State-Compute Replication: Parallelizing High-Speed Stateful Packet Processing

Academic Salon on High-Performance Ethernet: Host Networking and Monitoring

12 - 13 March 2025

POLITECNICO
MILANO 1863

RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

NEW YORK UNIVERSITY

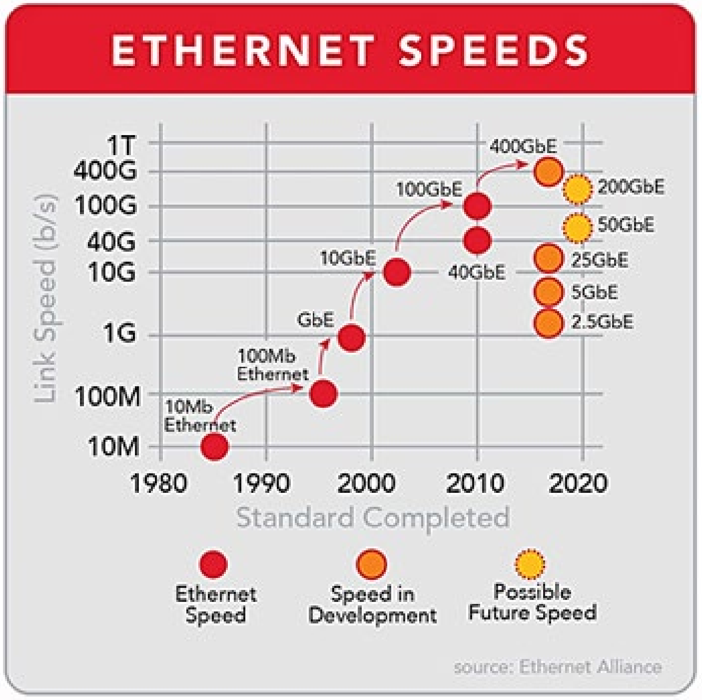
Software Packet Processing in post Moore's Law Era

- Throughput is a **first-class citizen** in modern networked systems
  - Software LBs, CDN nodes, DDoS mitigators depends on it

Software Packet Processing in post Moore's Law Era

- Throughput is a **first-class citizen** in modern networked systems
  - Software LBs, CDN nodes, DDoS mitigators depends on it

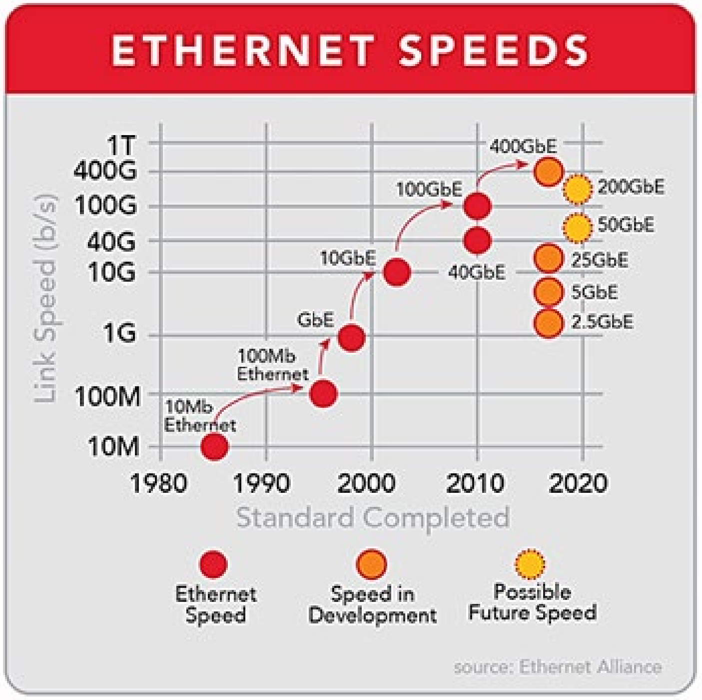
Increasing NICs speed



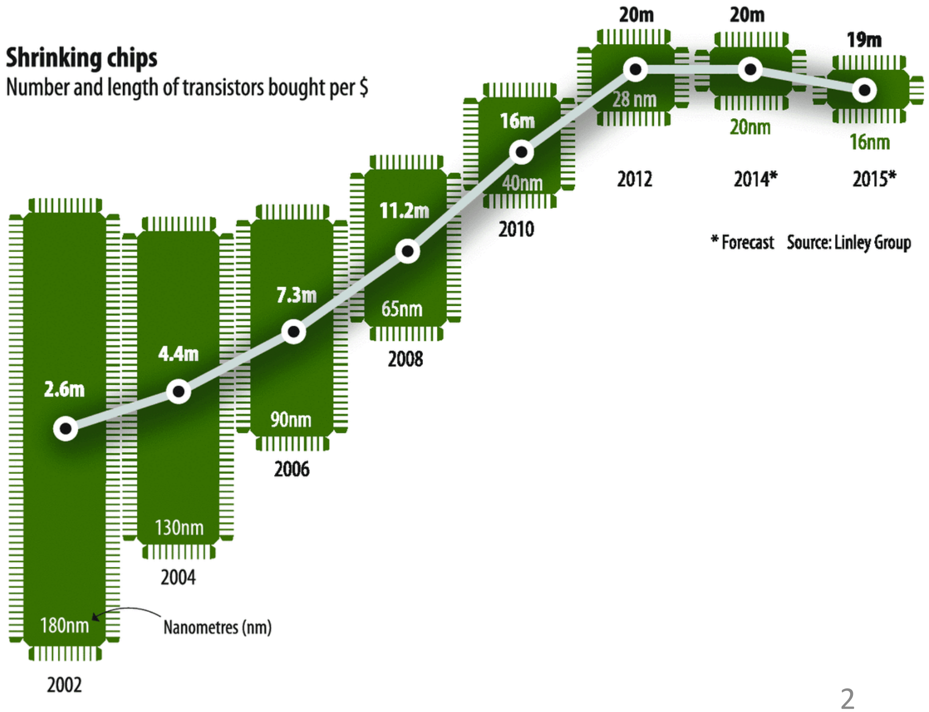
Software Packet Processing in post Moore's Law Era

- Throughput is a **first-class citizen** in modern networked systems
  - Software LBs, CDN nodes, DDoS mitigators depends on it

Increasing NICs speed



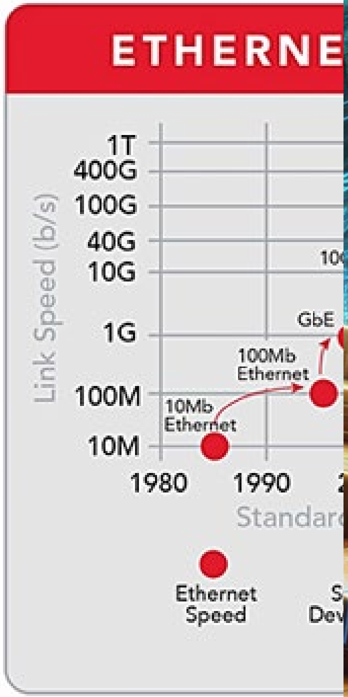
Slowdown of Moore's Law

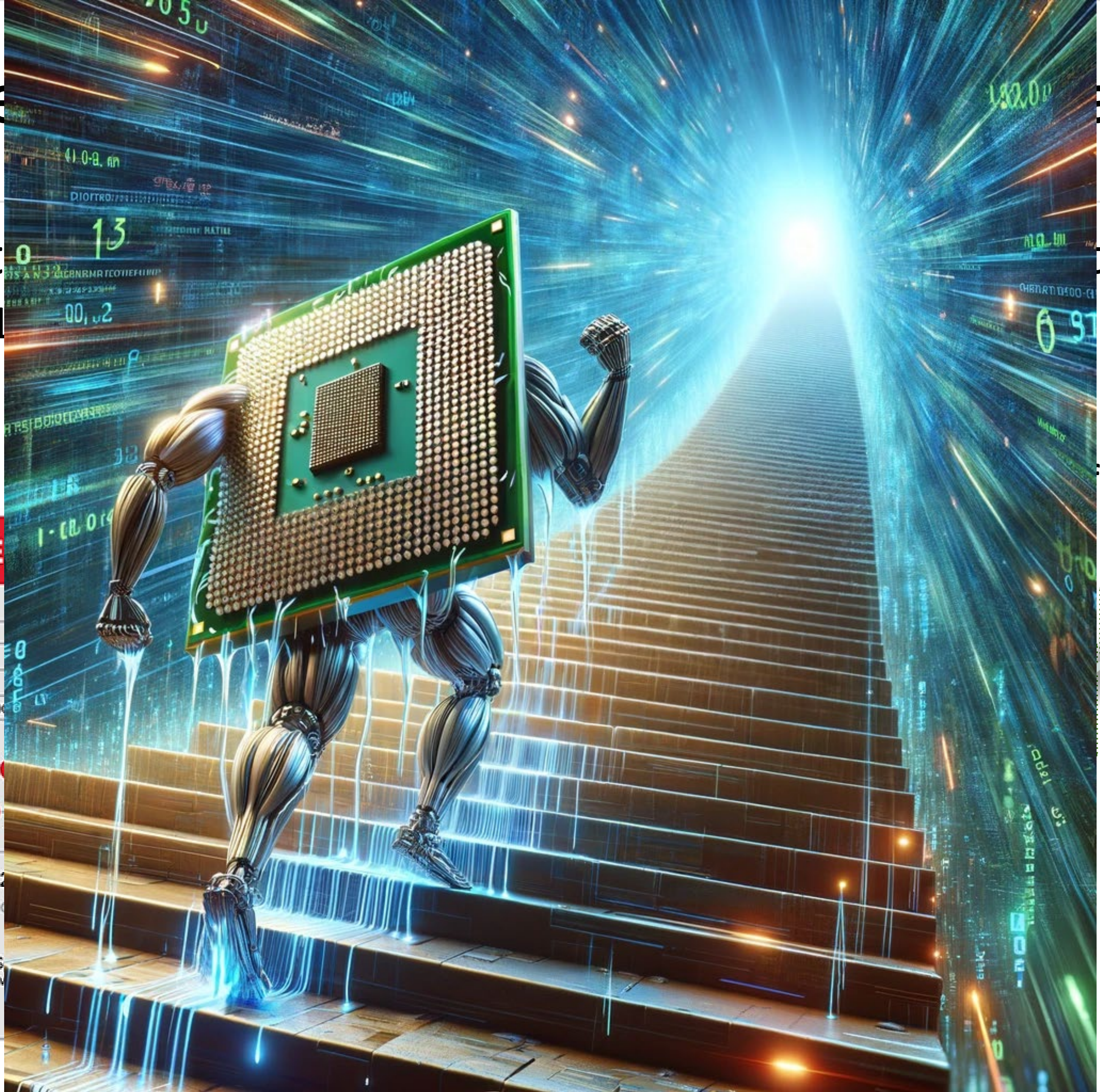


Software aw Era

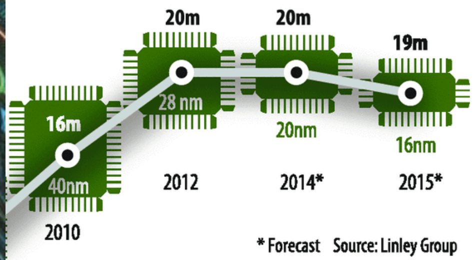
- Throughput tems
  - Software

Increasing





Moore's Law



*a CPU core that is struggling to cope with the increasing speed of networking

How to speed up packet processing?

- There have been significant efforts to speed up packet processing through
  - Better stack design

Rethinking Network Stack Design with Memory Snapshots

Michael Chan   Heiner Litz   David R. Cheriton
Department of Computer Science
Stanford University
{mcfchan , hlitz , cheriton}@stanford.edu

Network Stack Specialization for Performance

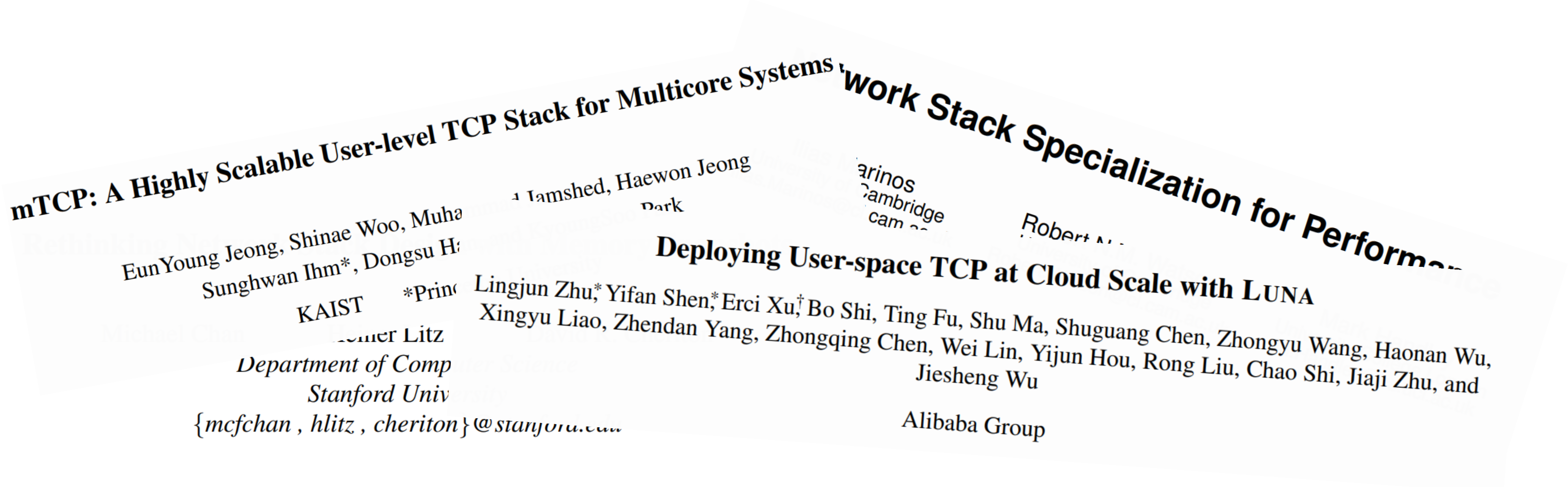
Ilias Marinos
University of Cambridge
Ilias.Marinos@cl.cam.ac.uk

Robert N.M. Watson
University of Cambridge
Robert.Watson@cl.cam.ac.uk

Mark Handley
University College London
M.Handley@cs.ucl.ac.uk

How to speed up software packet processing?

- There have been significant efforts to speed up packet processing through
  - Better stack design
  - Removing user-kernel crossings

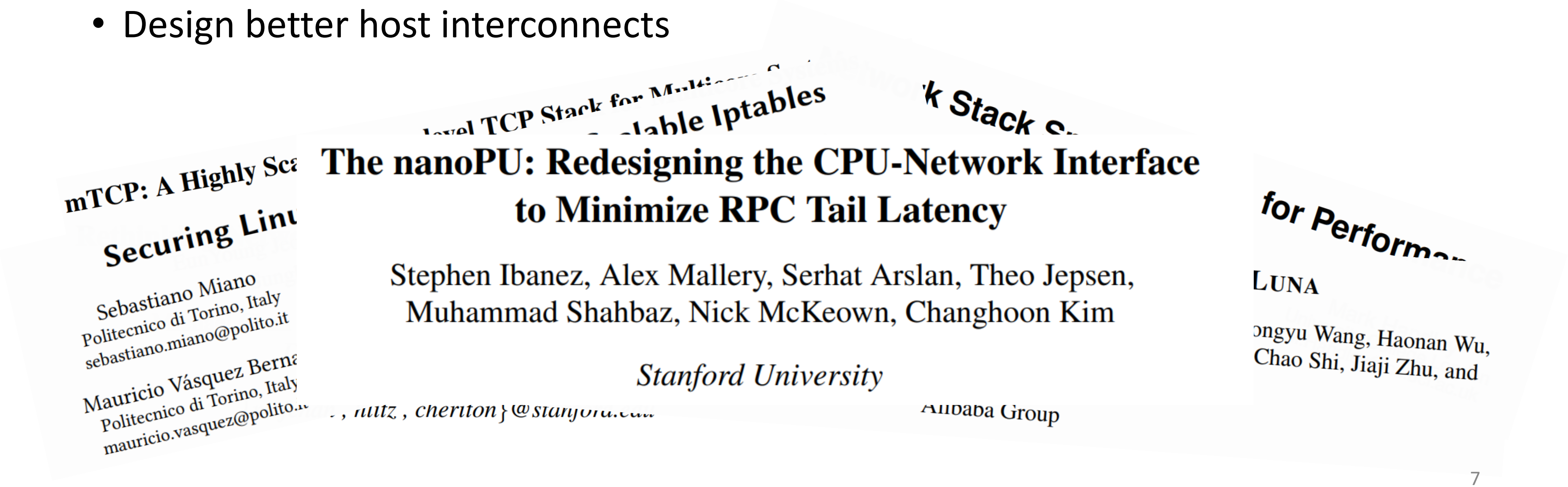


How to speed up software packet processing?

- There have been significant efforts to speed up packet processing through
  - Better stack design
  - Removing user-kernel crossings
  - Running software at lower layers of the stack

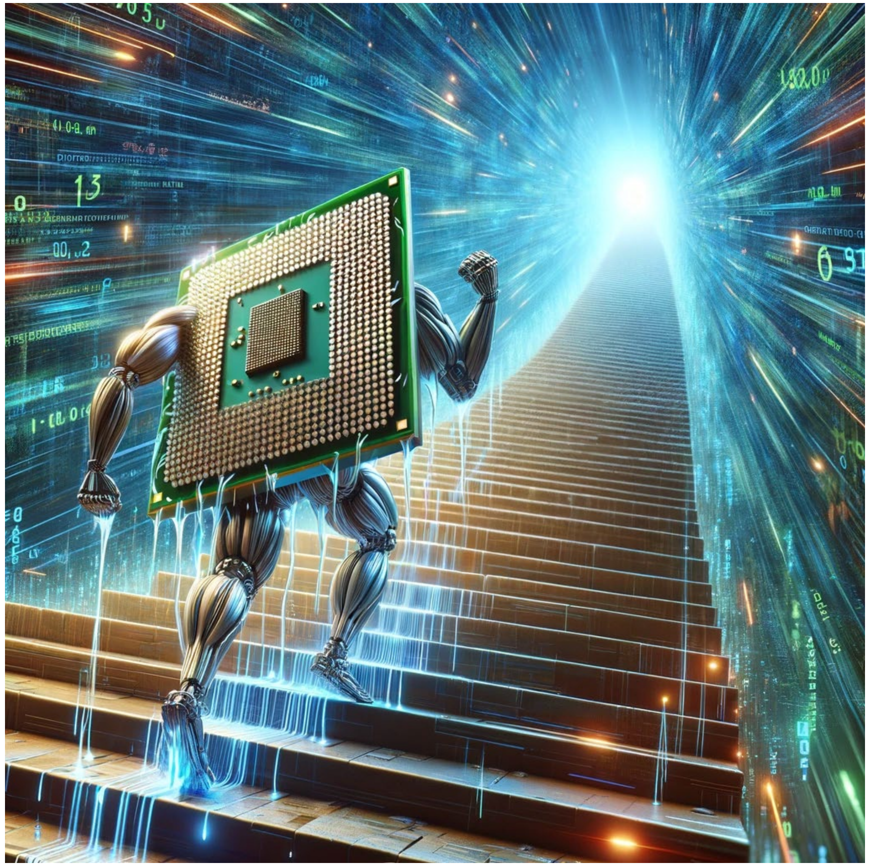
How to speed up software packet processing?

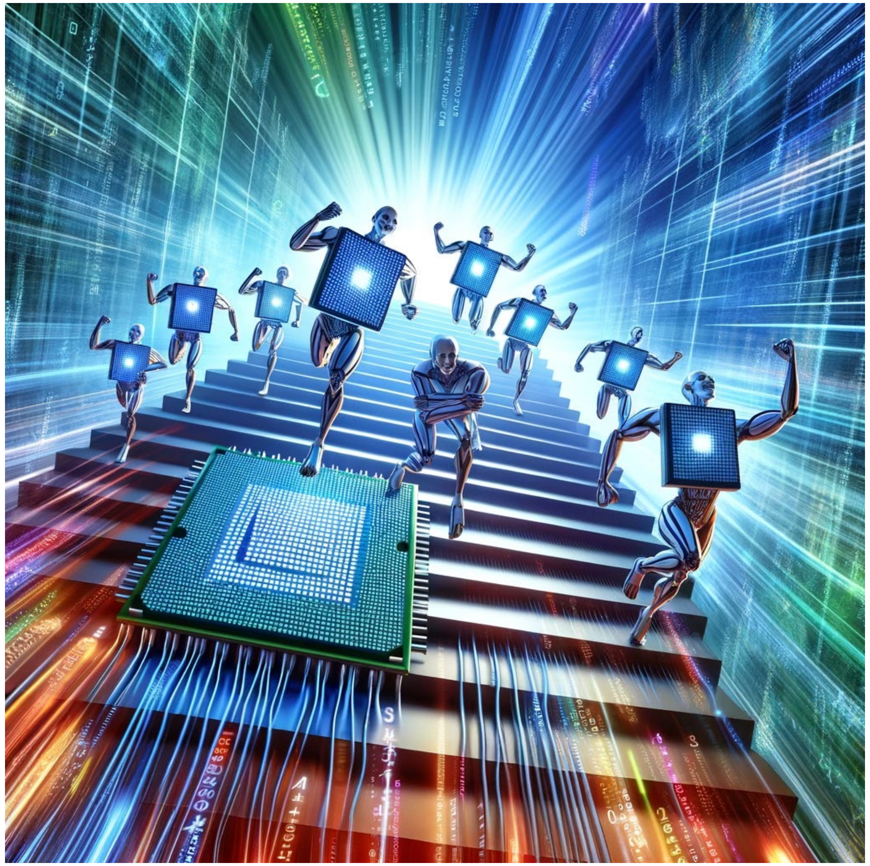
- There have been significant efforts to speed up packet processing through
  - Better stack design
  - Removing user-kernel crossings
  - Running software at lower layers of the stack
  - Design better host interconnects



This talk: Scaling Packet Processing using multiple cores

- **We present** a principle that enables **scaling a single, stateful** packet processing programs across multiple cores





What are the existing approaches? #1: Shared state

What are the existing approaches? #1: Shared state

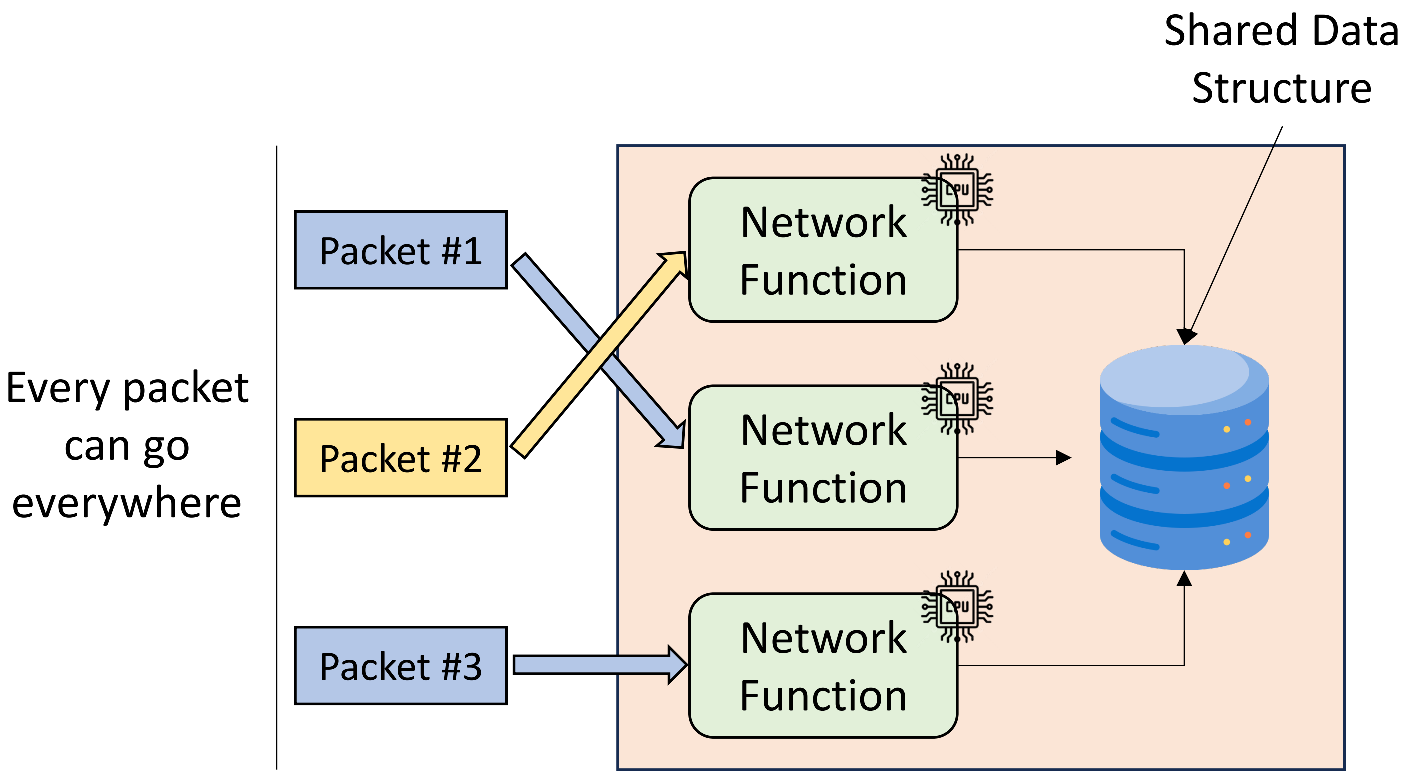
- Stateless applications are easy to scale, however…

