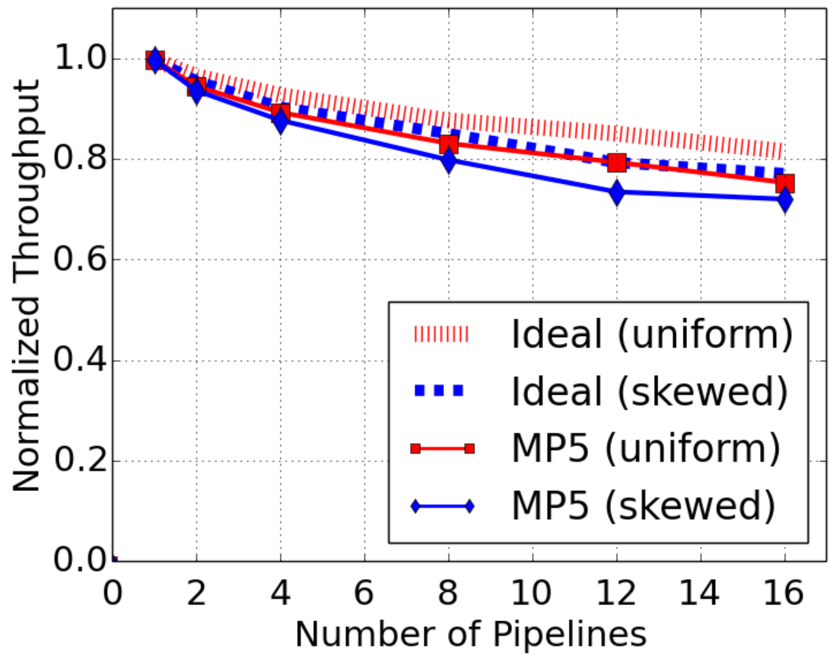


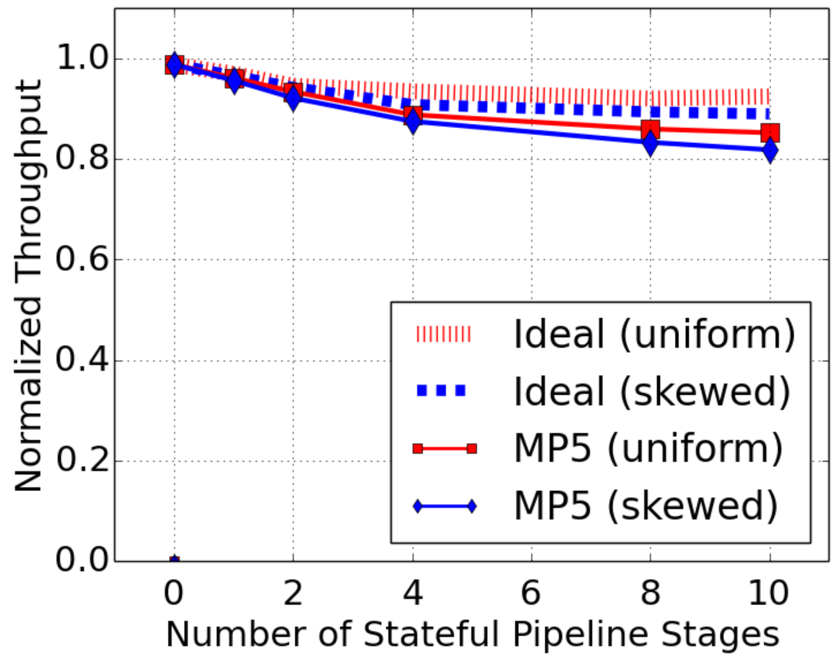
Goals and Techniques

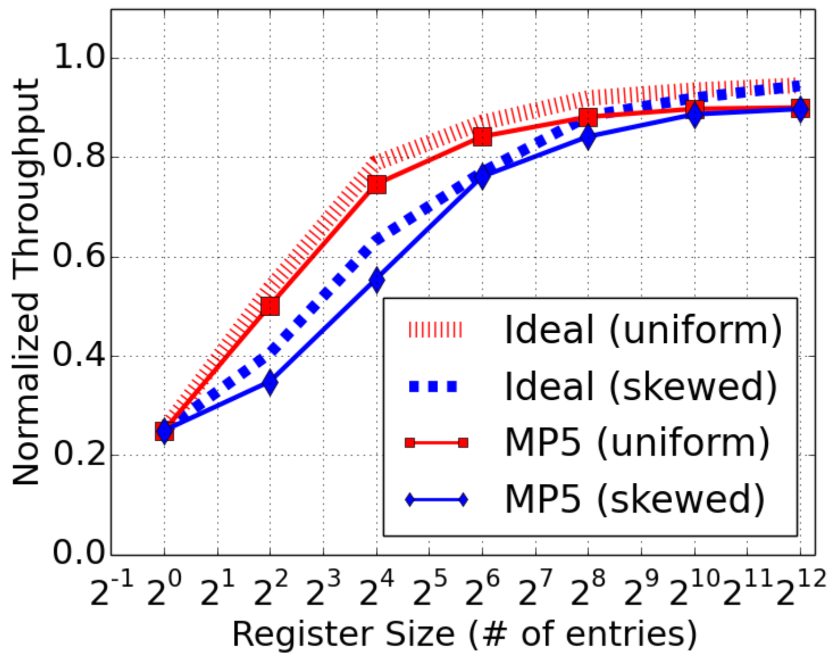
Techniques	Functional Equivalence		Performance	
	Stateless	Stateful	Stateless	Stateful
Replicate stateless processing	✓		✓	
+				
~~Limit stateful processing to single pipeline~~	~~✓~~	~~✓~~	~~✓~~	~~✗~~
+				
Dynamic state sharding & Feed-forward pkt steering	✓	✗	✓	✓
+				
Preemptive state access order enforcement	✓	✓	✓	✓