What are the existing approaches? #1: Shared state

- Stateless applications are easy to scale, however…
- …many packet processing applications are **stateful**
  - Maintain and update the regions of memory across many packets

What are the existing approaches? #1: Shared state

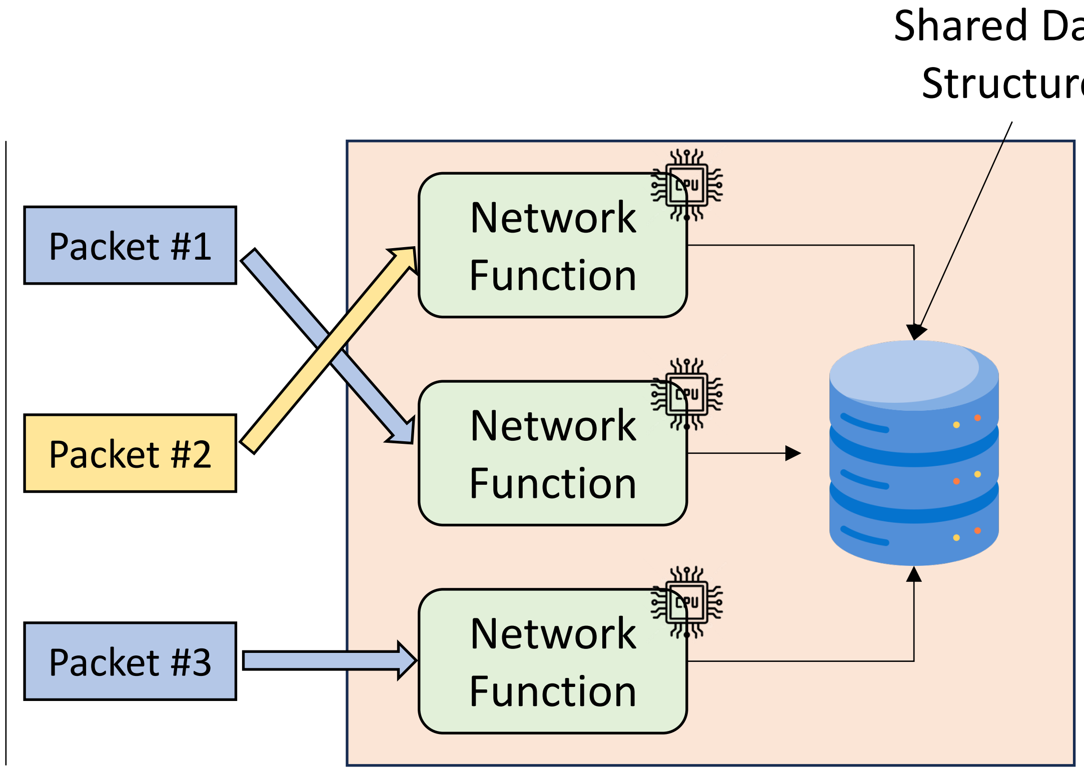
- Stateless applications are easy to scale, however…
- …many packet processing applications are **stateful**
  - Maintain and update the regions of memory across many packets
- Shared data structure + **explicit synchronization**

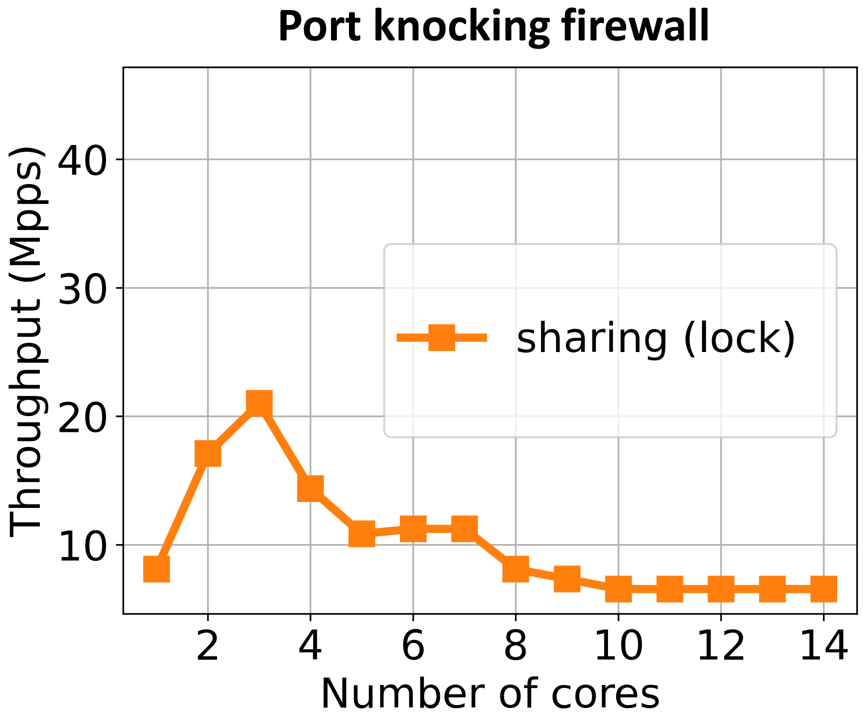


What are the existing approaches? #1: Shared state

- Stateless applications are easy to scale, however…
- …many packet processing applications are **stateful**
  - Maintain and update the regions of memory across many packets
- Shared data structure + **explicit synchronization**

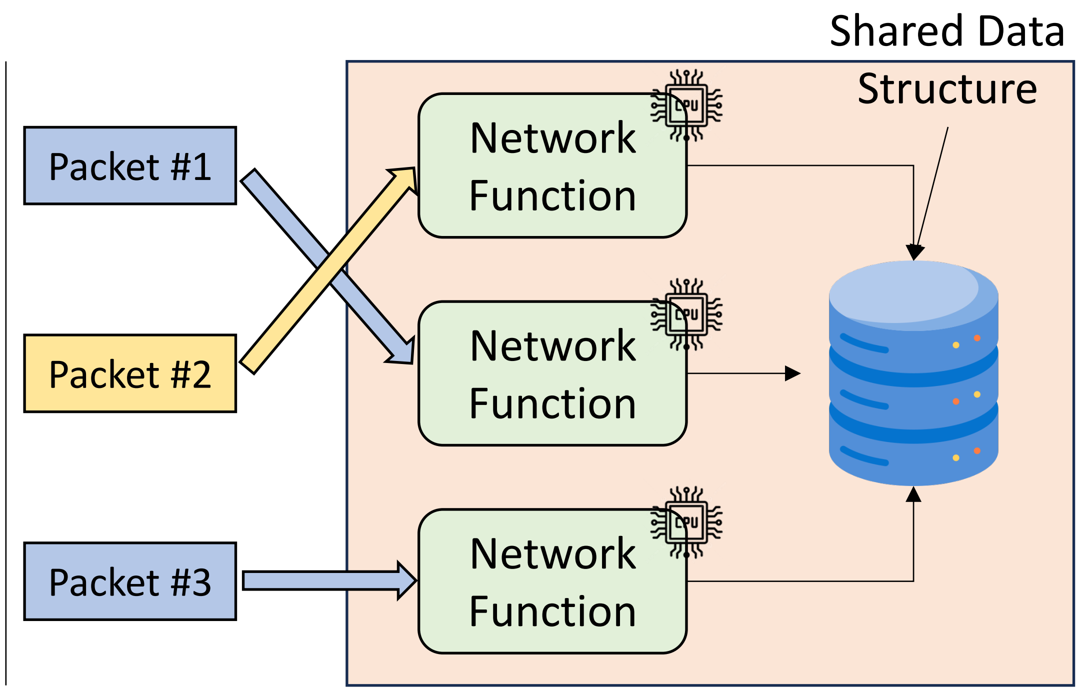
Every packet can go everywhere

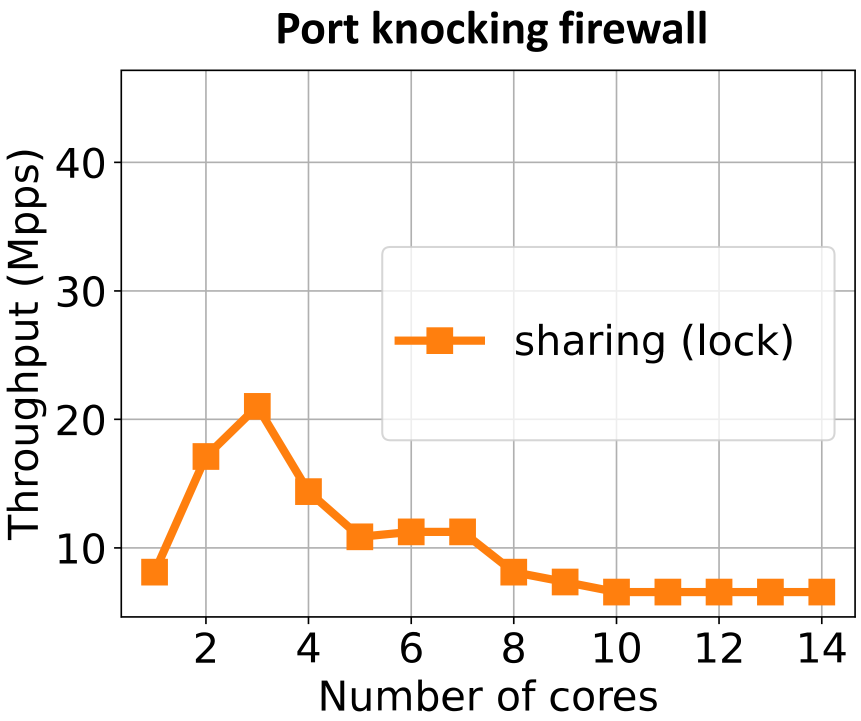




What are the existing approaches? #1: Shared state

Every packet can go everywhere



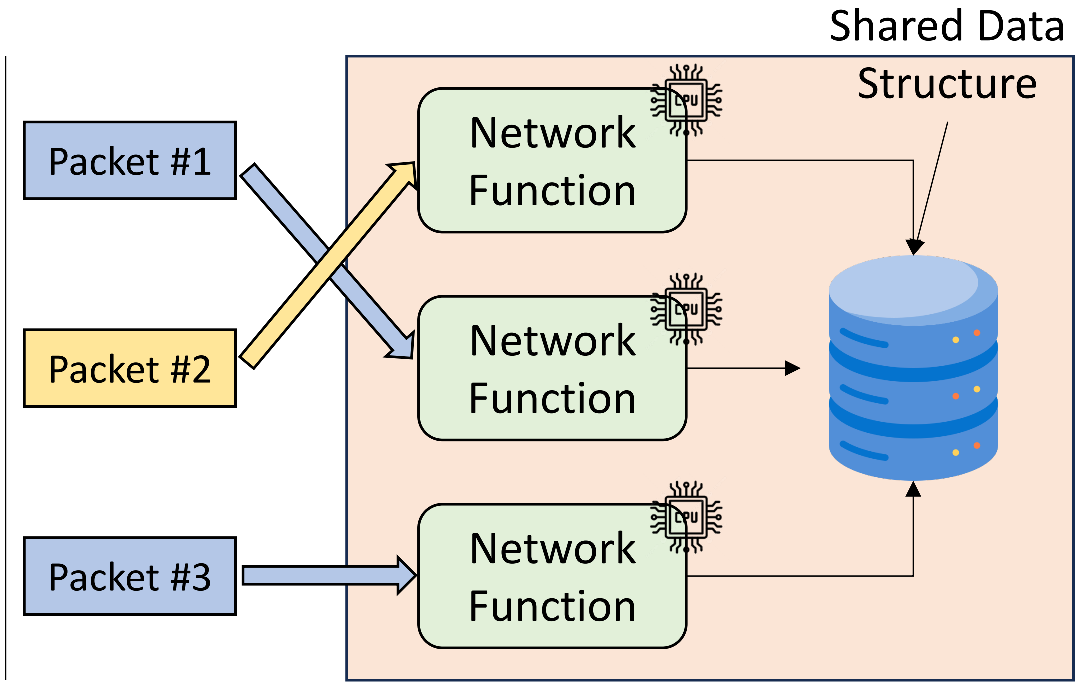


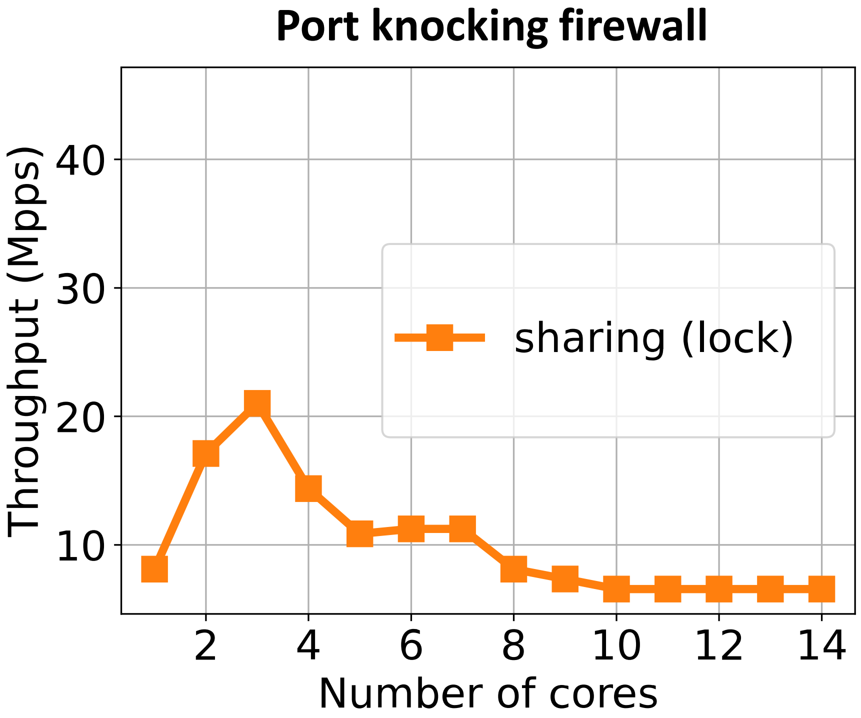
What are the existing approaches? #1: Shared state

✗ Flows in realistic network traffic follow heavy-tailed distributions

- Significant memory contention if packet are spread across cores

Every packet can go everywhere

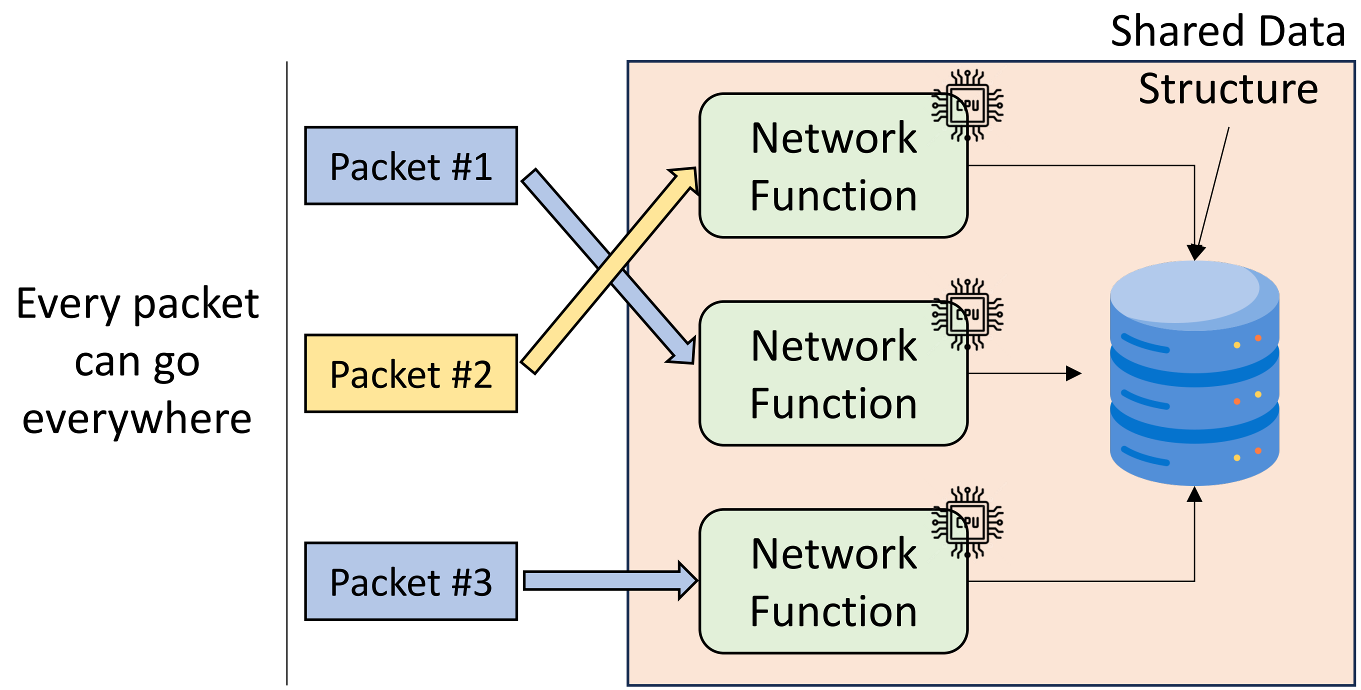


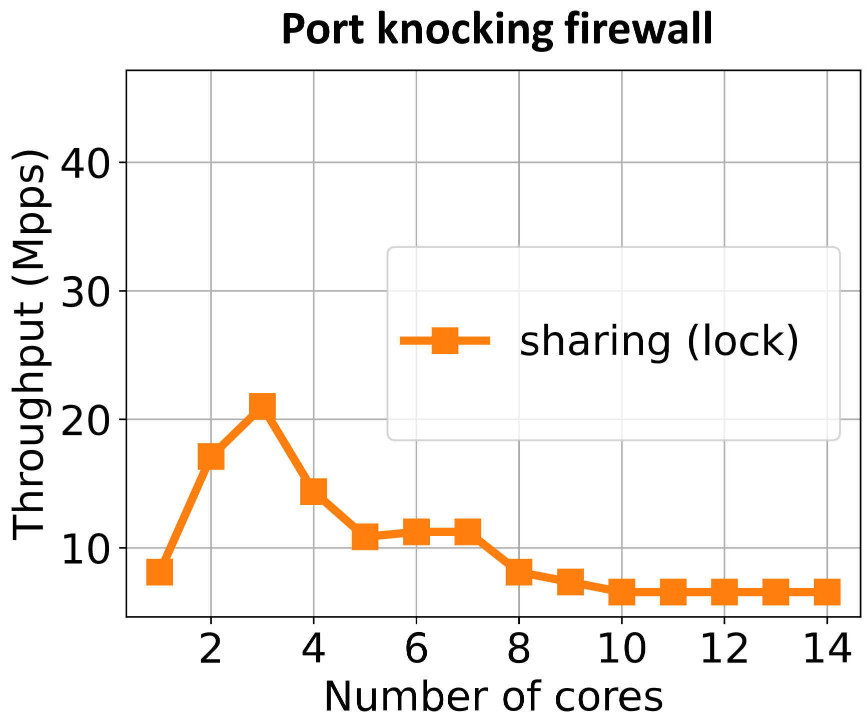


What are the existing approaches? #1: Shared state

- ✗ Flows in realistic network traffic follow heavy-tailed distributions
  - Significant memory contention if packet are spread across cores

- ✗ In many programs, state update operations are too complex to be implemented in transactional hardware (i.e., fetch-add-write)





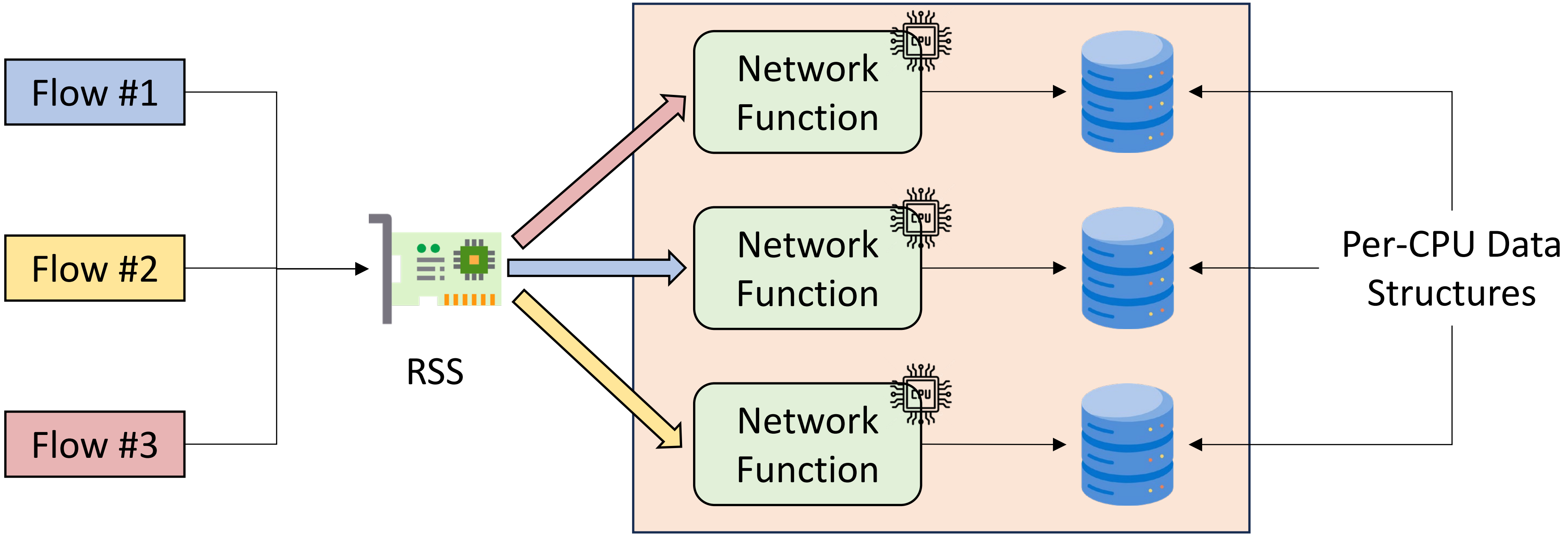
What are the existing approaches? #2: Sharding

What are the existing approaches? #2: Sharding

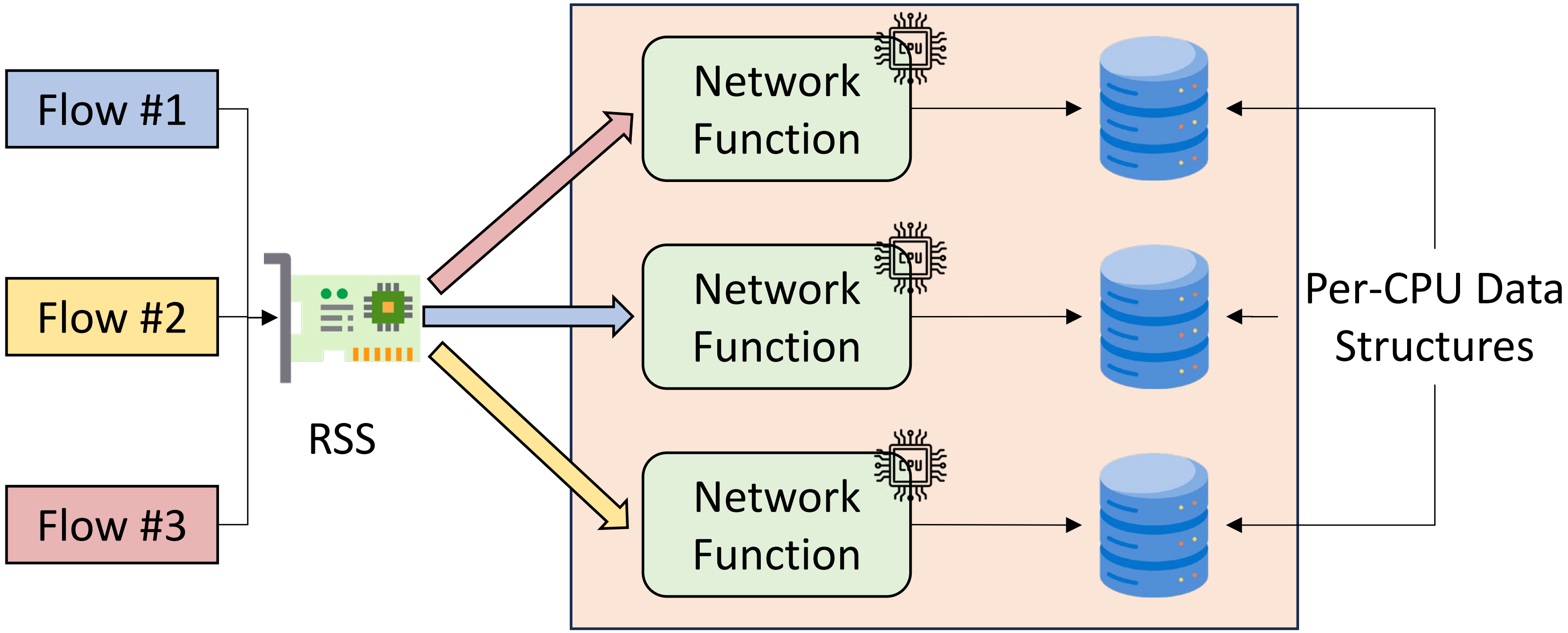
- Process packets that update the same memory at the same core, **sharding** the overall state of the program across cores
  - **NIC RSS** to direct packets from the same flow to the same core + **shared-nothing** data structures
  - The most used technique today

What are the existing approaches? #2: Sharding

- Process packets that update the same memory at the same core, **sharding** the overall state of the program across cores
  - **NIC RSS** to direct packets from the same flow to the same core + **shared-nothing** data structures
  - The most used technique today



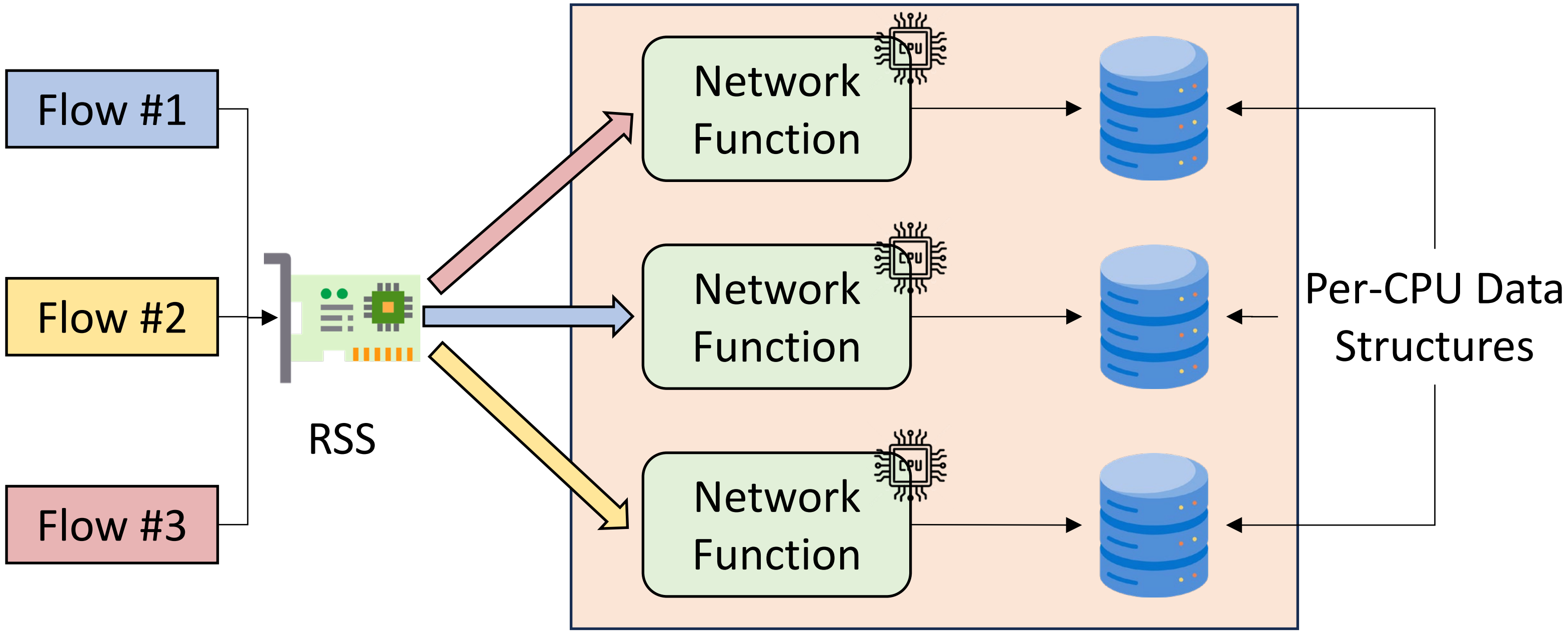
What are the existing approaches? #2: Sharding



What are the existing approaches? #2: Sharding

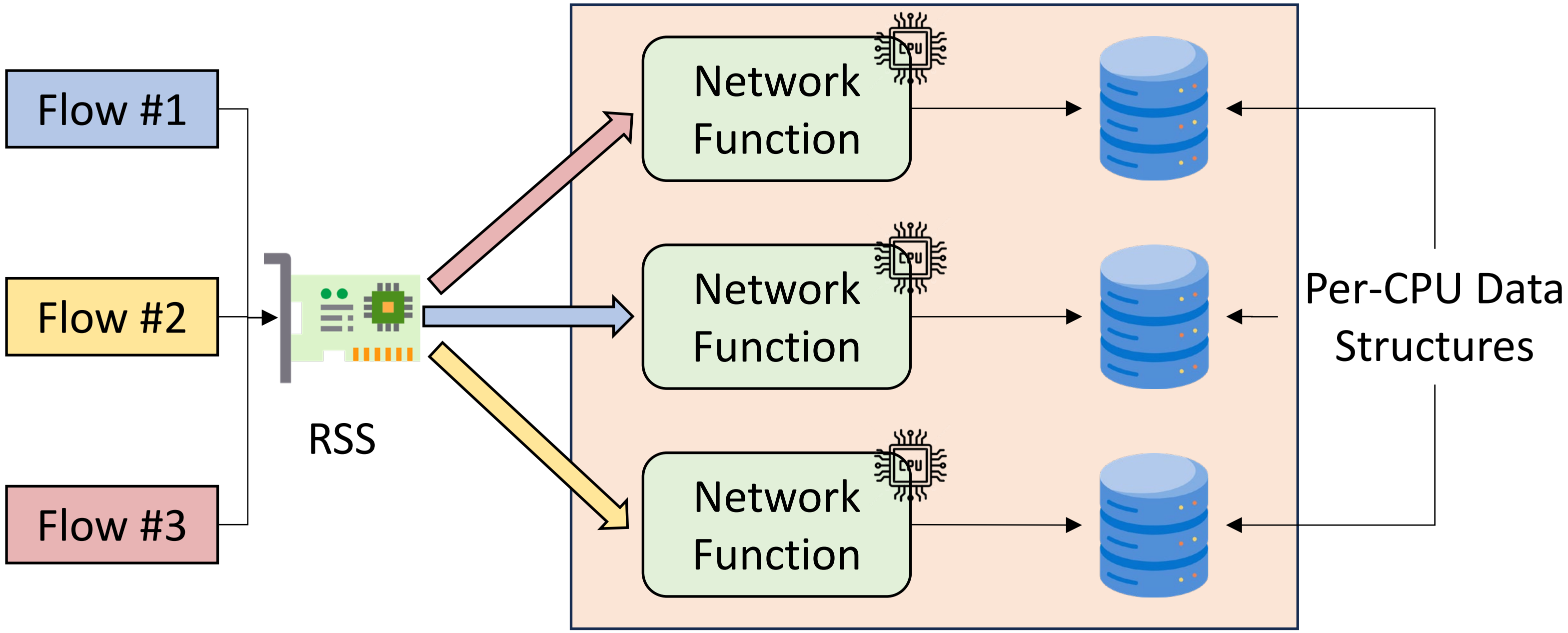
✗ Not always possible to avoid coordination through sharding

- There may be parts of the program state shared across packets (e.g., list of free ports in a NAT)



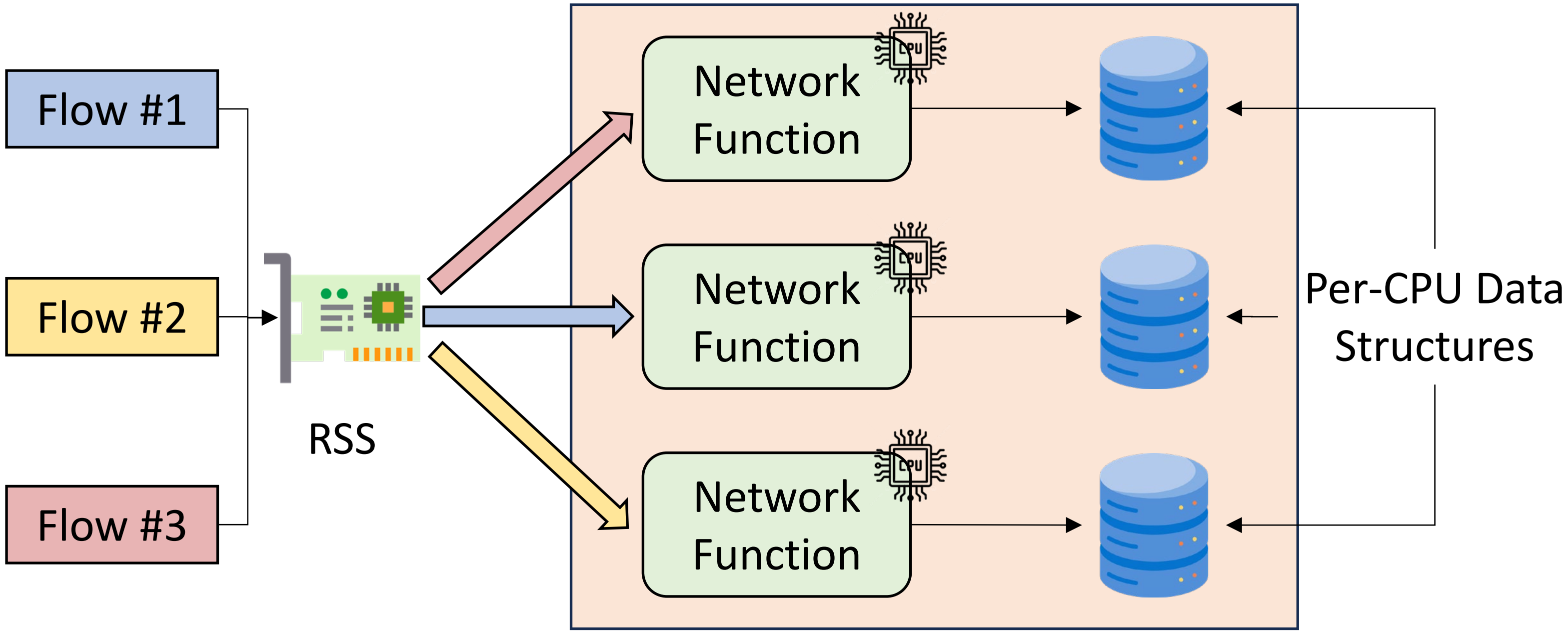
What are the existing approaches? #2: Sharding

- ✗ Not always possible to avoid coordination through sharding
  - There may be parts of the program state shared across packets (e.g., list of free ports in a NAT)
- ✗ Today's NIC RSS use a limited number of packet's header fields to steer packets



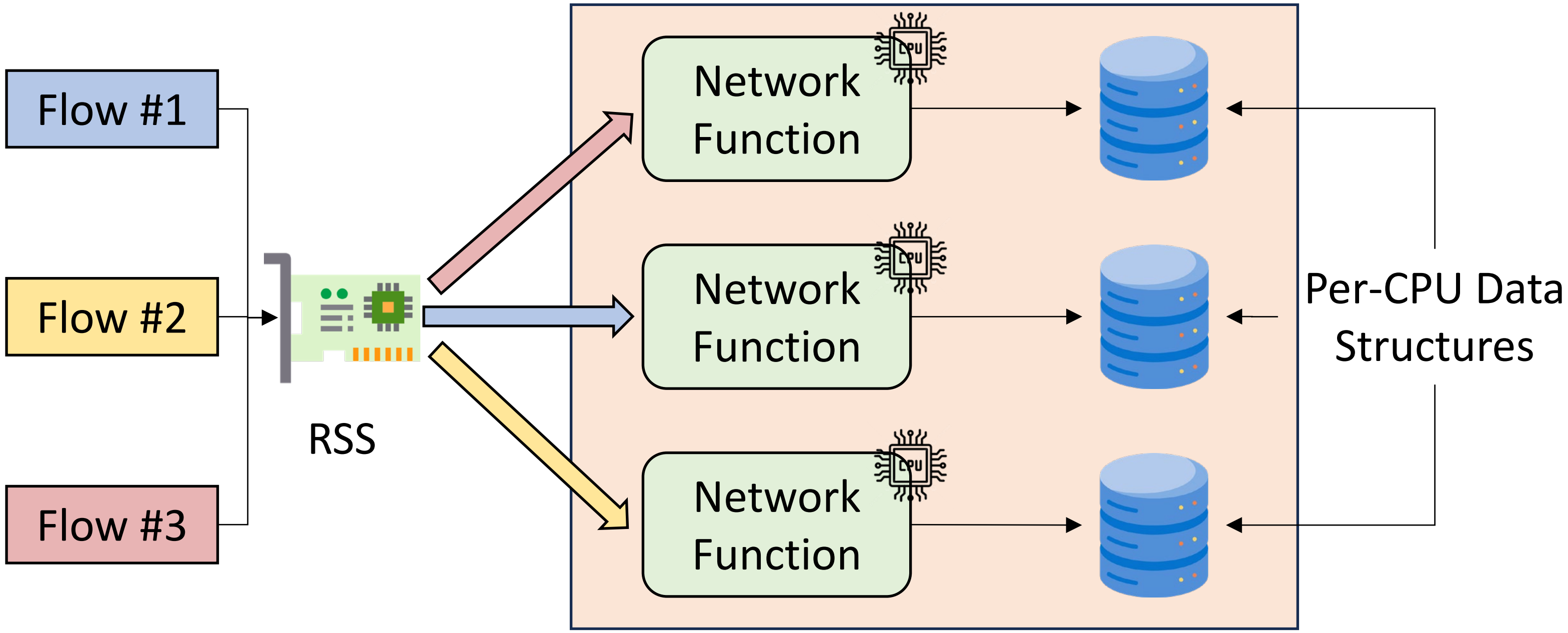
What are the existing approaches? #2: Sharding

- ✗ Not always possible to avoid coordination through sharding
  - There may be parts of the program state shared across packets (e.g., list of free ports in a NAT)
- ✗ Today's NIC RSS use a limited number of packet's header fields to steer packets
- ✗ Creates load imbalance across cores if some flows are heavier than others

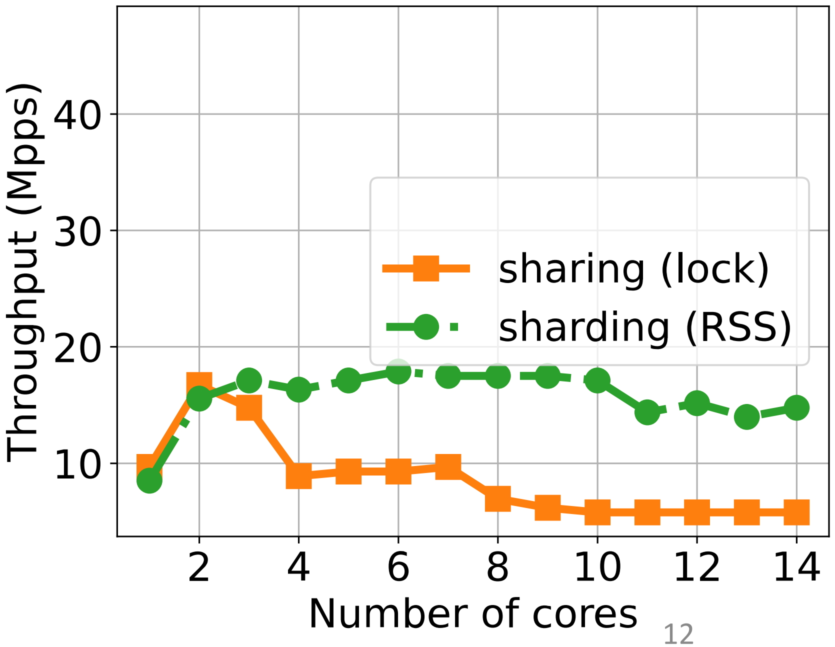


What are the existing approaches? #2: Sharding

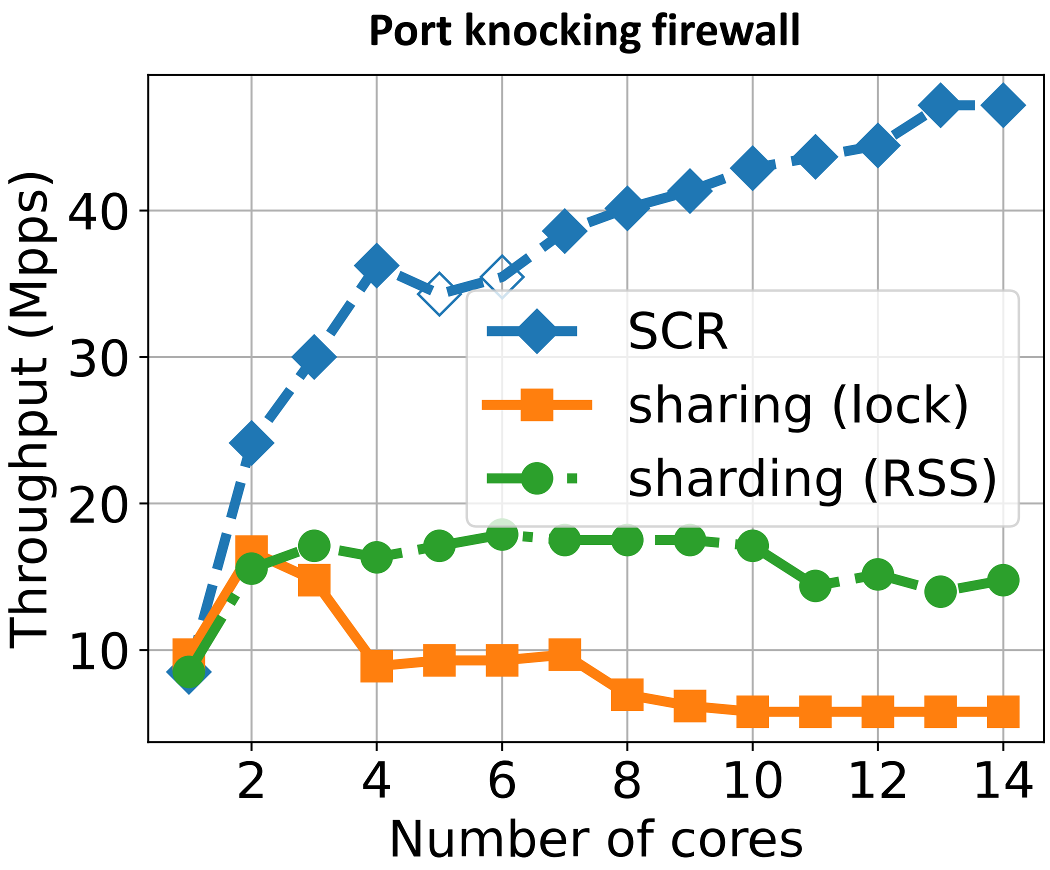
- ✗ Not always possible to avoid coordination through sharding
  - There may be parts of the program state shared across packets (e.g., list of free ports in a NAT)
- ✗ Today's NIC RSS use a limited number of packet's header fields to steer packets
- ✗ Creates load imbalance across cores if some flows are heavier than others



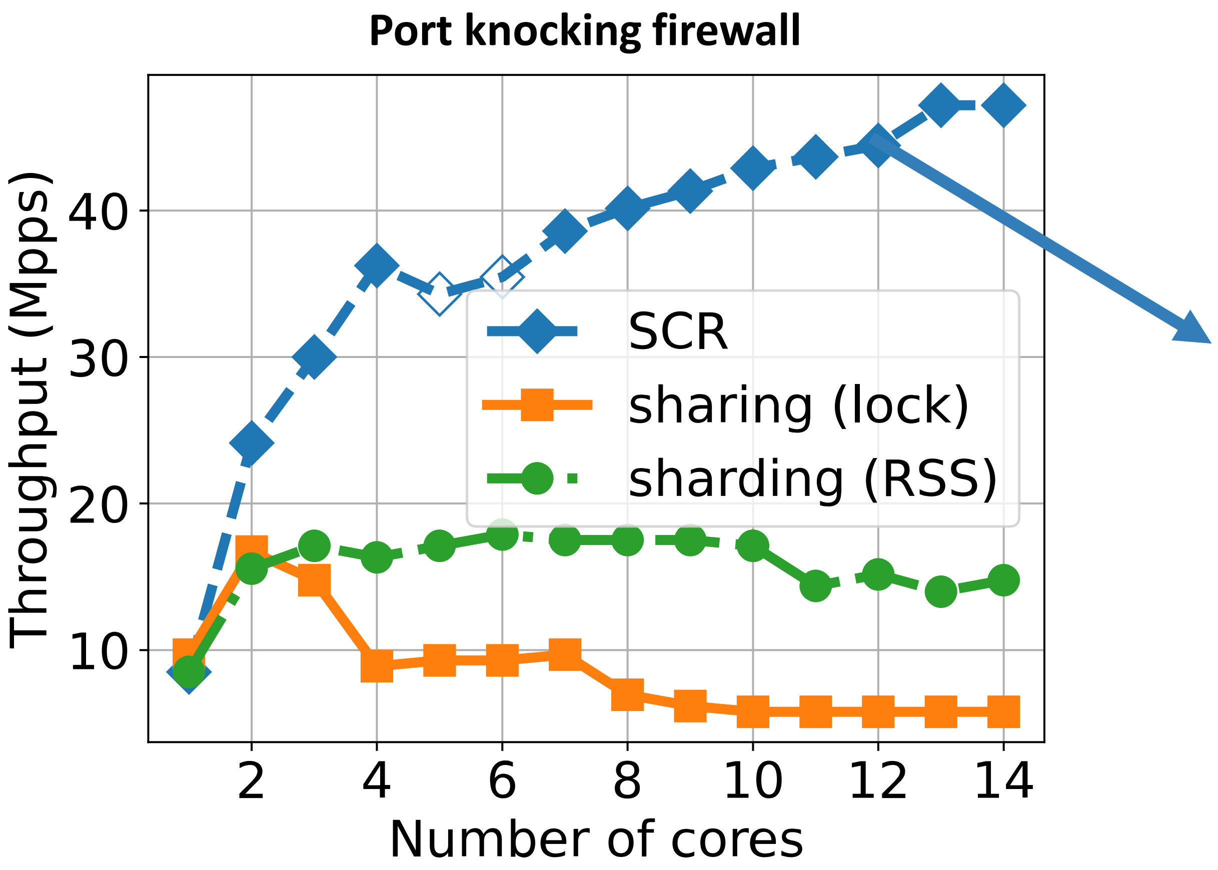
Port knocking firewall



State Compute Replication (SRC):
Scaling Single, Stateful Flows across multiple cores