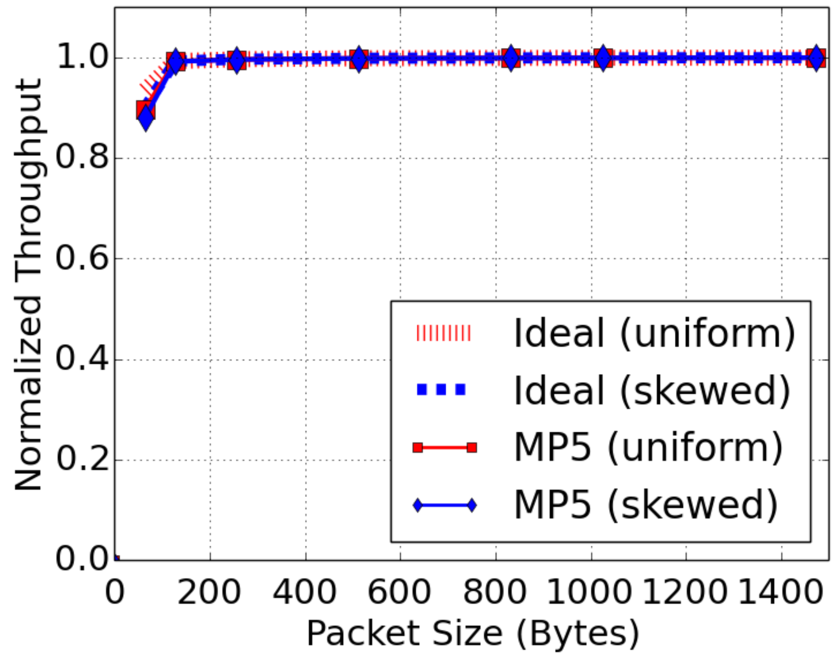
Performance Evaluation

Sensitivity Analysis

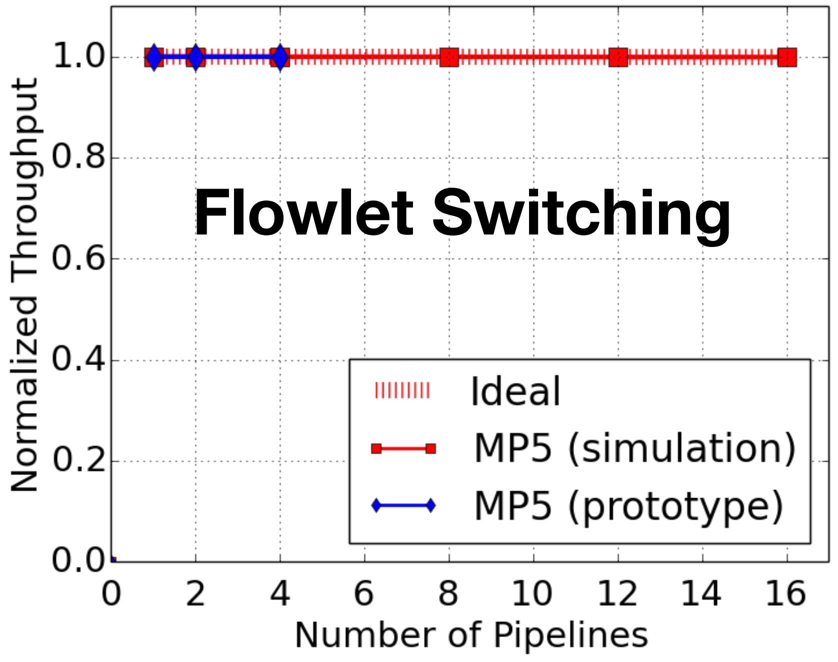