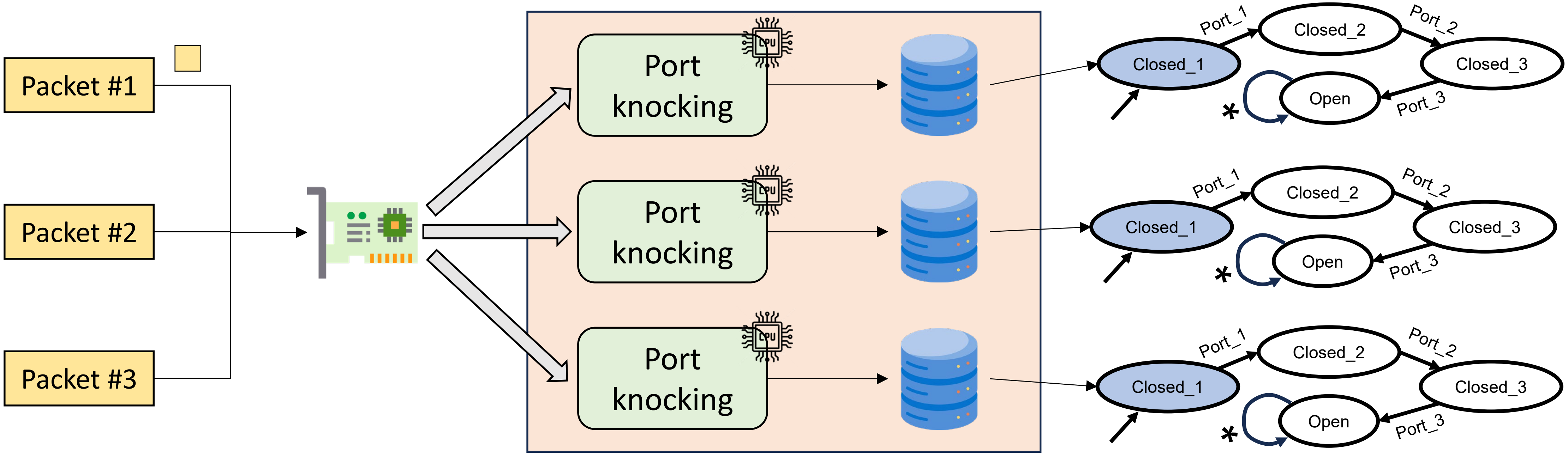
State Compute Replication (SRC):
Scaling Single, Stateful Flows across multiple cores



Throughput scales **linearly, deterministically** and **independently** from **flow size distribution**

How does this work? – Running Example

- **Port knocking firewall**
  - If a source transmits IPv4/TCP packets with the correct sequence of TCP destination ports, all further communication is permitted

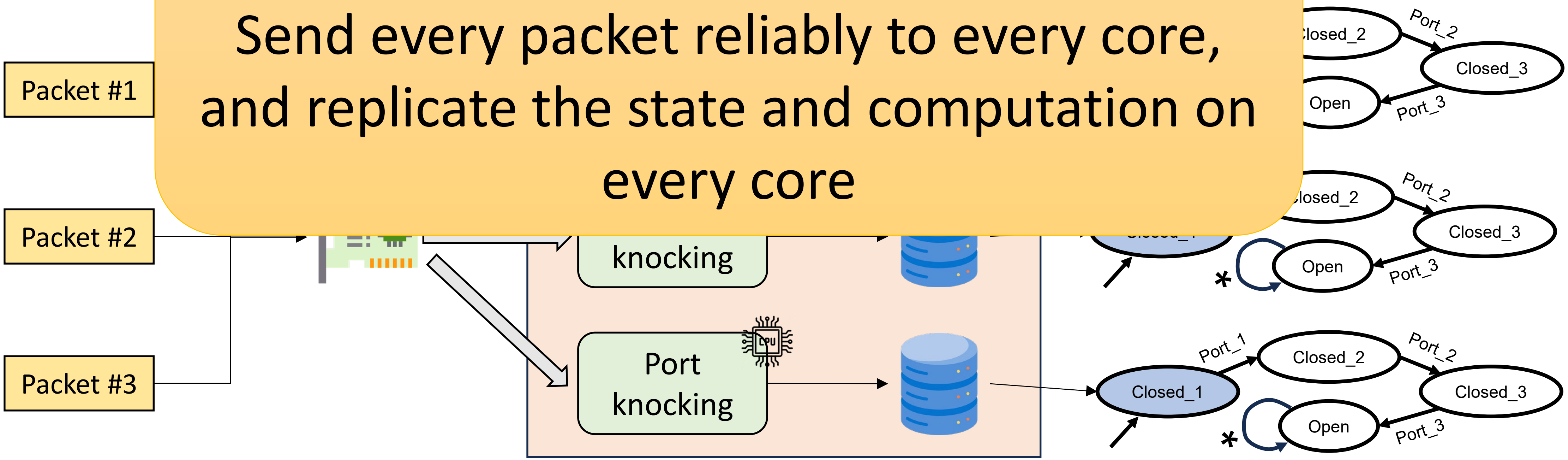


How does this work? – Running Example

- **Port knocking firewall**
  - If a sourc … TCP
    destina

Idea #1: Replication for correctness

Send every packet reliably to every core, and replicate the state and computation on every core



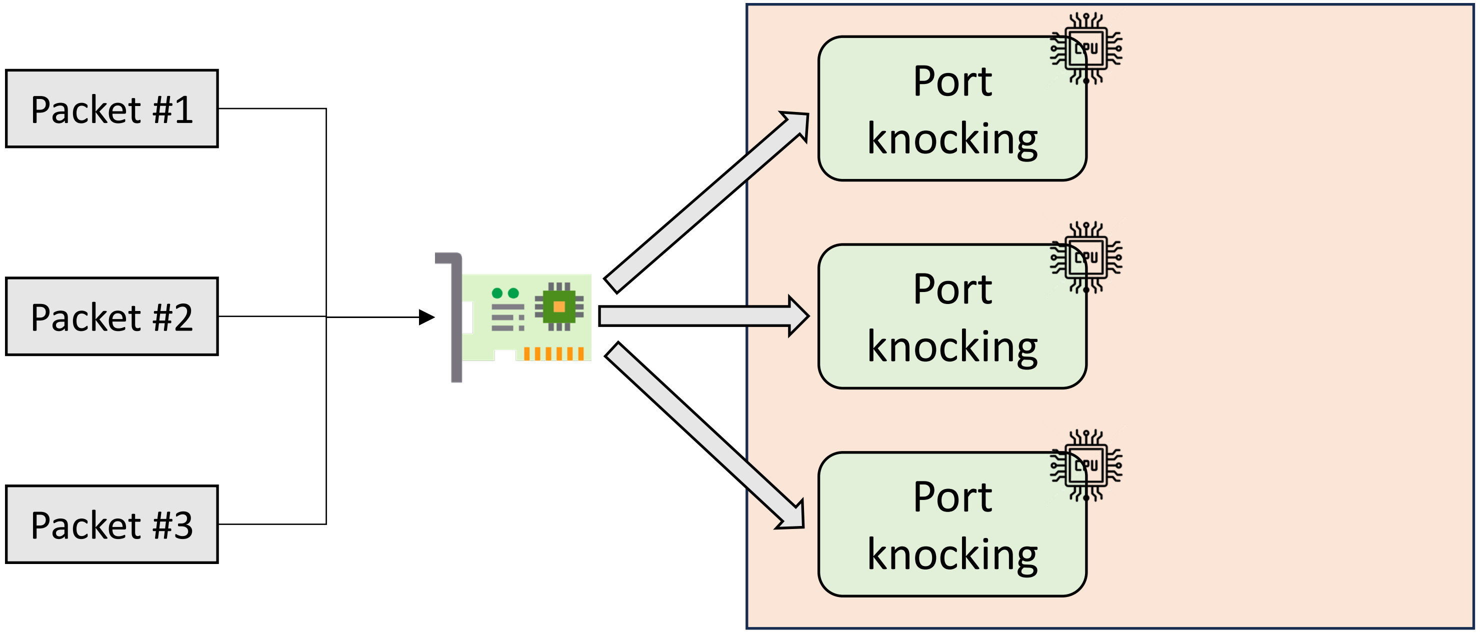
State Compute Replication (SRC):
Scaling Single, Stateful Flows across multiple cores

State Compute Replication (SRC): Scaling Single, Stateful Flows across multiple cores

- Packets can be perfectly sprayed across all the cores (or a set of cores)

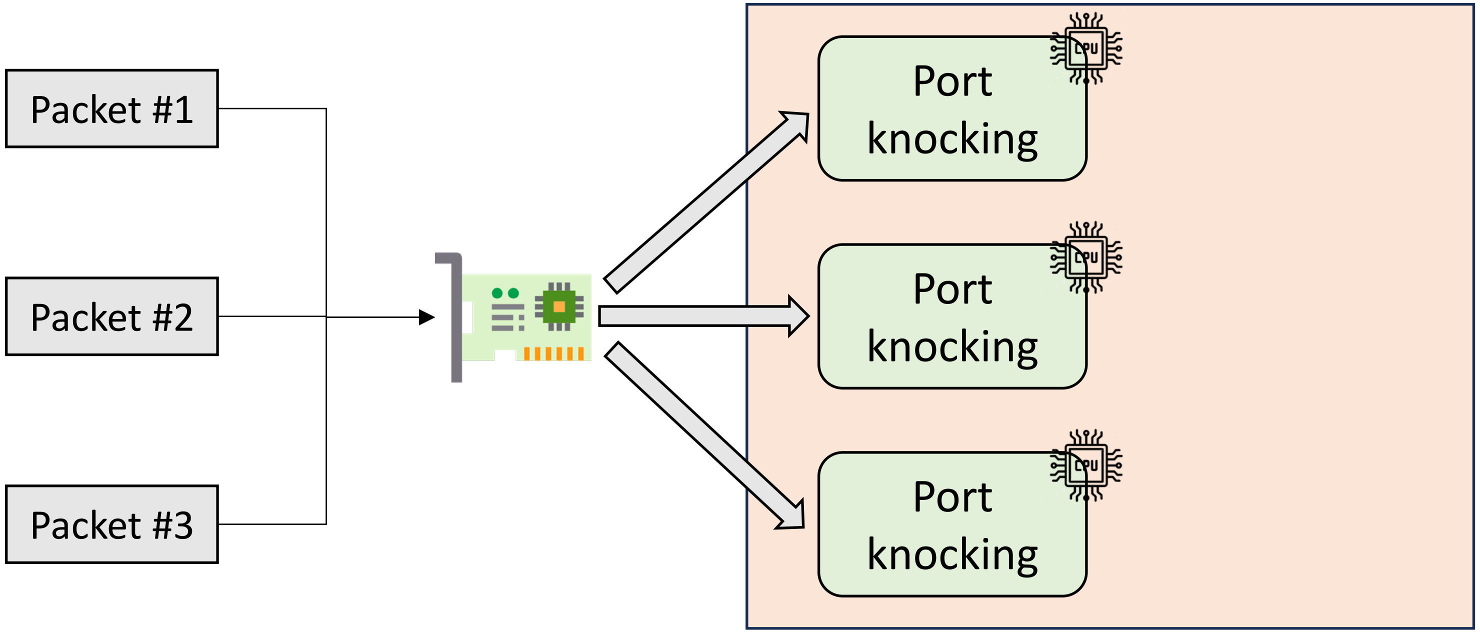
State Compute Replication (SRC): Scaling Single, Stateful Flows across multiple cores

- Packets can be perfectly sprayed across all the cores (or a set of cores)



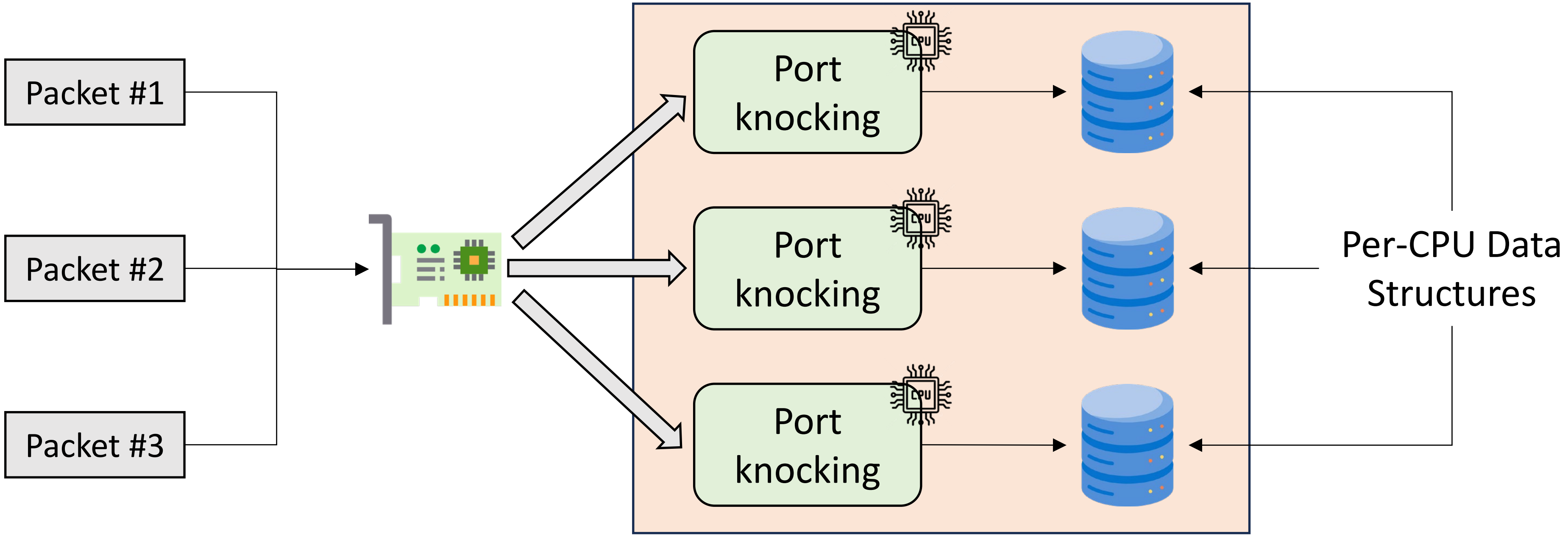
State Compute Replication (SRC): Scaling Single, Stateful Flows across multiple cores

- Packets can be perfectly sprayed across all the cores (or a set of cores)
- Every core will have its own state without sharing between the cores



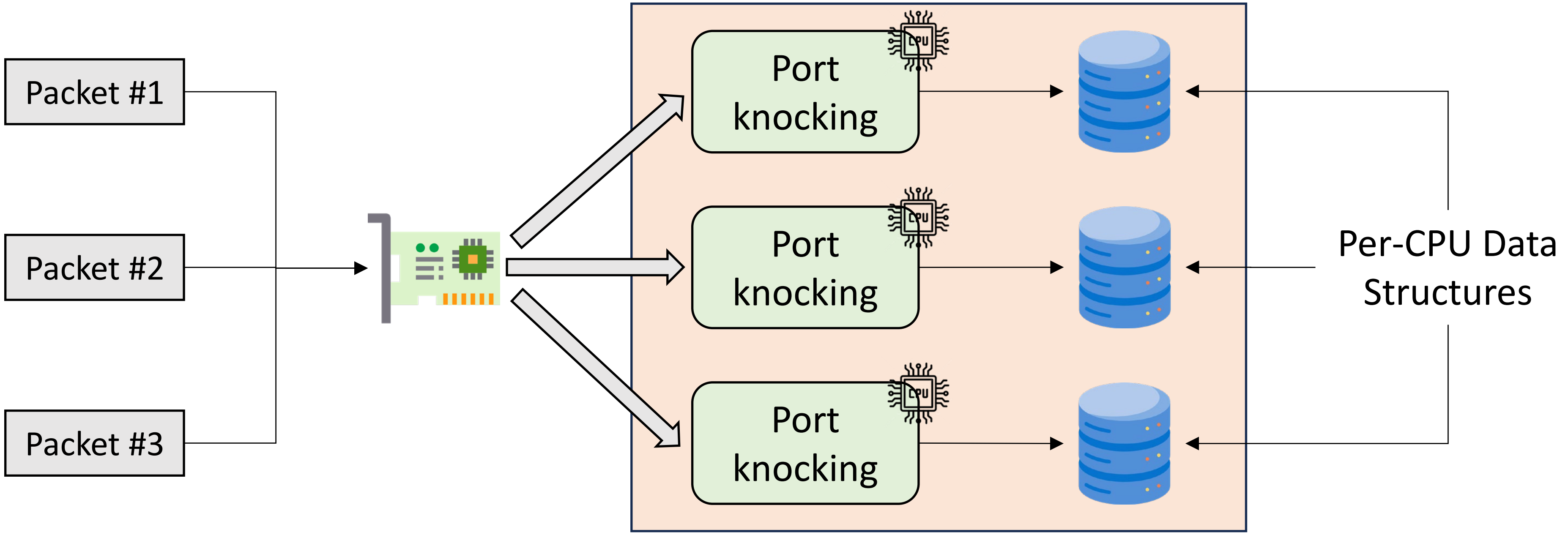
State Compute Replication (SRC): Scaling Single, Stateful Flows across multiple cores

- Packets can be perfectly sprayed across all the cores (or a set of cores)
- Every core will have its own state without sharing between the cores



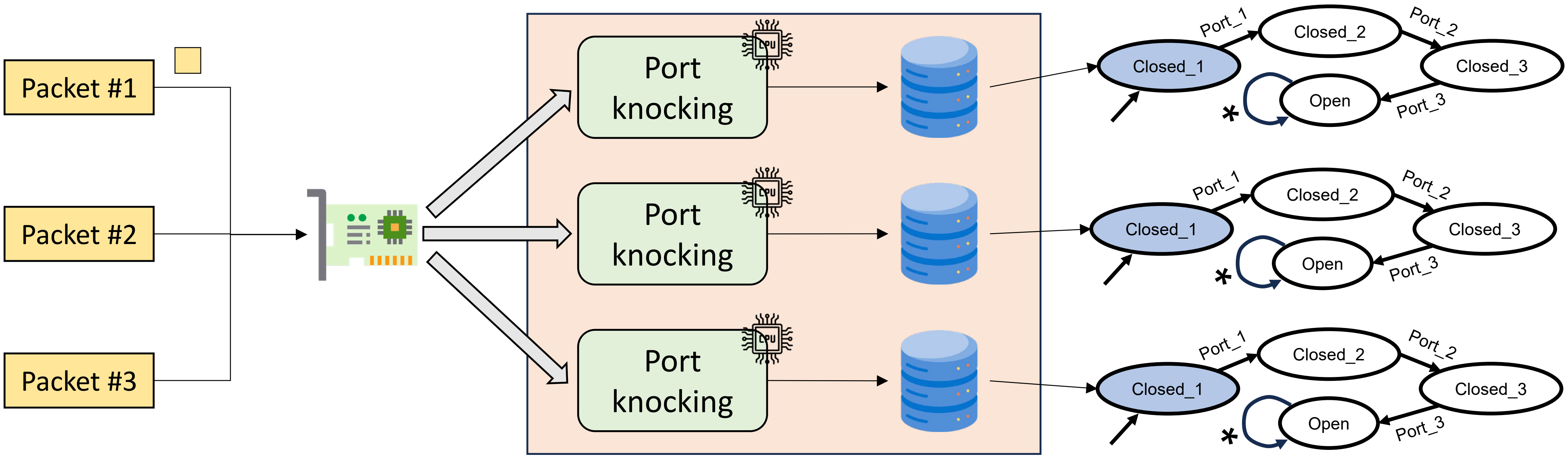
State Compute Replication (SRC):
Scaling Single, Stateful Flows across multiple cores

- Packets can be perfectly sprayed across all the cores (or a set of cores)
- Every core will have its own state without sharing between the cores
- **Given a set of *N* cores, every packet *p* will go to core *c = p % (N – 1)***



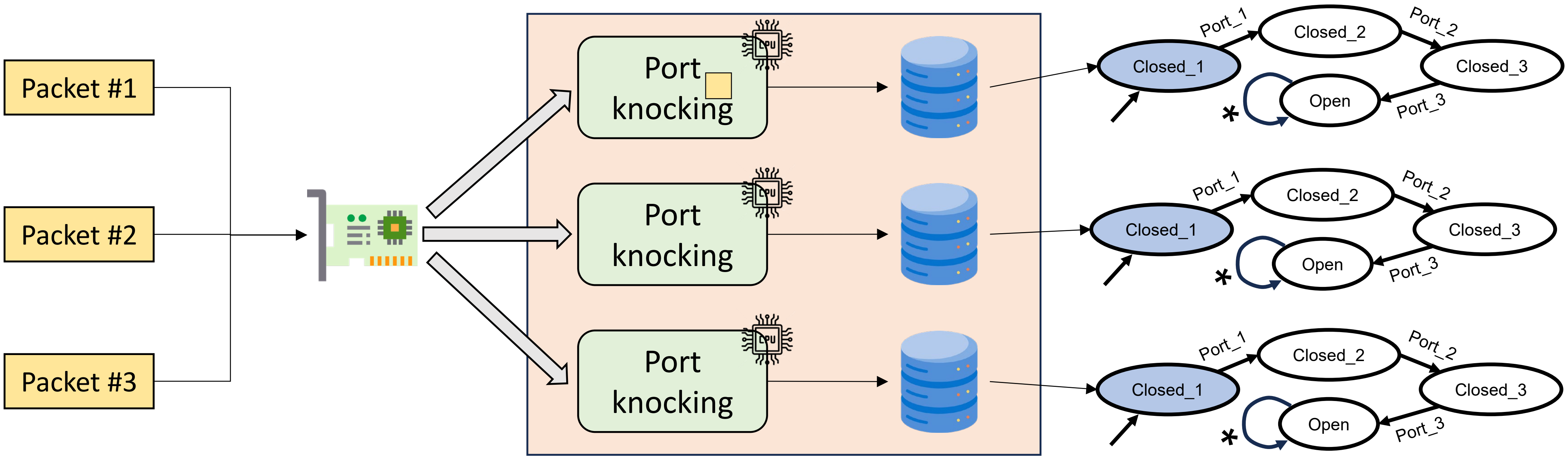
How does this work? – Running Example

- **Port knocking firewall**
  - If a source transmits IPv4/TCP packets with the correct sequence of TCP destination ports, all further communication is permitted



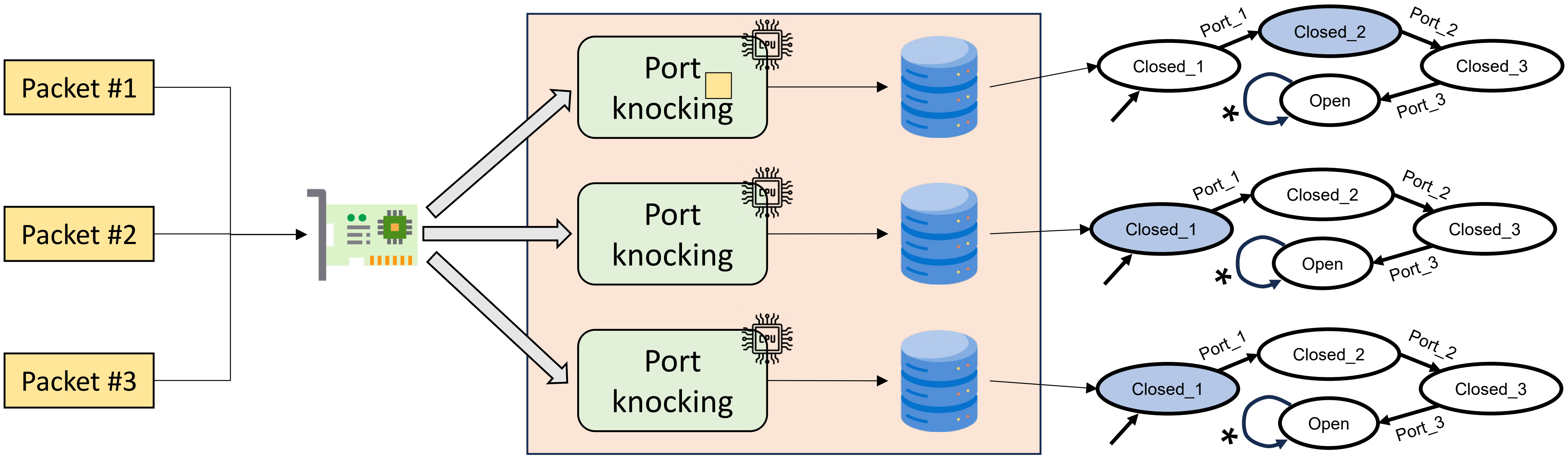
How does this work? – Running Example

- **Port knocking firewall**
  - If a source transmits IPv4/TCP packets with the correct sequence of TCP destination ports, all further communication is permitted



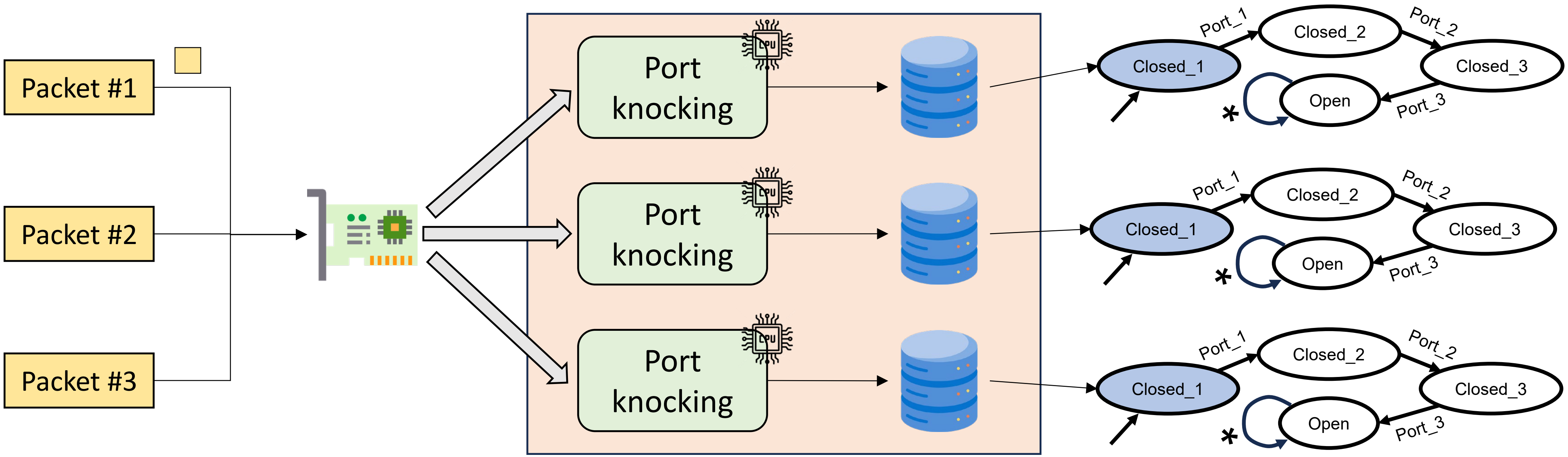
How does this work? – Running Example

- **Port knocking firewall**
  - If a source transmits IPv4/TCP packets with the correct sequence of TCP destination ports, all further communication is permitted



How does this work? – Running Example

- **Port knocking firewall**
  - If a source transmits IPv4/TCP packets with the correct sequence of TCP destination ports, all further communication is permitted

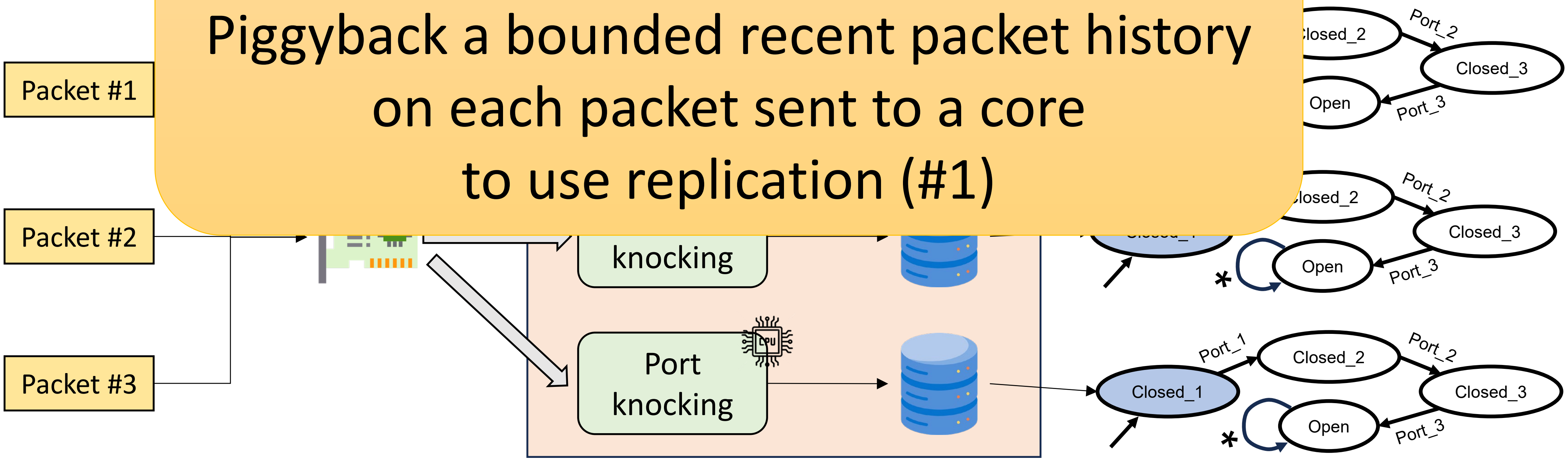


How does this work? – Running Example

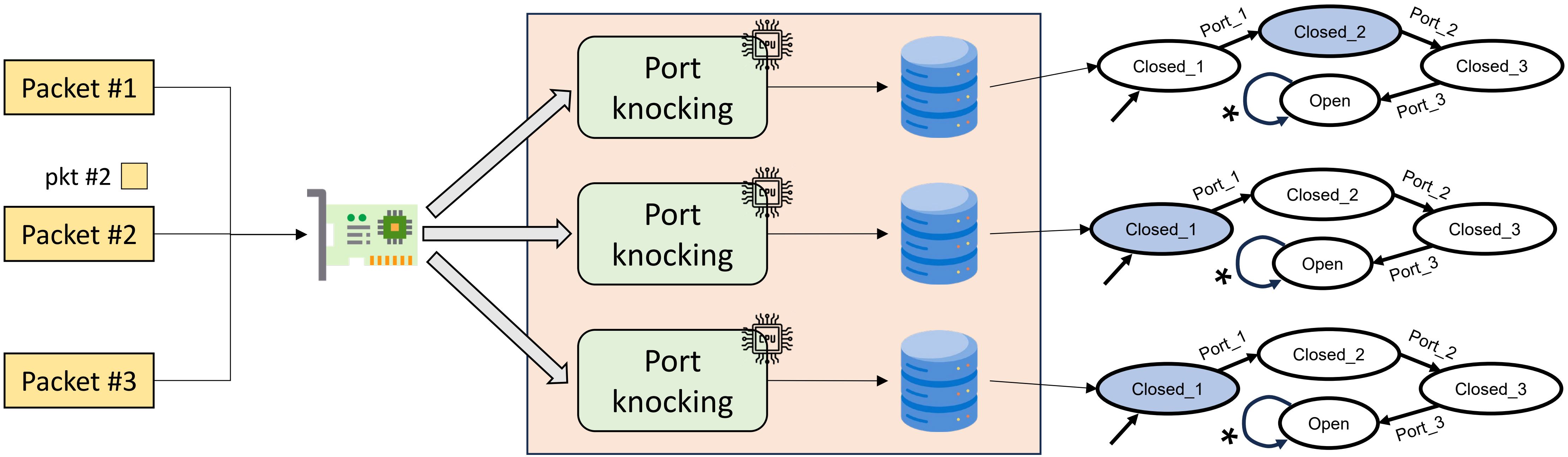
- **Port knocking firewall**
  - If a sourc… of TCP destina…

Idea #2: State Compute Replication

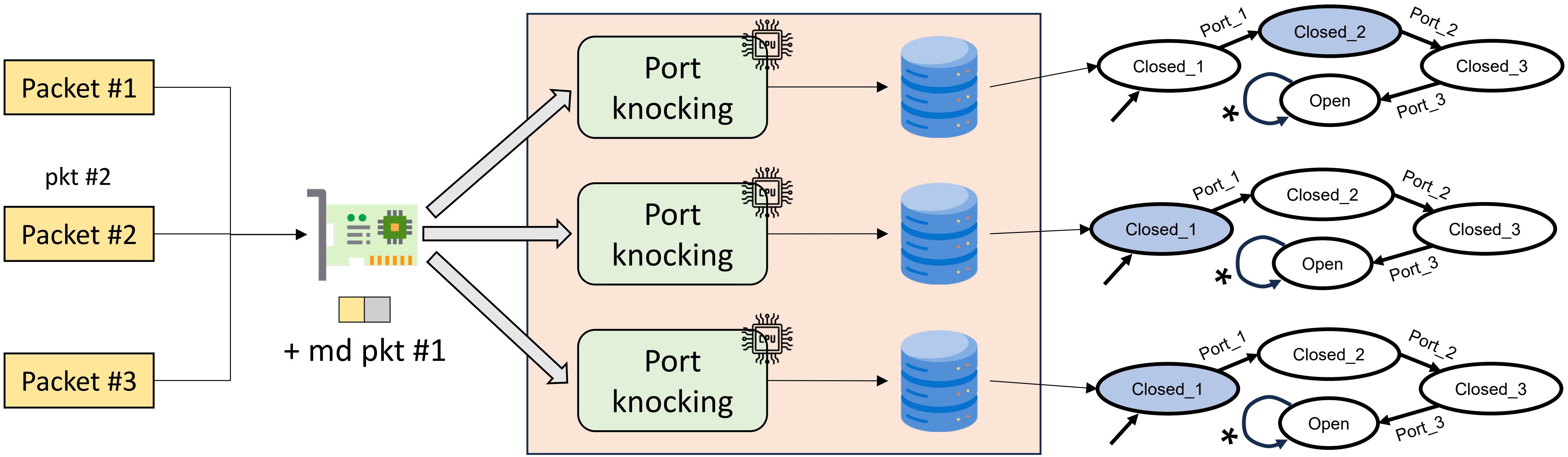
Piggyback a bounded recent packet history
on each packet sent to a core
to use replication (#1)



How does this work? – Running Example

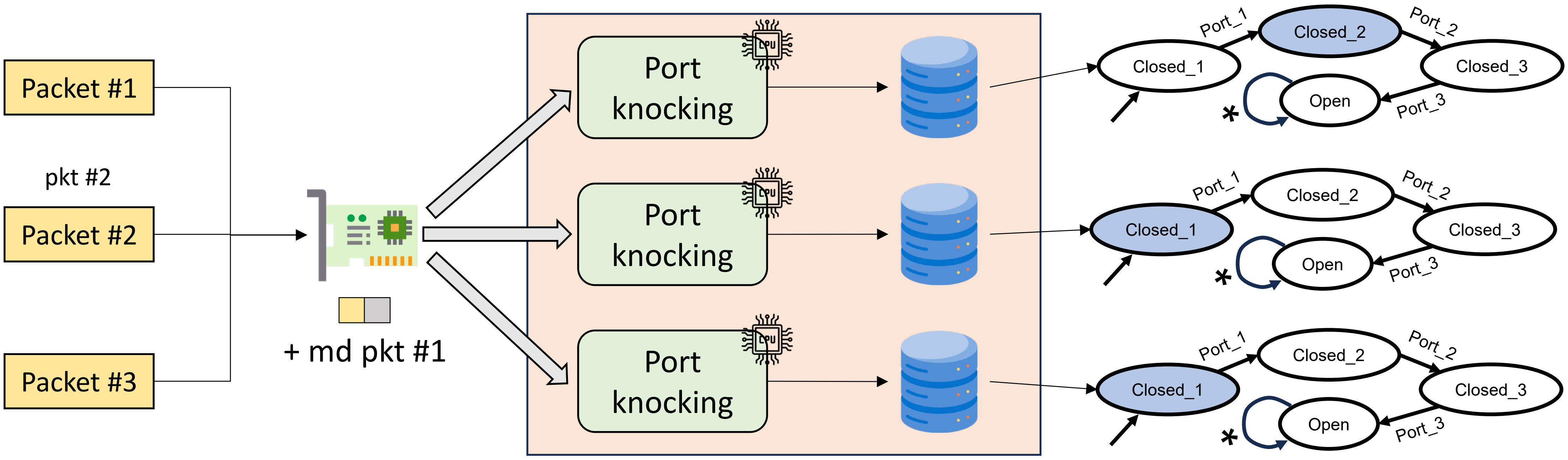


How does this work? – Running Example



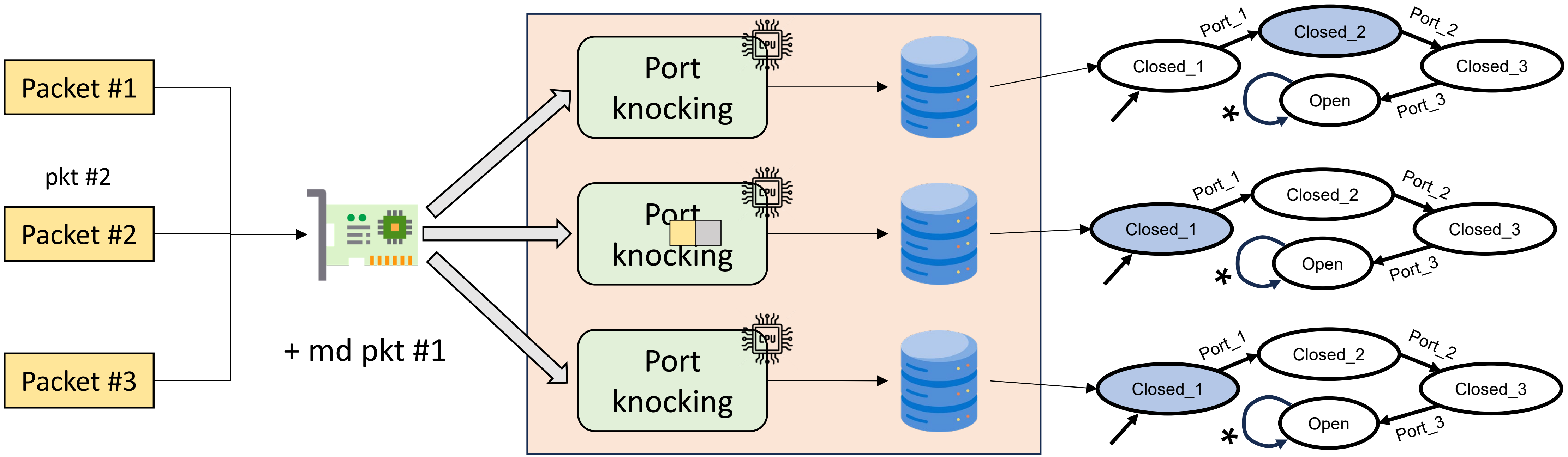
How does this work? – Running Example

- **Metadata**
  - Information of previous packets needed to update the state machine
    - In this example: **l3proto, l4proto, srcIP, dstPort**



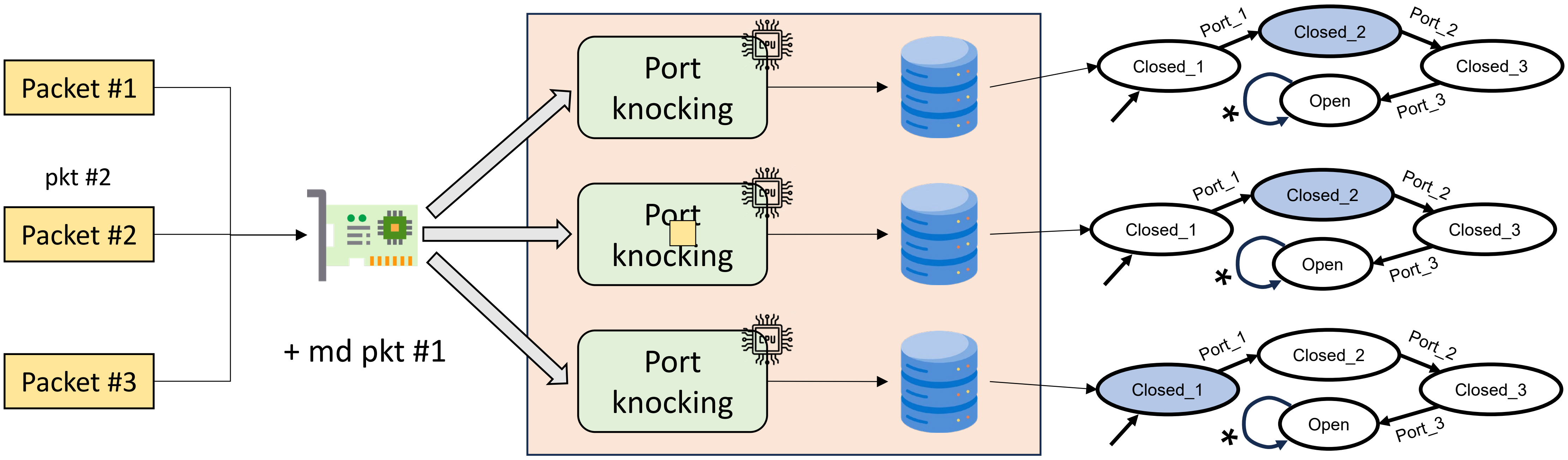
How does this work? – Running Example

- **Metadata**
  - Information of previous packets needed to update the state machine
    - In this example: **l3proto, l4proto, srcIP, dstPort**



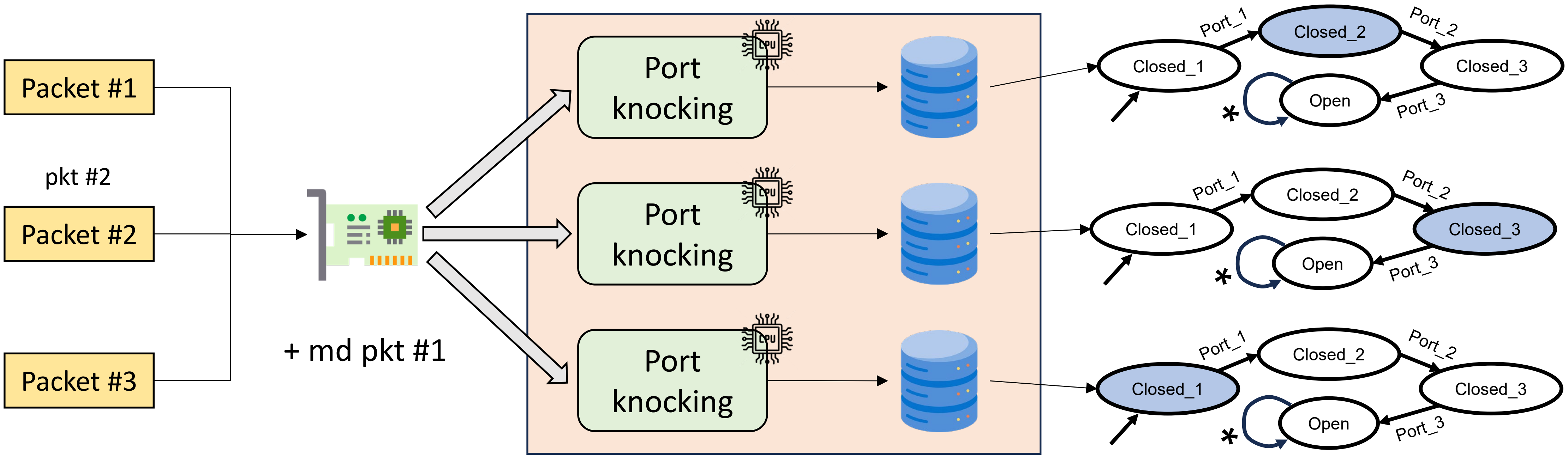
How does this work? – Running Example

- **Metadata**
  - Information of previous packets needed to update the state machine
    - In this example: **l3proto, l4proto, srcIP, dstPort**



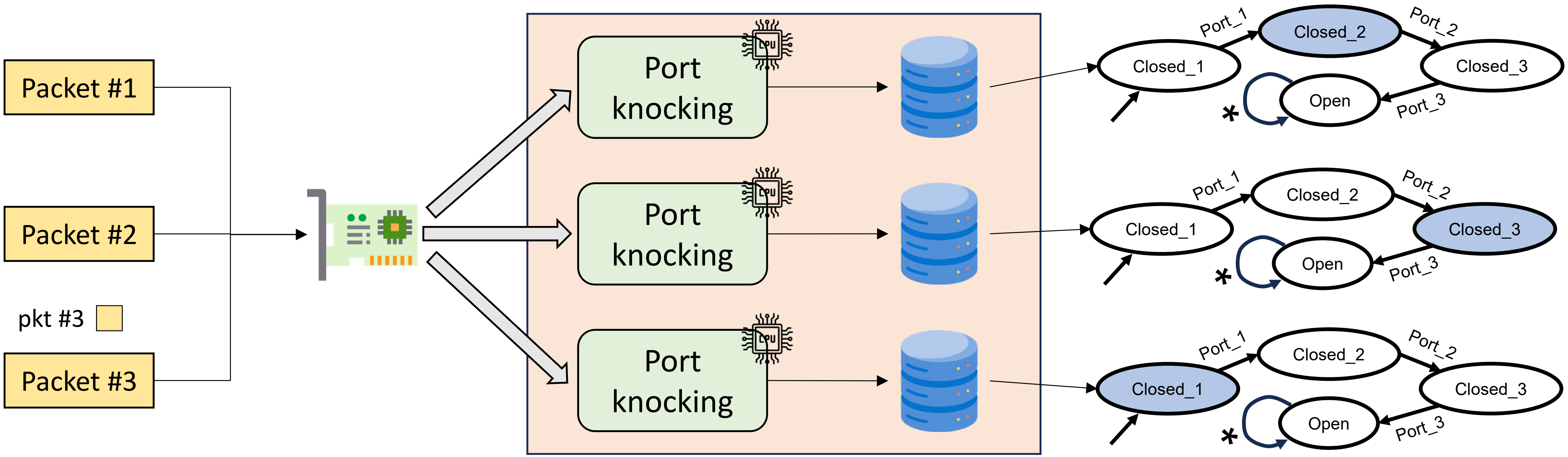
How does this work? – Running Example

- **Metadata**
  - Information of previous packets needed to update the state machine
    - In this example: **l3proto, l4proto, srcIP, dstPort**



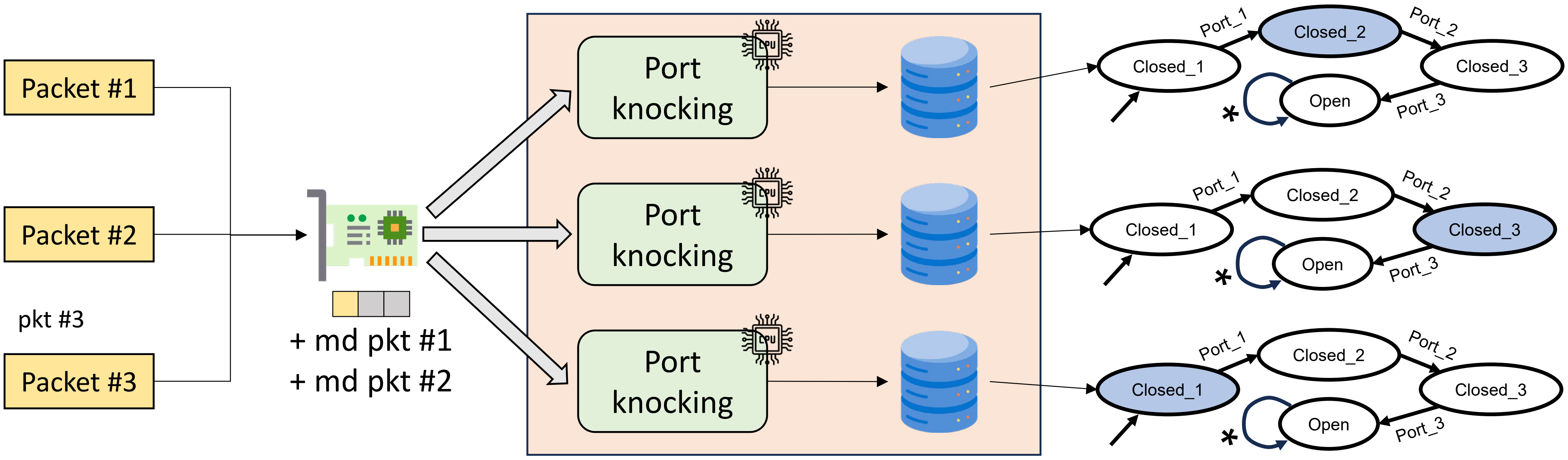
How does this work? – Running Example

- **Metadata**
  - Information of previous packets needed to update the state machine
    - In this example: **l3proto, l4proto, srcIP, dstPort**