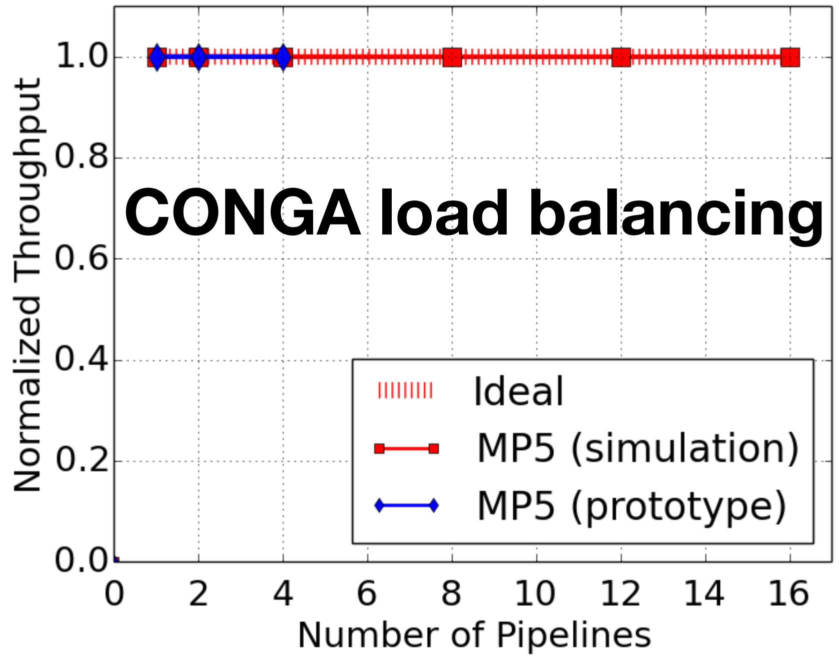


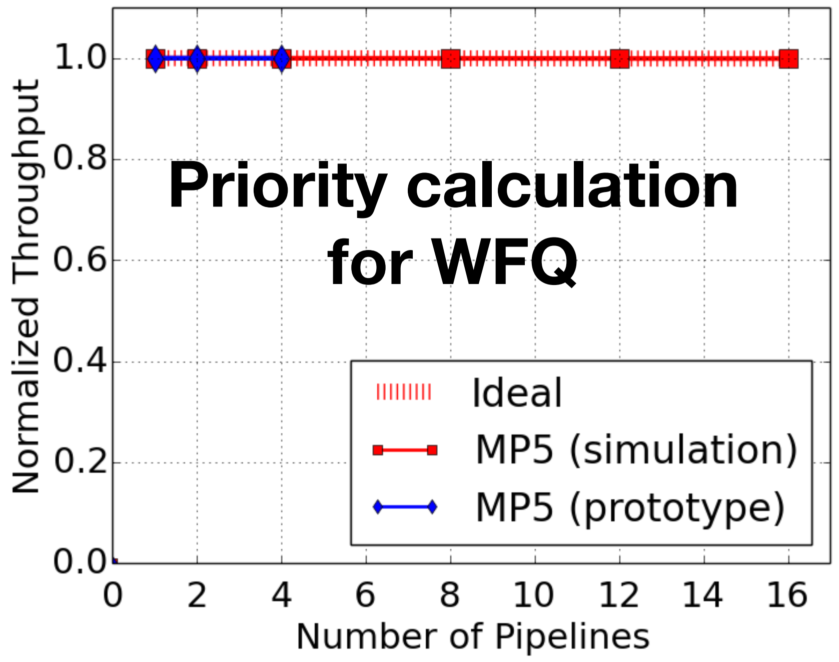


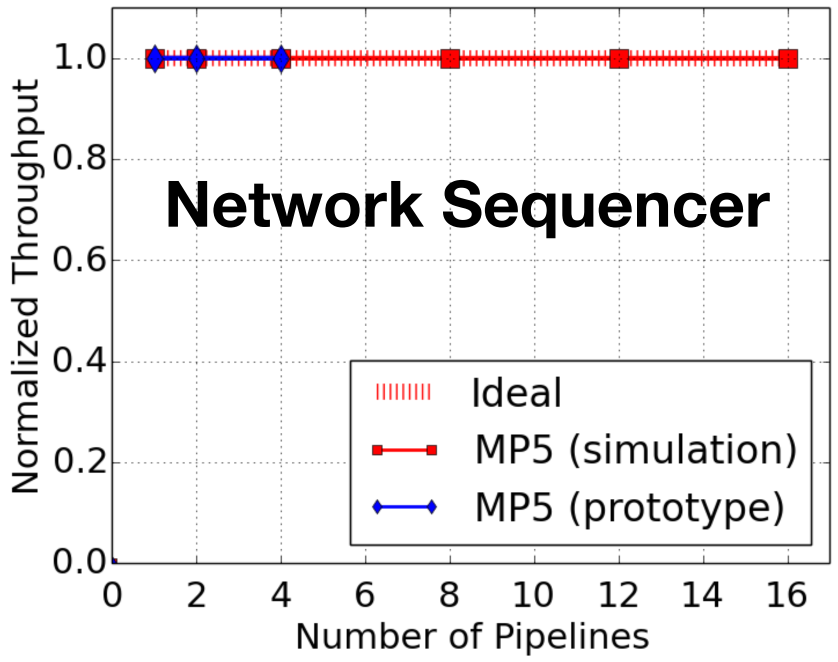