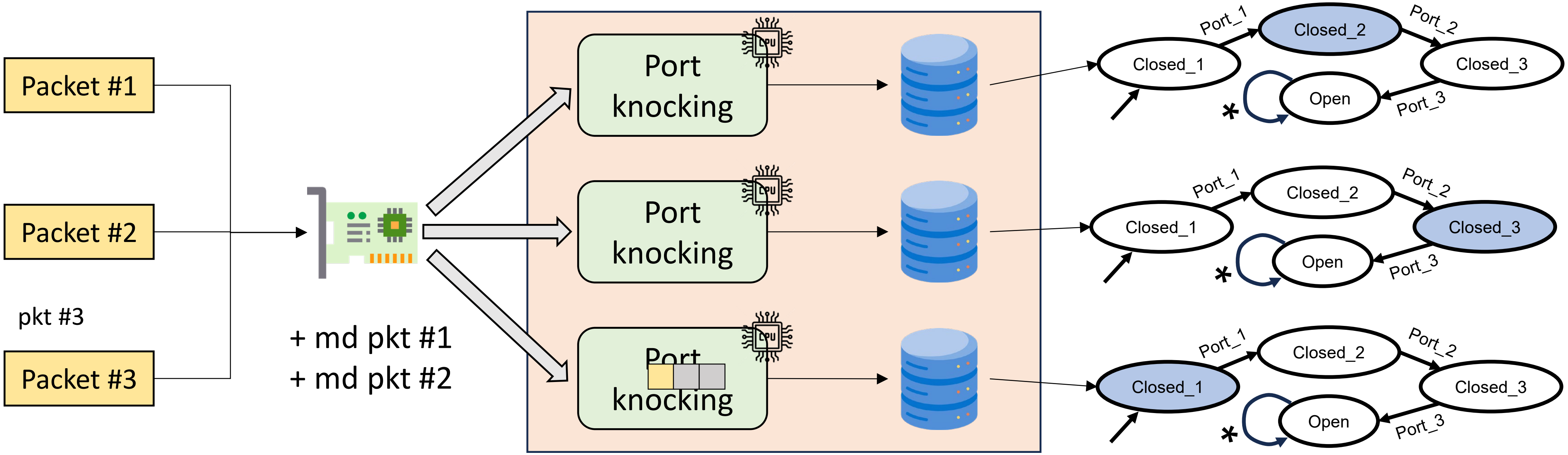
How does this work? – Running Example

- **Metadata**
  - Information of previous packets needed to update the state machine
    - In this example: **l3proto, l4proto, srcIP, dstPort**



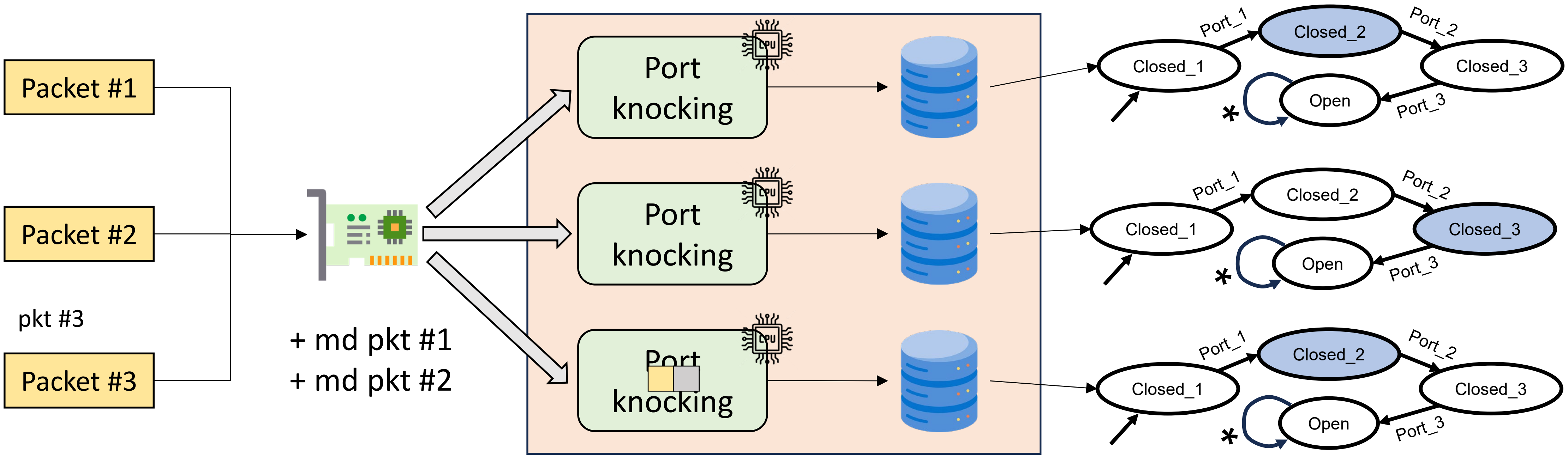
How does this work? – Running Example

- **Metadata**
  - Information of previous packets needed to update the state machine
    - In this example: **l3proto, l4proto, srcIP, dstPort**



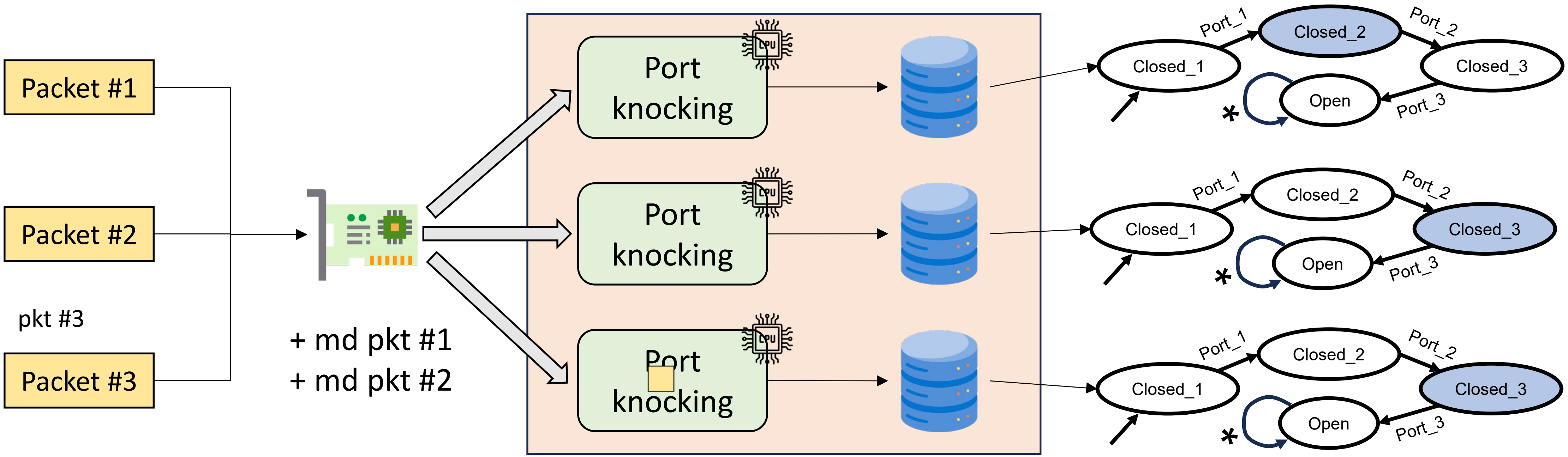
How does this work? – Running Example

- **Metadata**
  - Information of previous packets needed to update the state machine
    - In this example: **l3proto, l4proto, srcIP, dstPort**



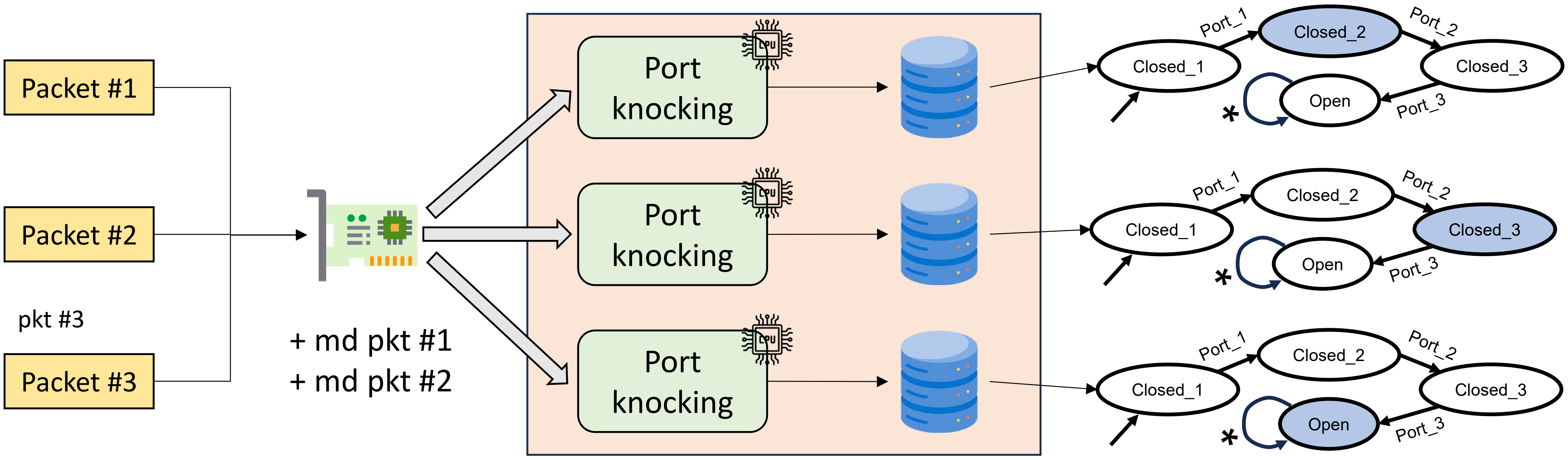
How does this work? – Running Example

- **Metadata**
  - Information of previous packets needed to update the state machine
    - In this example: **l3proto, l4proto, srcIP, dstPort**

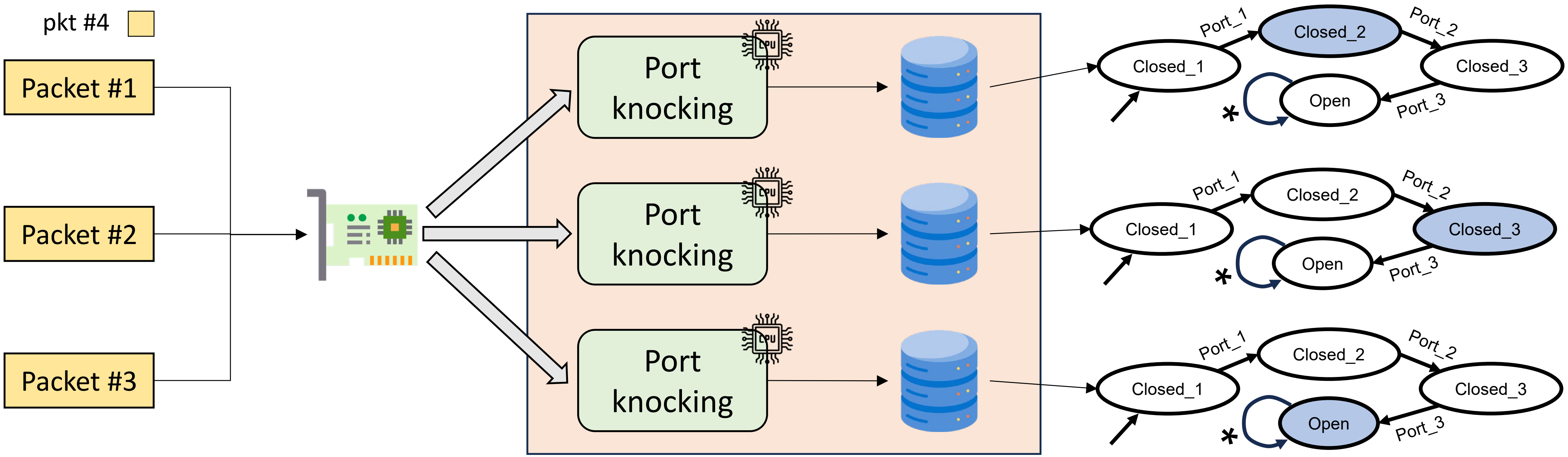


How does this work? – Running Example

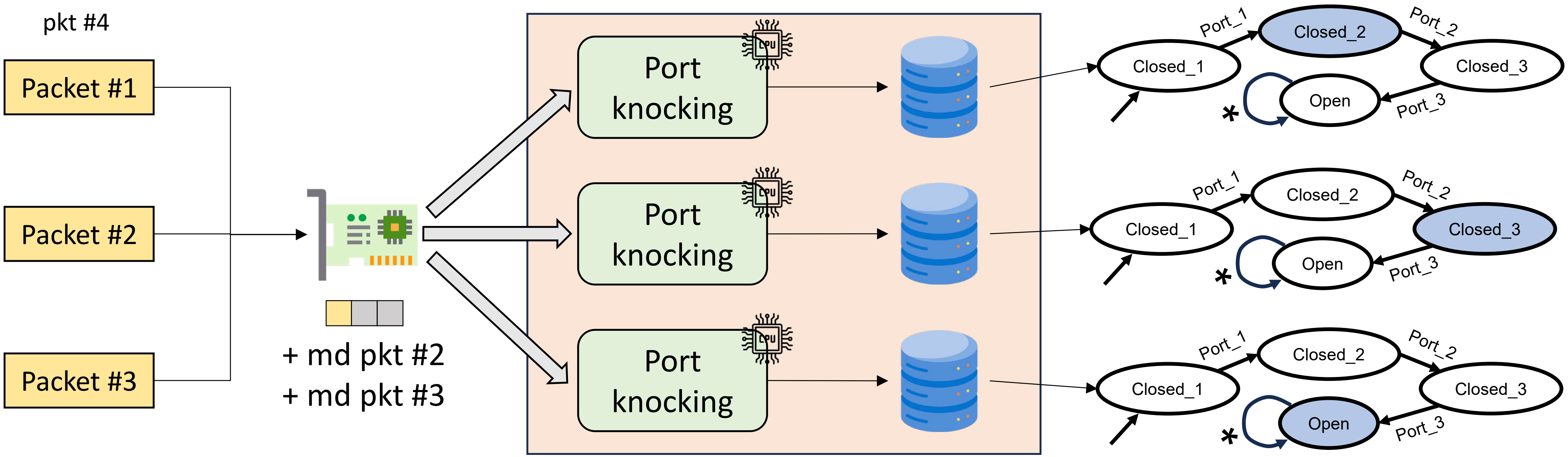
- **Metadata**
  - Information of previous packets needed to update the state machine
    - In this example: **l3proto, l4proto, srcIP, dstPort**



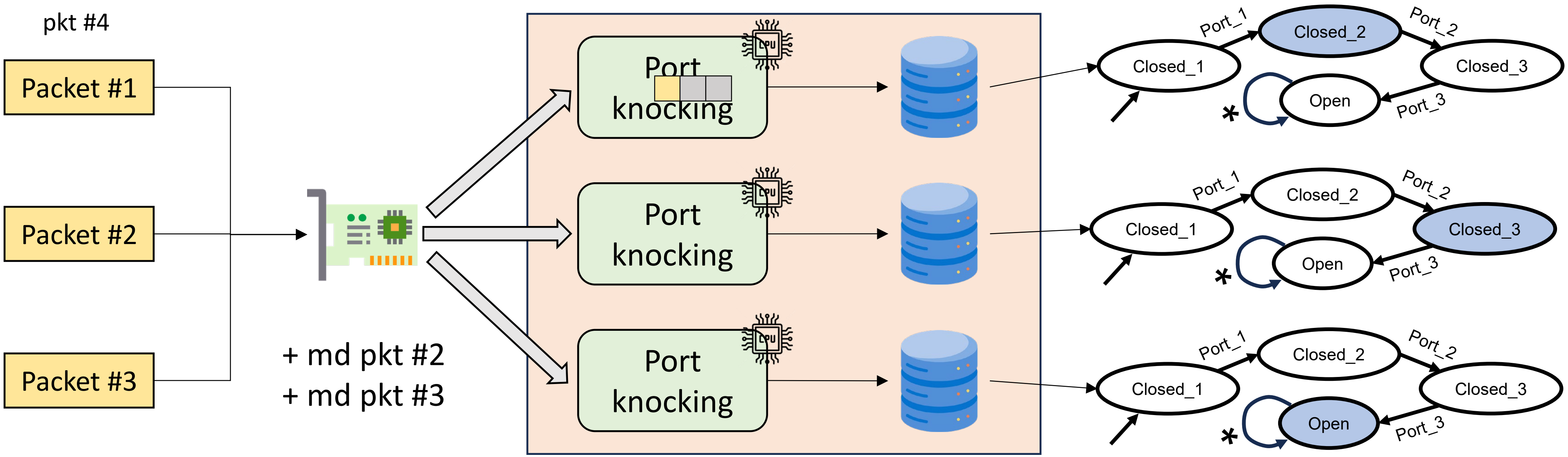
How does this work? – Running Example



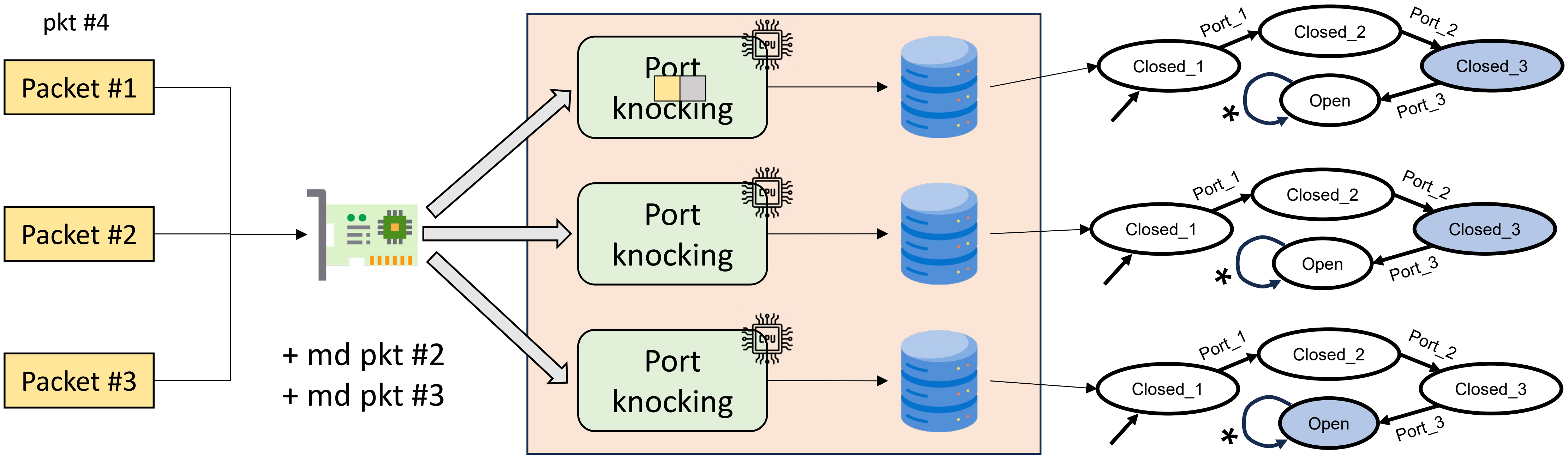
How does this work? – Running Example



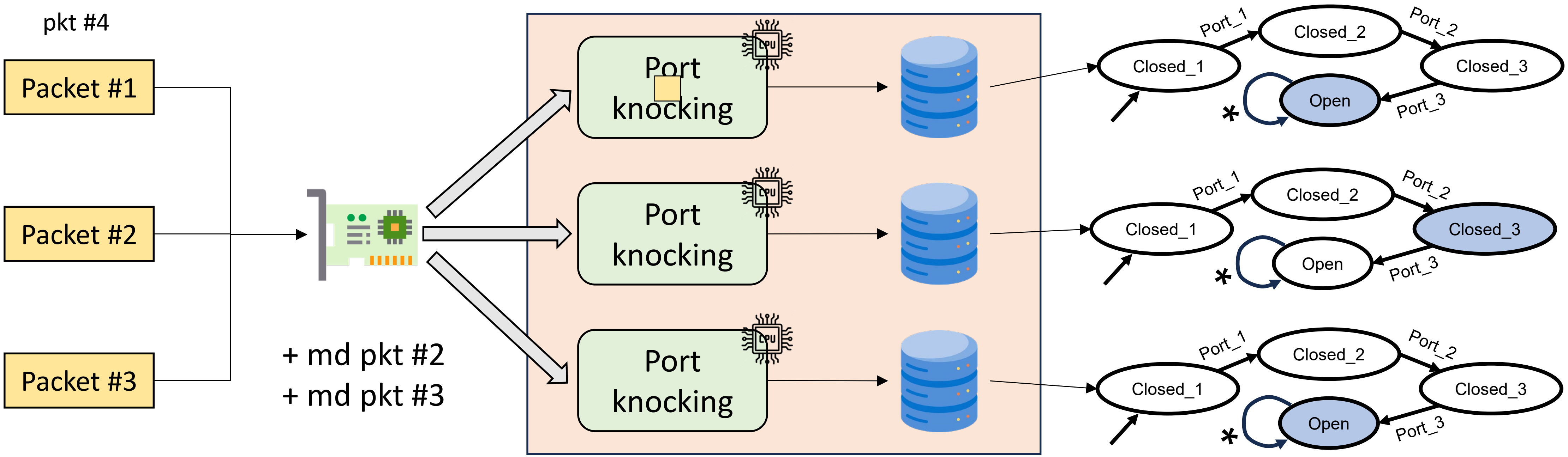
How does this work? – Running Example



How does this work? – Running Example

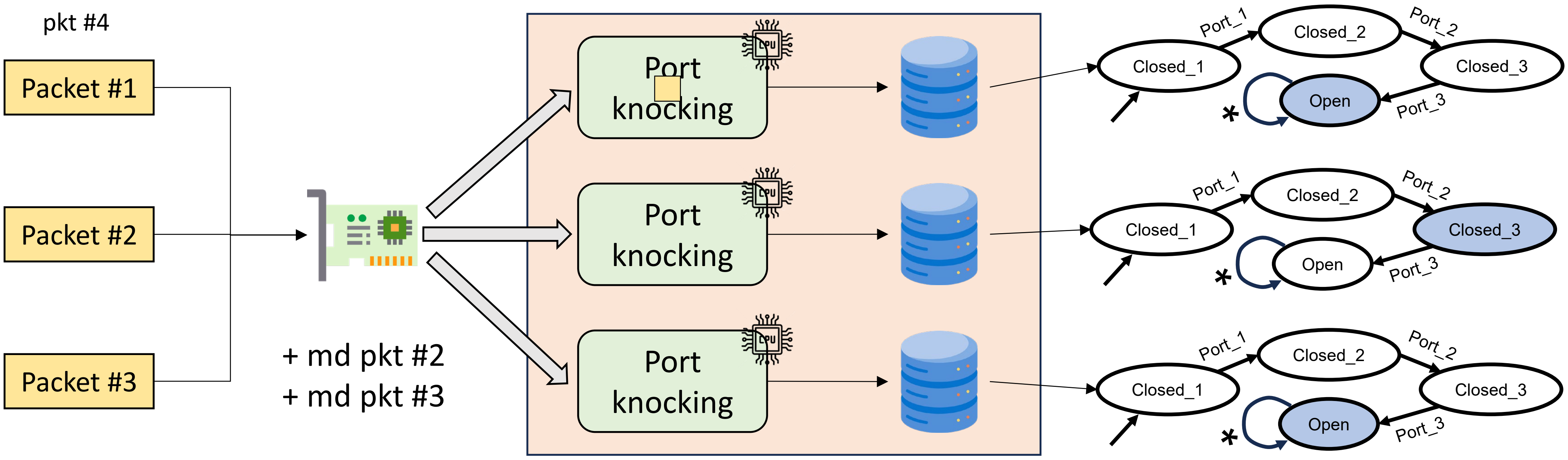


How does this work? – Running Example



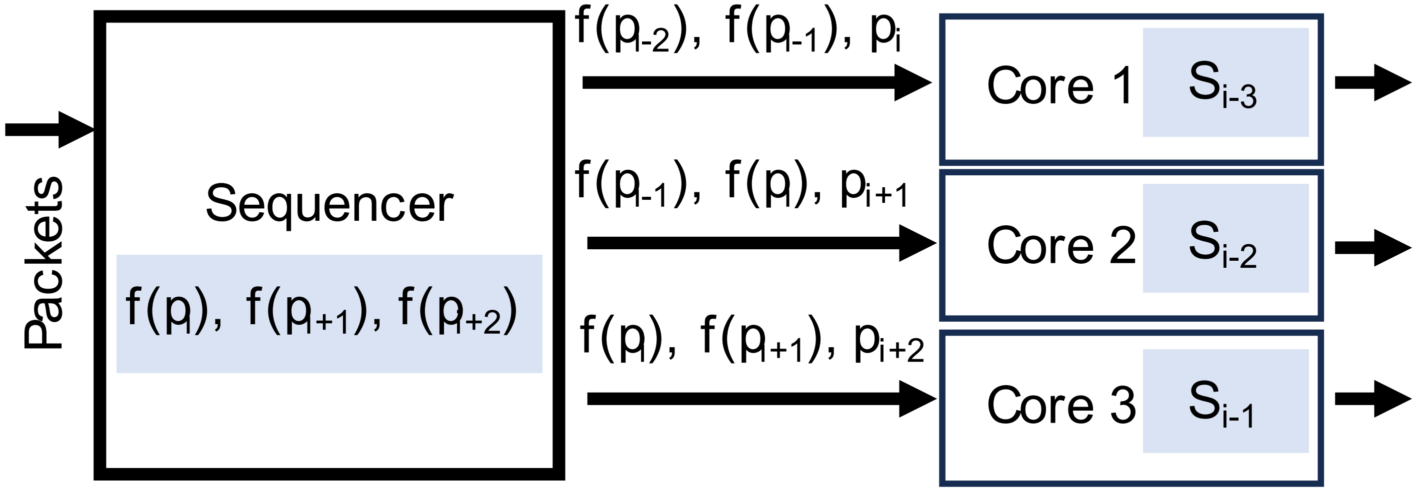
How does this work? – Running Example

- In general, every packet has:
  - $\boldsymbol{Current\,pkt} + \sum \boldsymbol{md}(\boldsymbol{N} - \mathbf{1})$
  - **Where N is the number of cores**



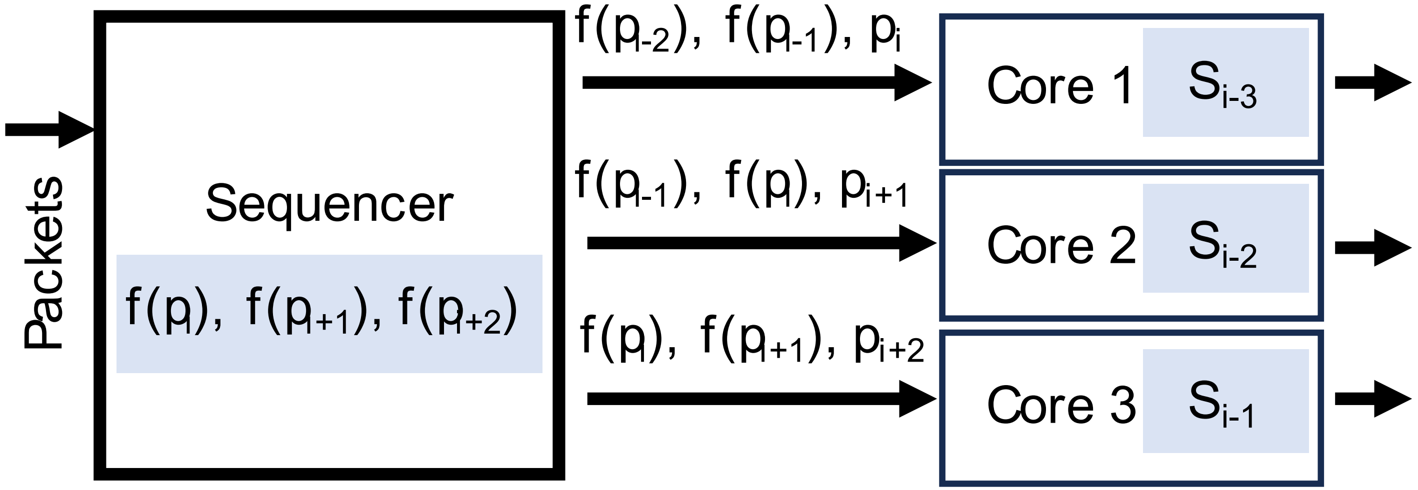
Operationalizing SCR: The Packet History Sequencer

Operationalizing SCR: The Packet History Sequencer



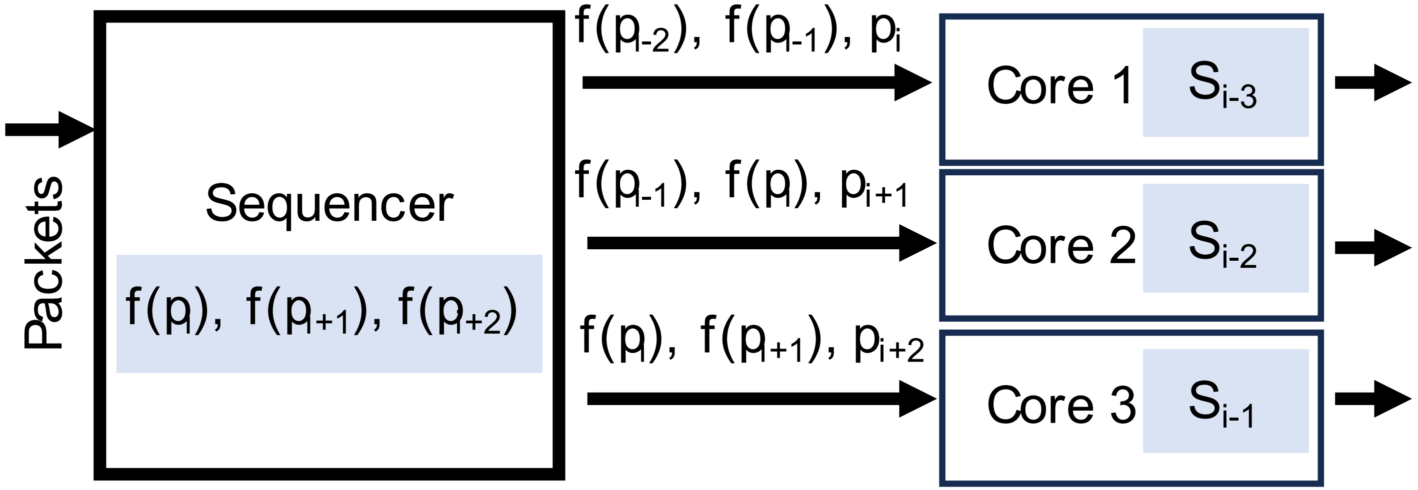
Operationalizing SCR: The Packet History Sequencer

1. Steer packets across cores in round-robin fashion



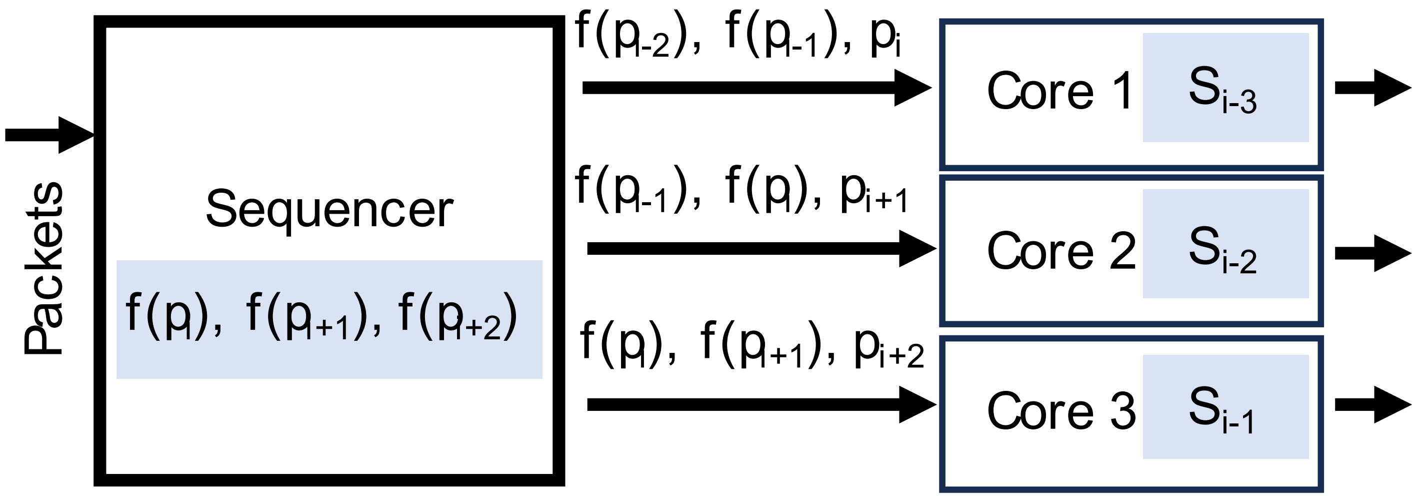
Operationalizing SCR: The Packet History Sequencer

1. Steer packets across cores in round-robin fashion
2. Maintain the most recent packet history across all packets



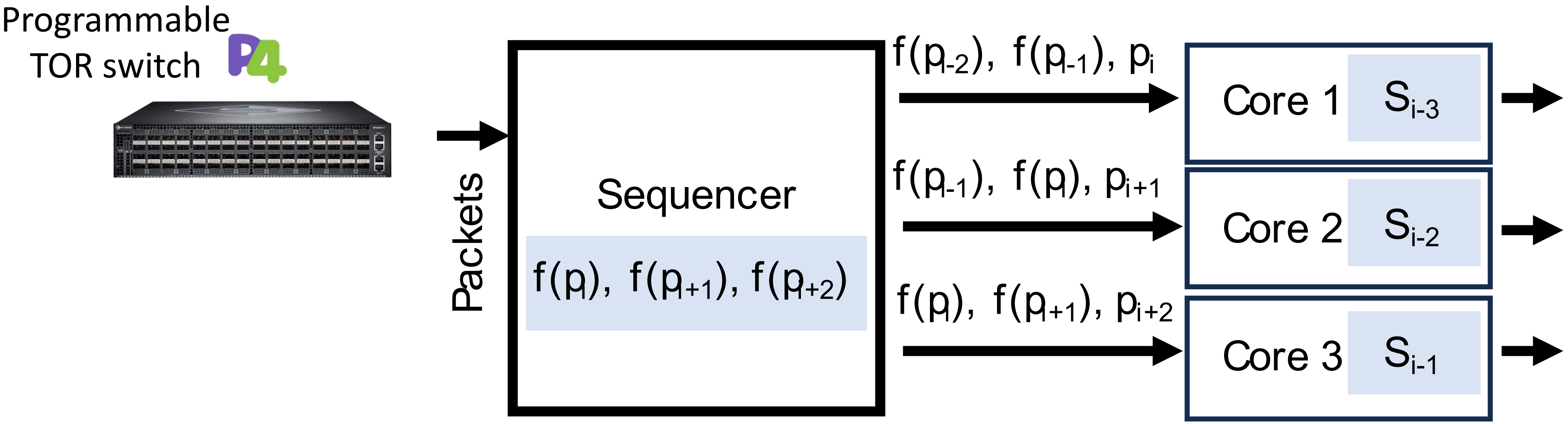
Operationalizing SCR: The Packet History Sequencer

1. Steer packets across cores in round-robin fashion
2. Maintain the most recent packet history across all packets
3. Piggyback the packet history on each packet steered to the cores



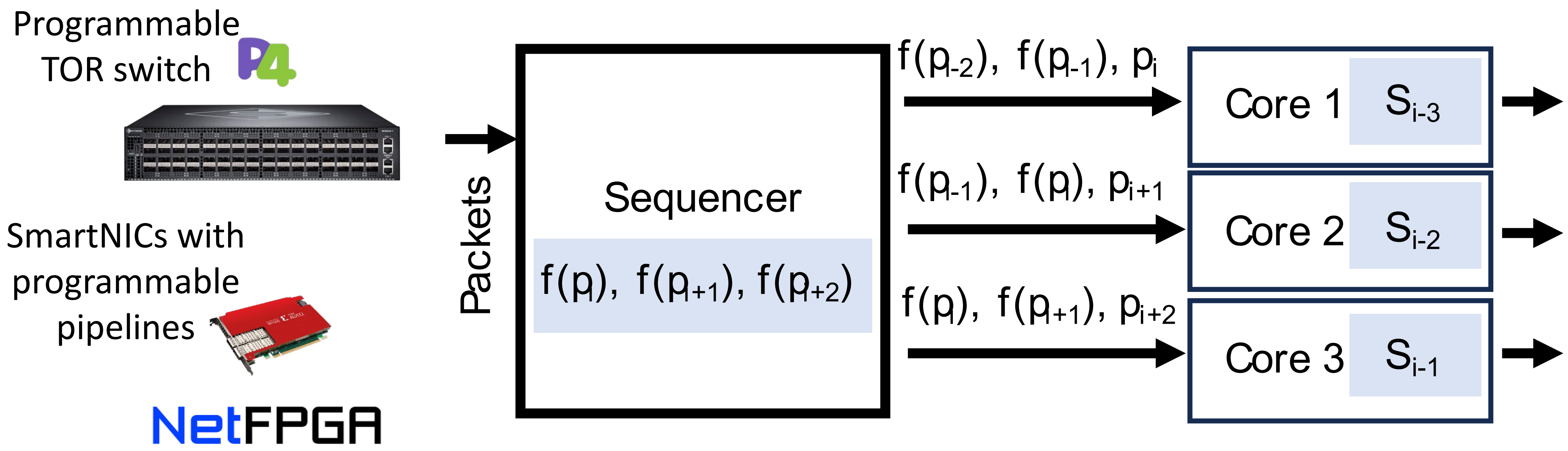
Operationalizing SCR: The Packet History Sequencer

1. Steer packets across cores in round-robin fashion
2. Maintain the most recent packet history across all packets
3. Piggyback the packet history on each packet steered to the cores



Operationalizing SCR: The Packet History Sequencer

1. Steer packets across cores in round-robin fashion
2. Maintain the most recent packet history across all packets
3. Piggyback the packet history on each packet steered to the cores



Operationalizing SCR: SCR-Aware Programming

```
int l3proto, l4proto, srcip, dport, i, j;

for (j = 0; j < NUM_META; j++) {
  i = (index + j) % NUM_META; // ring buffer
  struct meta *pkt = data + i * sizeof(meta);
  l3proto = pkt->l3proto;
  l4proto = pkt->l4proto;
  srcip = pkt->srcip;
  dport = pkt->dport;
  if (l3proto != IPv4 || l4proto != TCP)
    continue; // no state txns or pkt verdicts
  /* Update state for this srcip and dport: */
  /* map_lookup; get_new_state; map_update. */
  /* Note: No pkt verdicts for historic pkts.
}/

pkt_start = data +
  NUM_META * sizeof(struct meta)
  + sizeof(index);
```

```
struct ethhdr* eth = pkt_start; // parse Ethernet
int l3proto = eth->proto; // layer-3 protocol
int off = sizeof(struct ethhdr);

struct iphdr* iph = pkt_start + off;
int l4proto = iph->protocol; // layer-4 protocol
if (l3proto != IPv4 || l4proto != TCP)
    return XDP_DROP; // drop non IPv4/TCP pkts

int srcip = iph->src; // source IP addr
off += sizeof(struct iphdr);
struct tcphdr* tcp = pkt_start + off;
int dport = tcp->dport; // TCP dst port

/* Extract & update firewall state for this src. */
int state = map_lookup(states, srcip);
int new_state = get_new_state(state, dport);
map_update(states, srcip, new_state);

/* Final packet verdict */
if (new_state == OPEN)
    return XDP_TX; // allow traversal

return XDP_DROP; // drop everything else
```

Operationalizing SCR: SCR-Aware Programming

- Define per-core state data structures and per-packet metadata structures.

```
int l3proto, l4proto, srcip, dport, i, j;

for (j = 0; j < NUM_META; j++) {
  i = (index + j) % NUM_META; // ring buffer
  struct meta *pkt = data + i * sizeof(meta);
  l3proto = pkt->l3proto;
  l4proto = pkt->l4proto;
  srcip = pkt->srcip;
  dport = pkt->dport;
  if (l3proto != IPv4 || l4proto != TCP)
    continue; // no state txns or pkt verdicts
  /* Update state for this srcip and dport: */
  /* map_lookup; get_new_state; map_update. */
  /* Note: No pkt verdicts for historic pkts.
}/

pkt_start = data +
  NUM_META * sizeof(struct meta)
  + sizeof(index);
```

```
struct ethhdr* eth = pkt_start; // parse Ethernet
int l3proto = eth->proto; // layer-3 protocol
int off = sizeof(struct ethhdr);

struct iphdr* iph = pkt_start + off;
int l4proto = iph->protocol; // layer-4 protocol
if (l3proto != IPv4 || l4proto != TCP)
    return XDP_DROP; // drop non IPv4/TCP pkts

int srcip = iph->src; // source IP addr
off += sizeof(struct iphdr);
struct tcphdr* tcp = pkt_start + off;
int dport = tcp->dport; // TCP dst port

/* Extract & update firewall state for this src. */
int state = map_lookup(states, srcip);
int new_state = get_new_state(state, dport);
map_update(states, srcip, new_state);

/* Final packet verdict */
if (new_state == OPEN)
    return XDP_TX; // allow traversal

return XDP_DROP; // drop everything else
```

Operationalizing SCR: SCR-Aware Programming

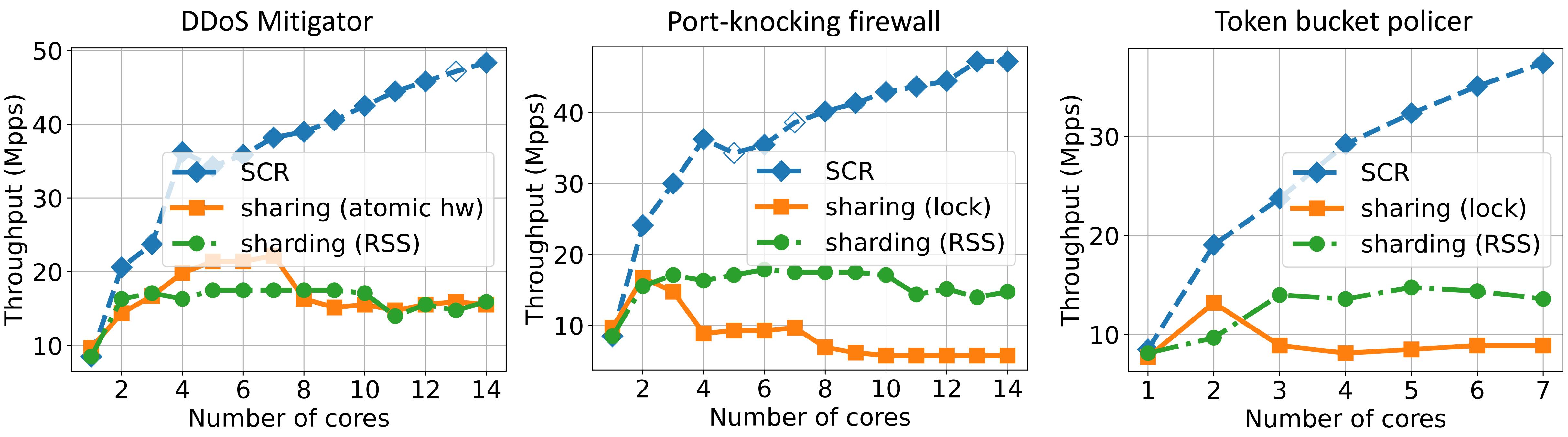
- Define per-core state data structures and per-packet metadata structures.
- Fast-forward the state machine using the packet history.

```
int l3proto, l4proto, srcip, dport, i, j;

for (j = 0; j < NUM_META; j++) {
  i = (index + j) % NUM_META; // ring buffer
  struct meta *pkt = data + i * sizeof(meta);
  l3proto = pkt->l3proto;
  l4proto = pkt->l4proto;
  srcip = pkt->srcip;
  dport = pkt->dport;
  if (l3proto != IPv4 || l4proto != TCP)
    continue; // no state txns or pkt verdicts
  /* Update state for this srcip and dport: */
  /* map_lookup; get_new_state; map_update. */
  /* Note: No pkt verdicts for historic pkts.
}/

pkt_start = data +
  NUM_META * sizeof(struct meta)
  + sizeof(index);
```

```
struct ethhdr* eth = pkt_start; // parse Ethernet
int l3proto = eth->proto; // layer-3 protocol
int off = sizeof(struct ethhdr);

struct iphdr* iph = pkt_start + off;
int l4proto = iph->protocol; // layer-4 protocol
if (l3proto != IPv4 || l4proto != TCP)
    return XDP_DROP; // drop non IPv4/TCP pkts

int srcip = iph->src; // source IP addr
off += sizeof(struct iphdr);
struct tcphdr* tcp = pkt_start + off;
int dport = tcp->dport; // TCP dst port

/* Extract & update firewall state for this src. */
int state = map_lookup(states, srcip);
int new_state = get_new_state(state, dport);
map_update(states, srcip, new_state);

/* Final packet verdict */
if (new_state == OPEN)
    return XDP_TX; // allow traversal

return XDP_DROP; // drop everything else
```

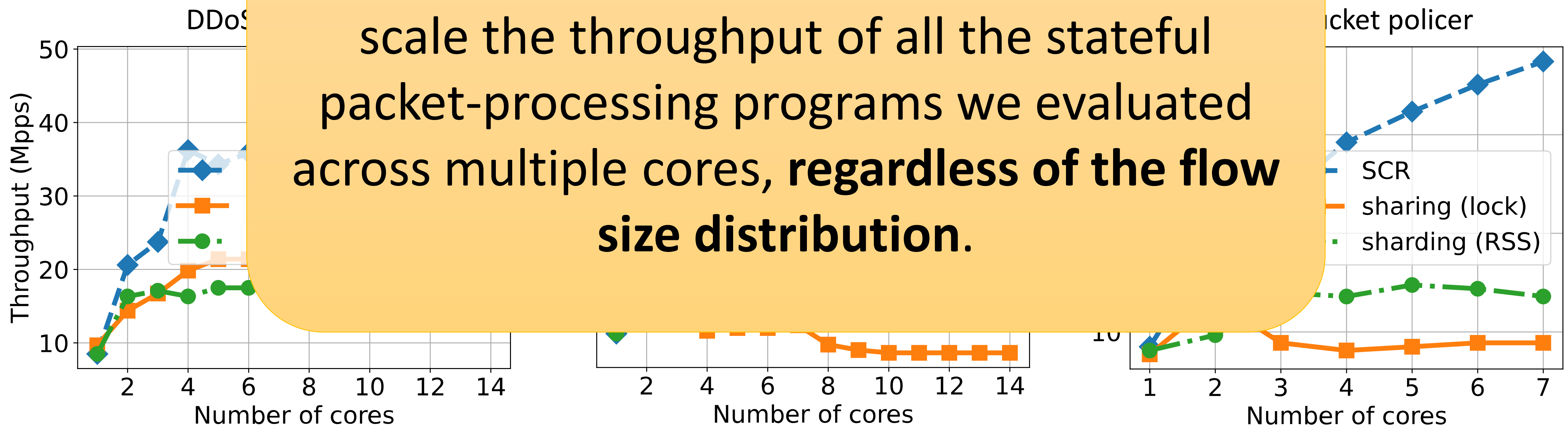
Experimental Results - Throughput

- We tested SCR on a set of eBPF/XDP applications, using realistic traffic traces (CAIDA, University DC)



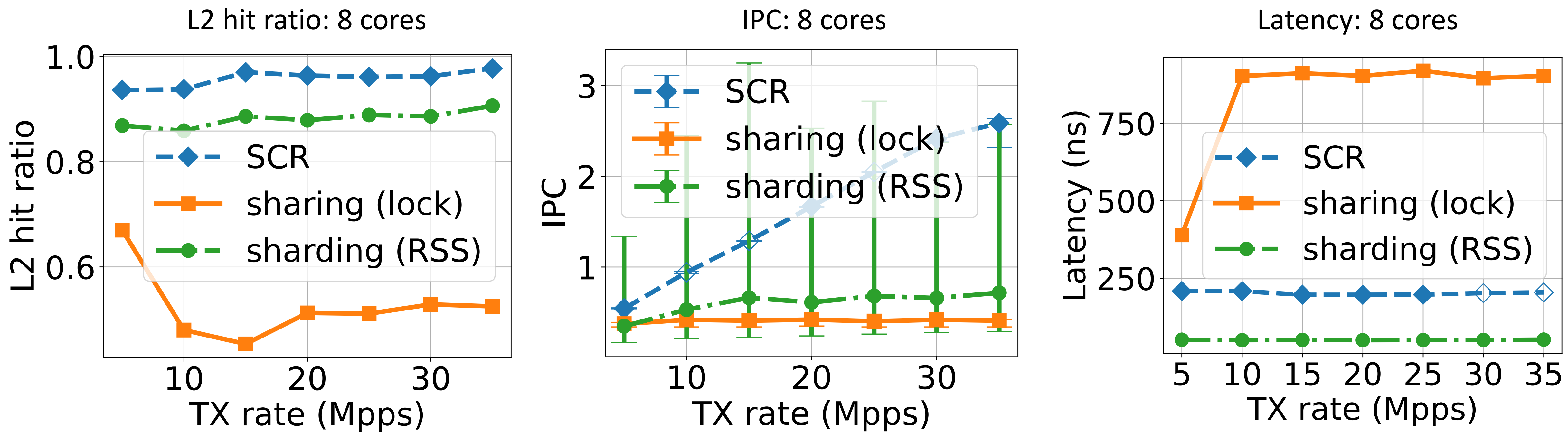
Experimental Results - Throughput

- We tested SCR on a set of eBPF/XDP applications, using realistic traffic traces (C