Realistic Workloads & Applications

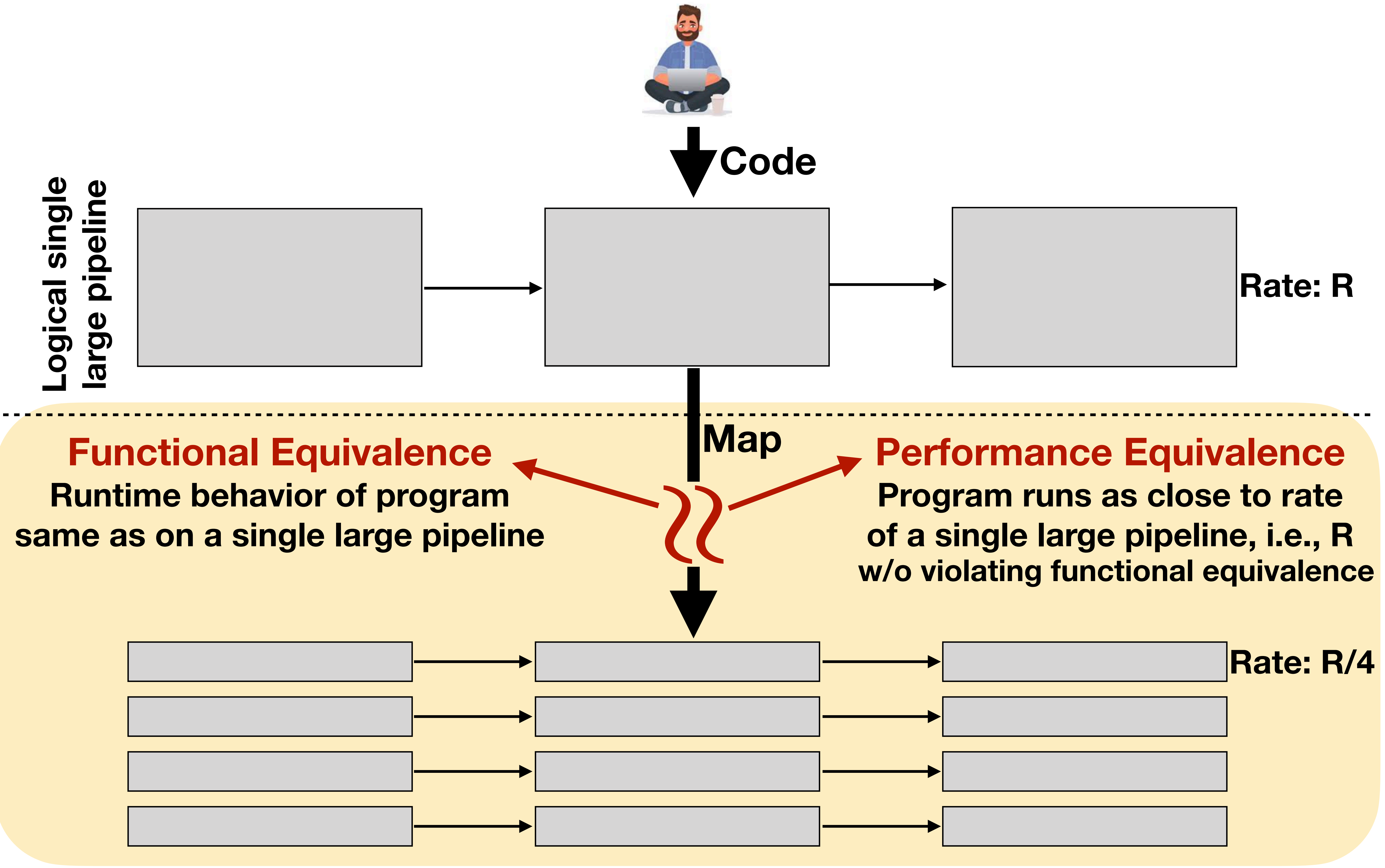






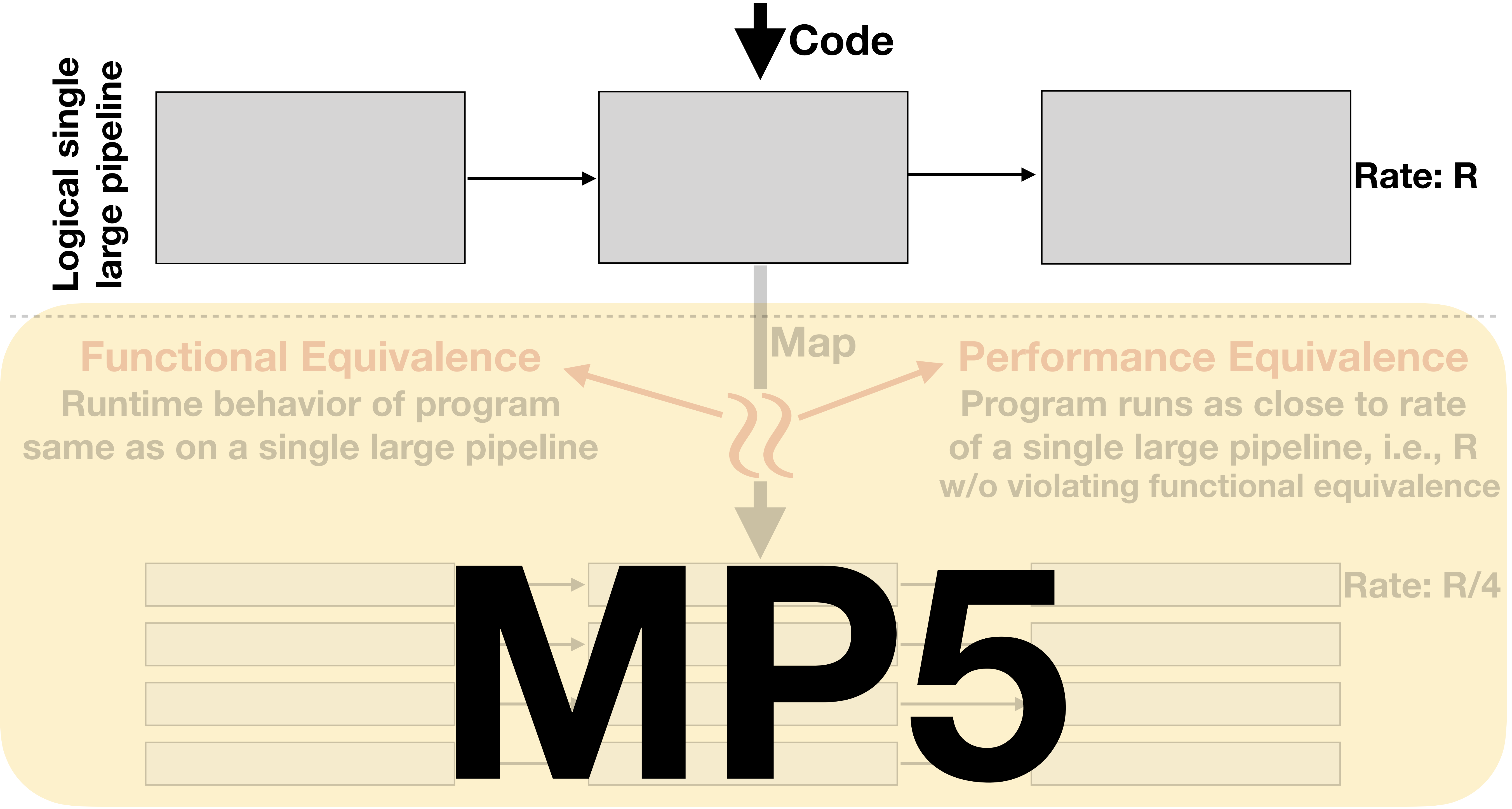


Summary



Code
Logical single large pipeline
Rate: R
Map
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

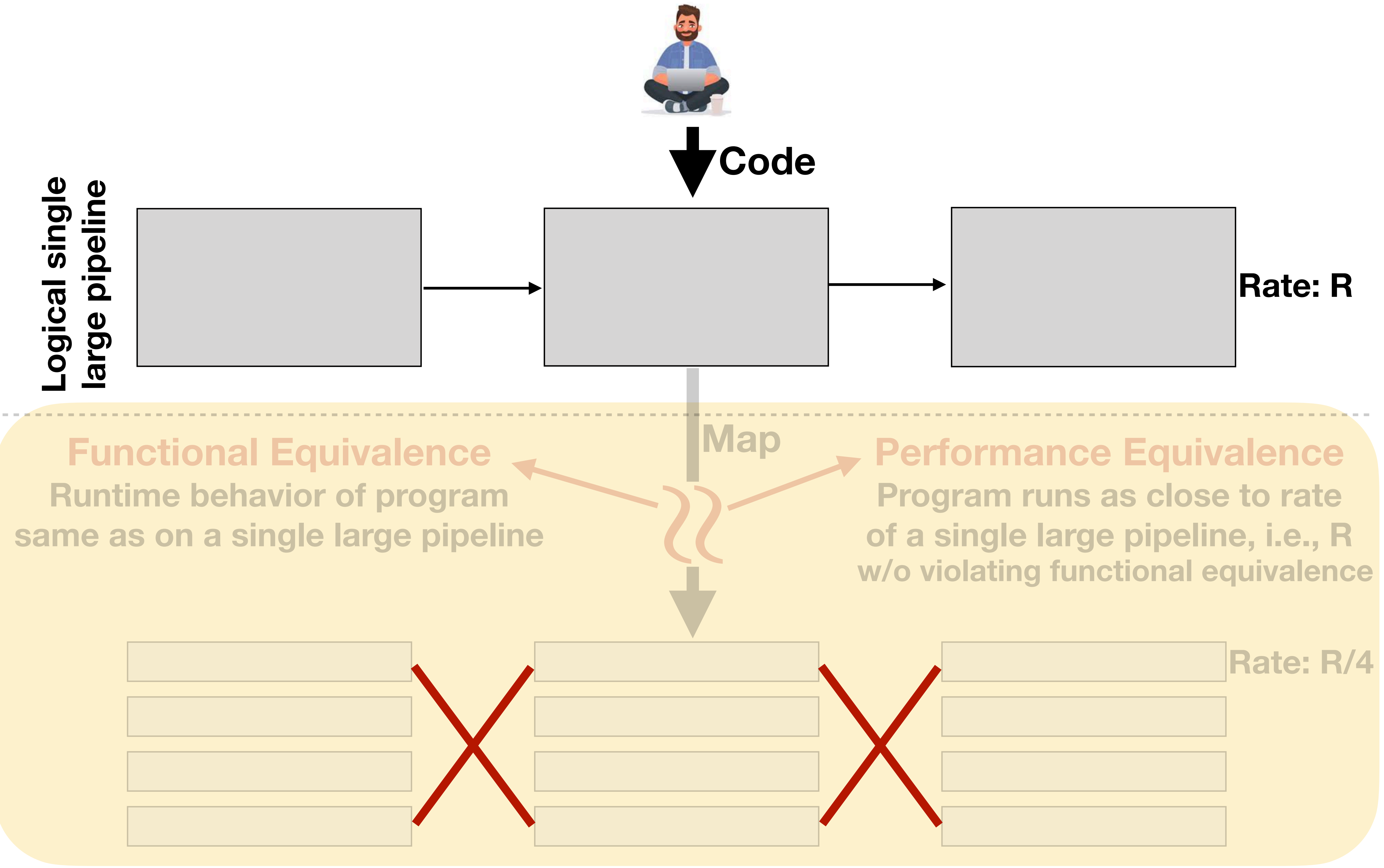
Summary



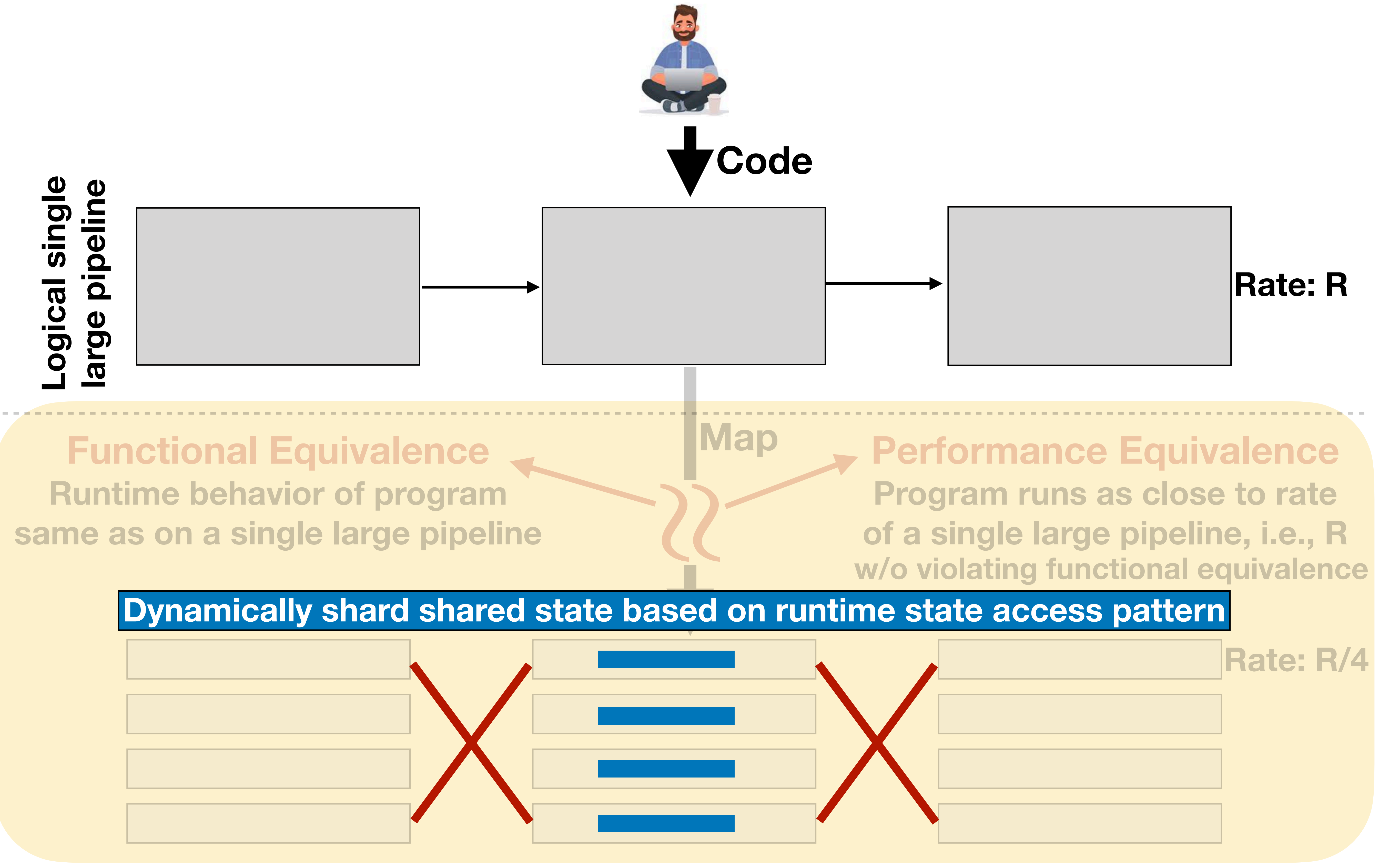
Code
Logical single large pipeline
Rate: R
Map
Functional Equivalence
Runtime behavior of program same as on a single large pipeline
Performance Equivalence
Program runs as close to rate of a single large pipeline, i.e., R w/o violating functional equivalence
Rate: R/4

MP5

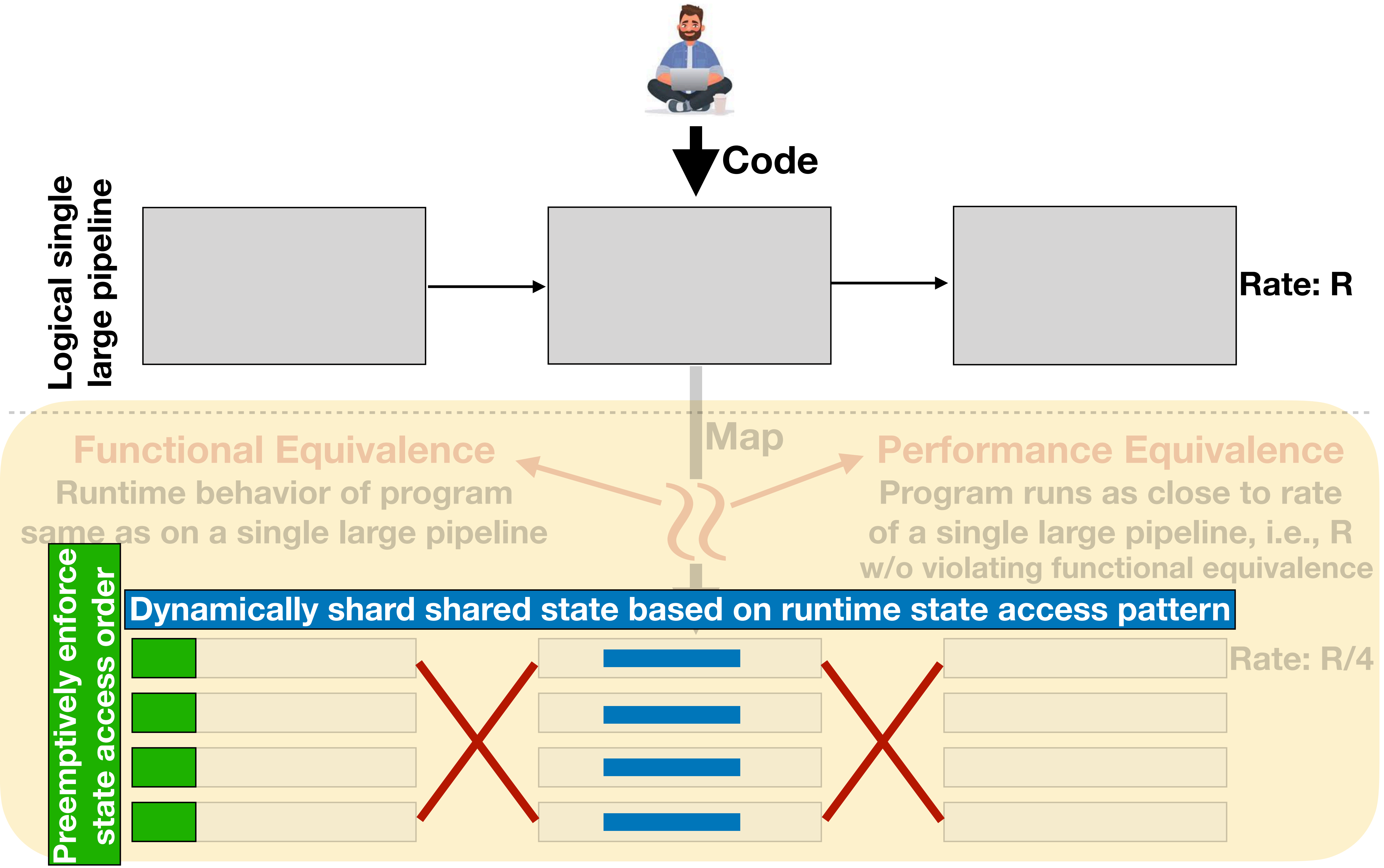
Summary


Code
Logical single large pipeline
Rate: R
Map
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

Summary


Code
Logical single large pipeline
Rate: R
Map
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
Rate: R/4

Summary


Code
Logical single large pipeline
Rate: R
Map
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
Rate: R/4

Thank you!