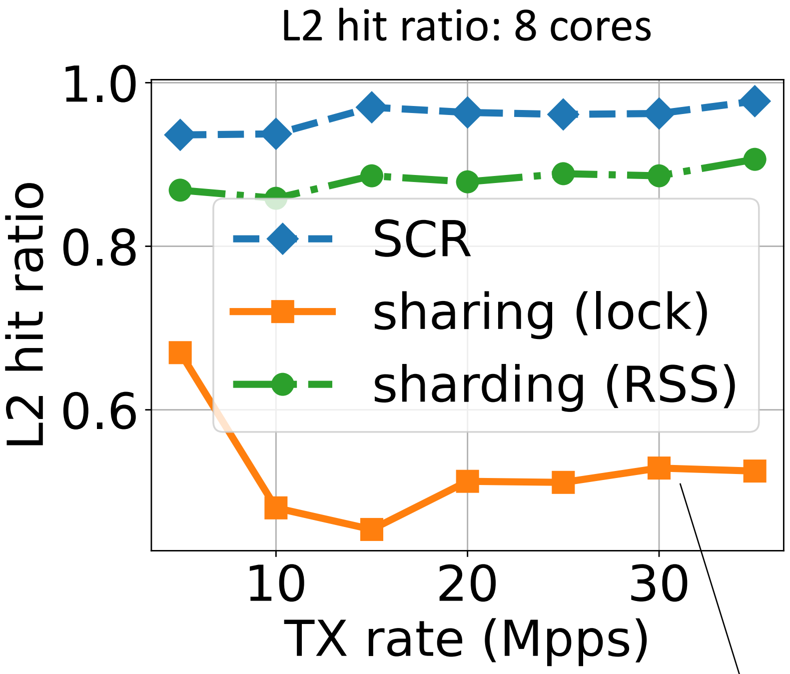


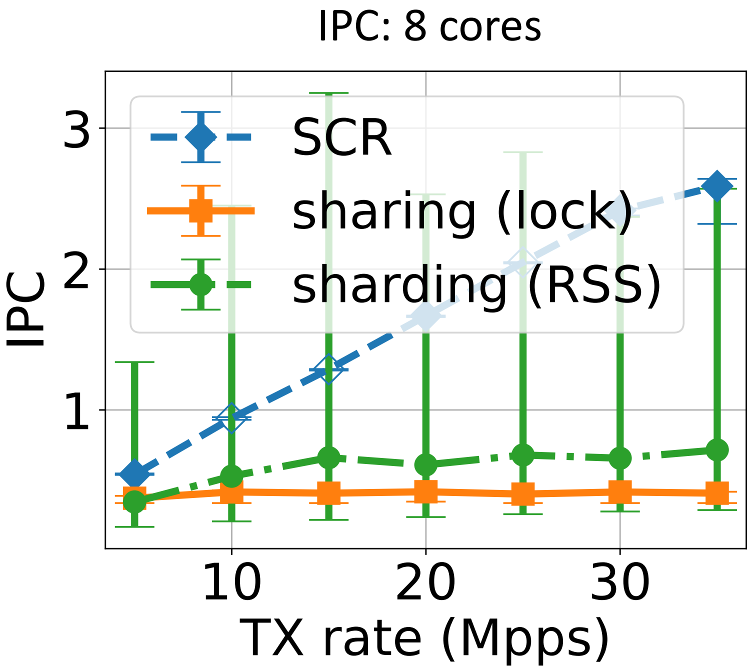
SCR is the only scaling technique that can scale the throughput of all the stateful packet-processing programs we evaluated across multiple cores, **regardless of the flow size distribution**.

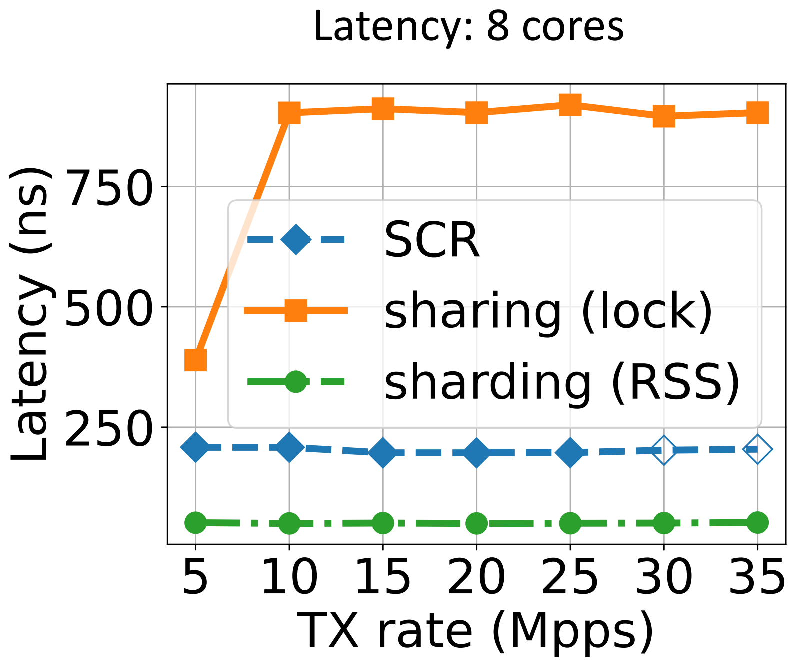
Why does SCR scales better than the other techniques?



Why does SCR scales better than the other techniques?

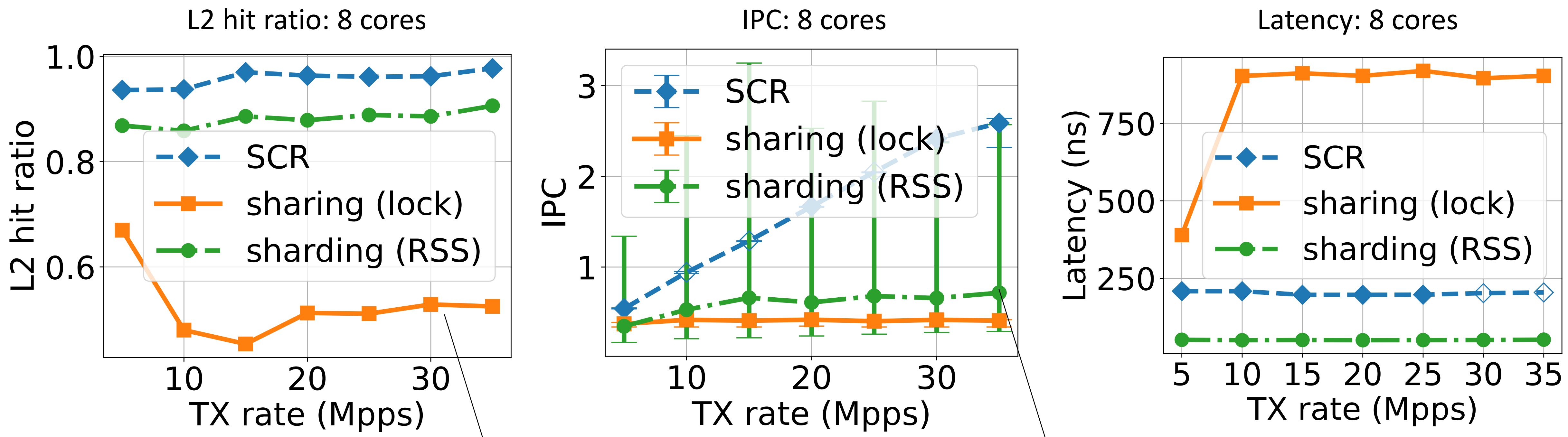






Lock and cache line contention across cores

Why does SCR scales better than the other techniques?

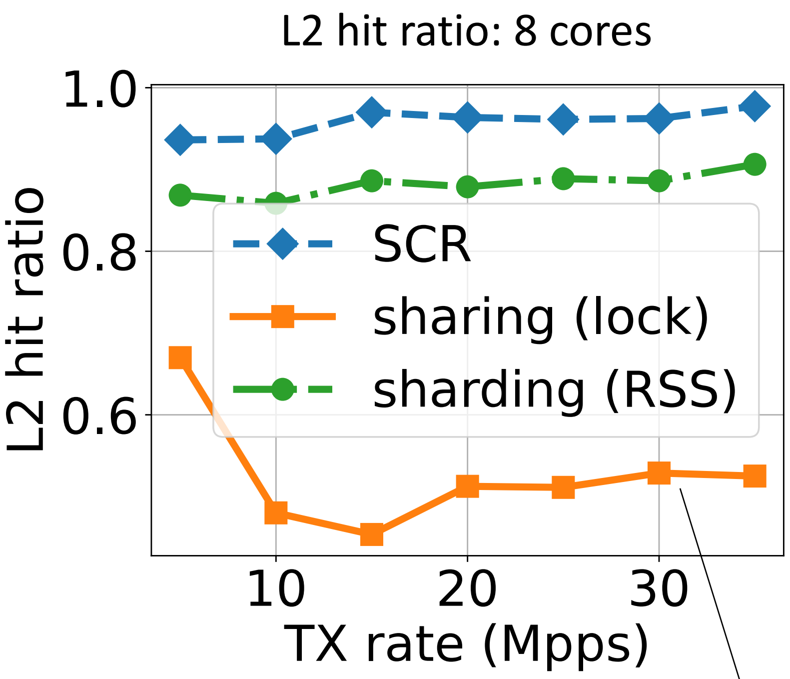


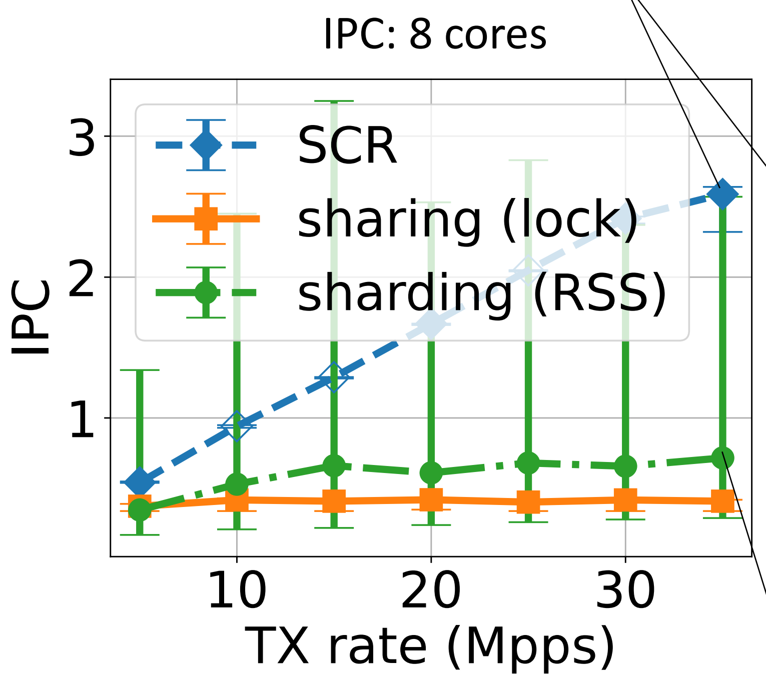
Lock and cache line contention across cores

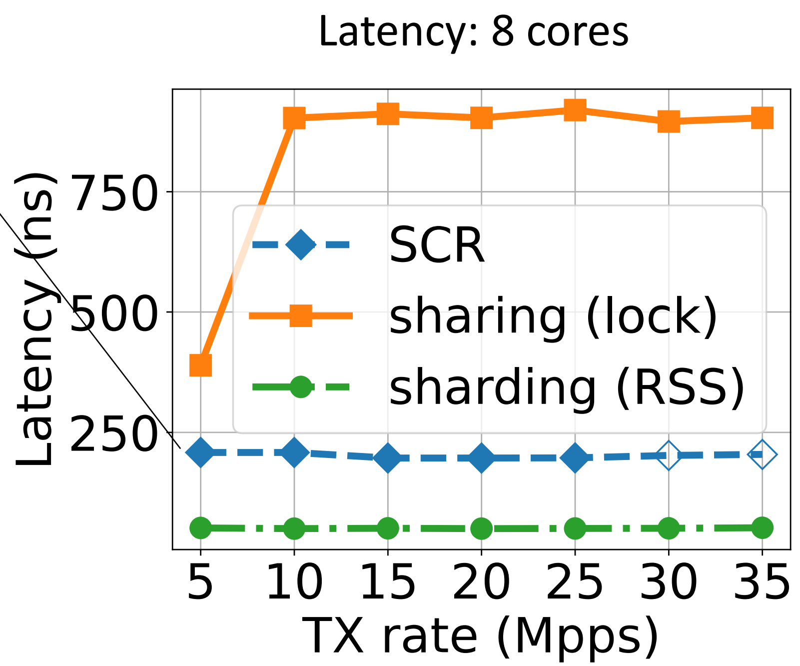
High variation, indicating imbalance of CPU work

Why does SCR scales better than the other techniques?

Consistent high IPC with more cores, because of metadata history processing





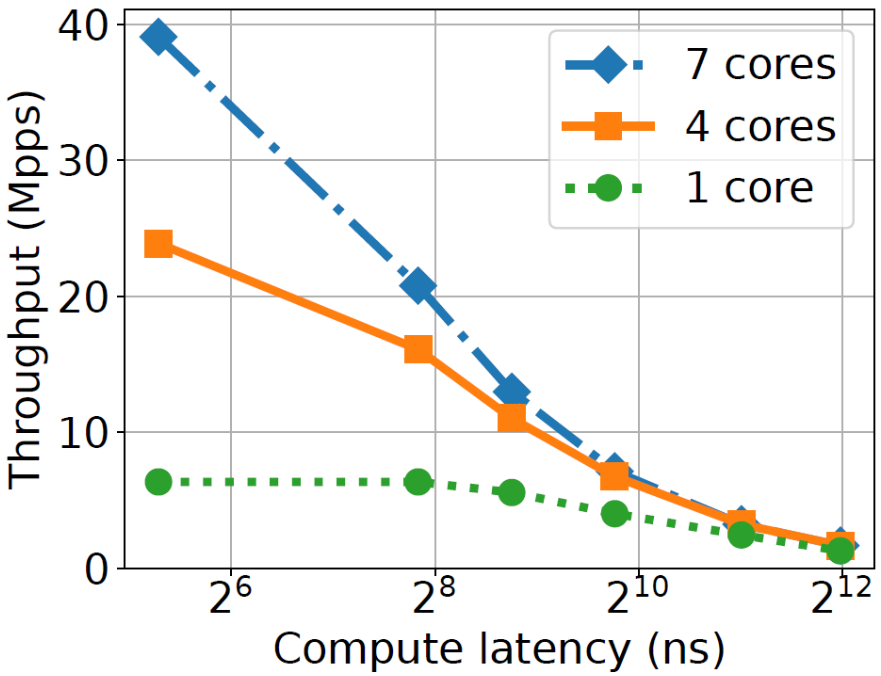


Lock and cache line contention across cores

High variation, indicating imbalance of CPU work

All that glitters is not gold - SCR limitations

1. If the compute latency increases in comparison to the dispatch latency, the effectiveness of SCR's multi-core scaling reduces
   - Every core has to do "more work" to catch up with the state



All that glisters is not gold! SCR limitations

token bucket policer

All that glisters is not gold! SCR limitations

2. SCR's attachment of histories to packets incurs non-negligible overheads

token bucket policer

All that glisters is not gold! SCR limitations

2. SCR's attachment of histories to packets incurs non-negligible overheads

token bucket policer

All that glisters is not gold! SCR limitations

2. SCR's attachment of histories to packets incurs non-negligible overheads

token bucket policer

- Can **increase L3 cache pressure** due to higher DDIO cache occupancy

All that glisters is not gold! SCR limitations

2. SCR's attachment of histories to packets incurs non-negligible overheads

token bucket policer

- Can **increase L3 cache pressure** due to higher DDIO cache occupancy
- **Increases PCIe transactions** and bandwidth

All that glisters is not gold! SCR limitations

2. SCR's attachment of histories to packets incurs non-negligible overheads

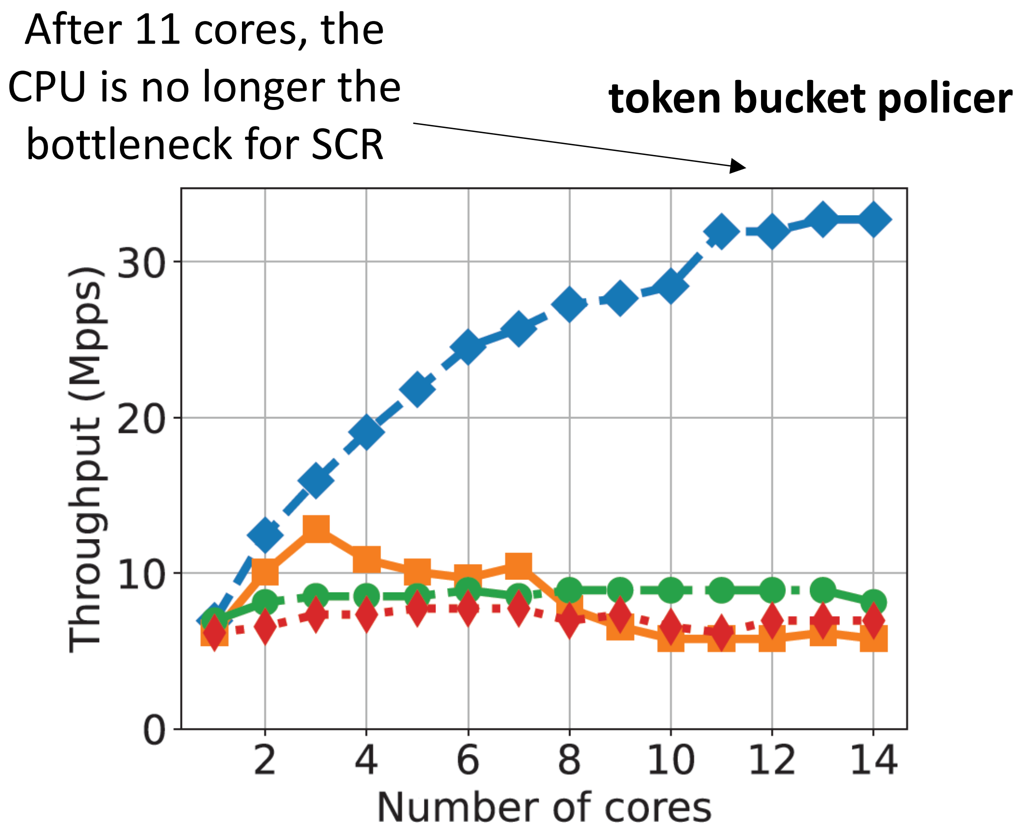
token bucket policer

- Can **increase L3 cache pressure** due to higher DDIO cache occupancy
- **Increases PCIe transactions** and bandwidth
- When packet history is appended outside the NIC (e.g., TOR switch), SCR may **saturate the NIC earlier than other approaches**

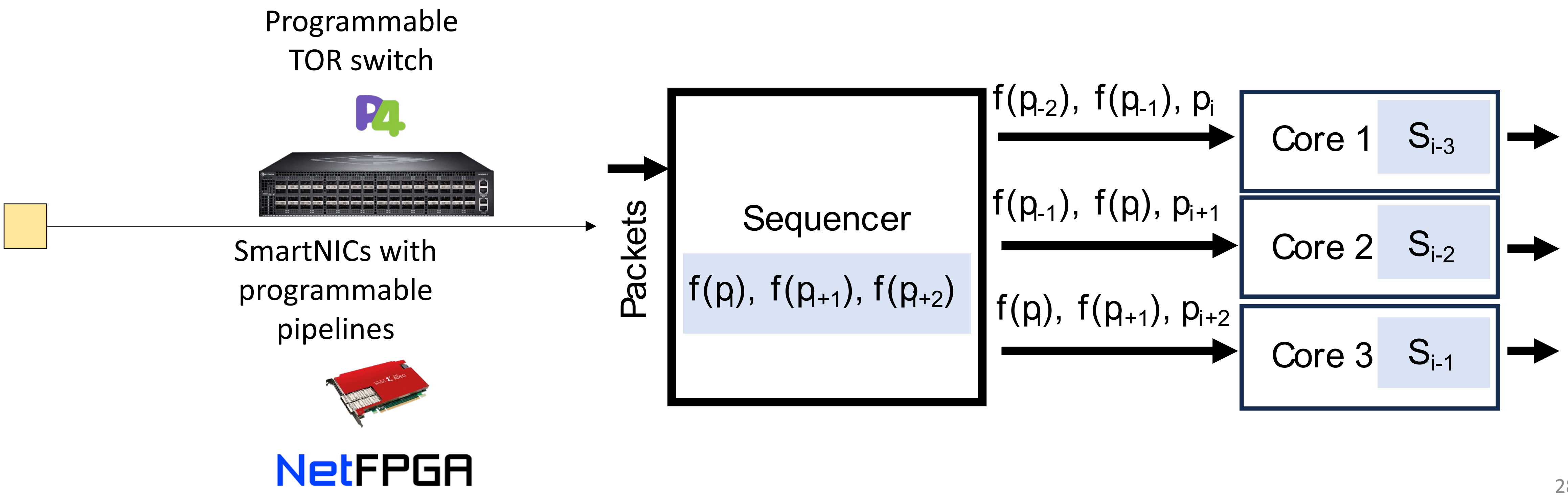
All that glisters is not gold! SCR limitations

2. SCR's attachment of histories to packets incurs non-negligible overheads

- Can **increase L3 cache pressure** due to higher DDIO cache occupancy
- **Increases PCIe transactions** and bandwidth
- When packet history is appended outside the NIC (e.g., TOR switch), SCR may **saturate the NIC earlier than other approaches**

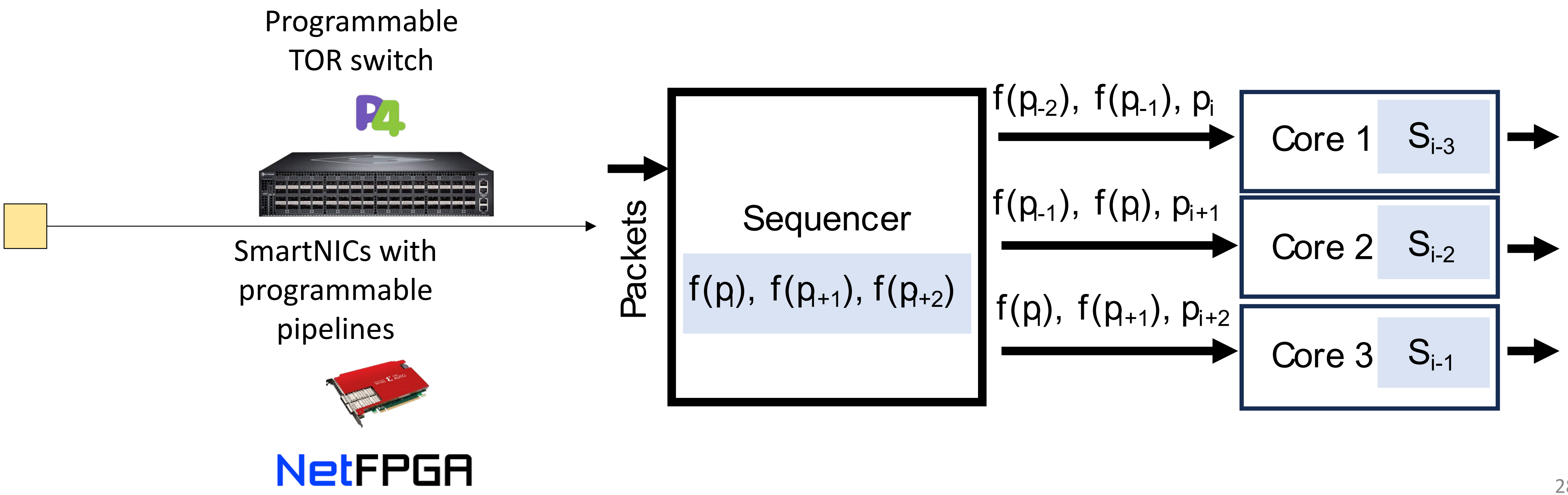


Handling Packet Loss



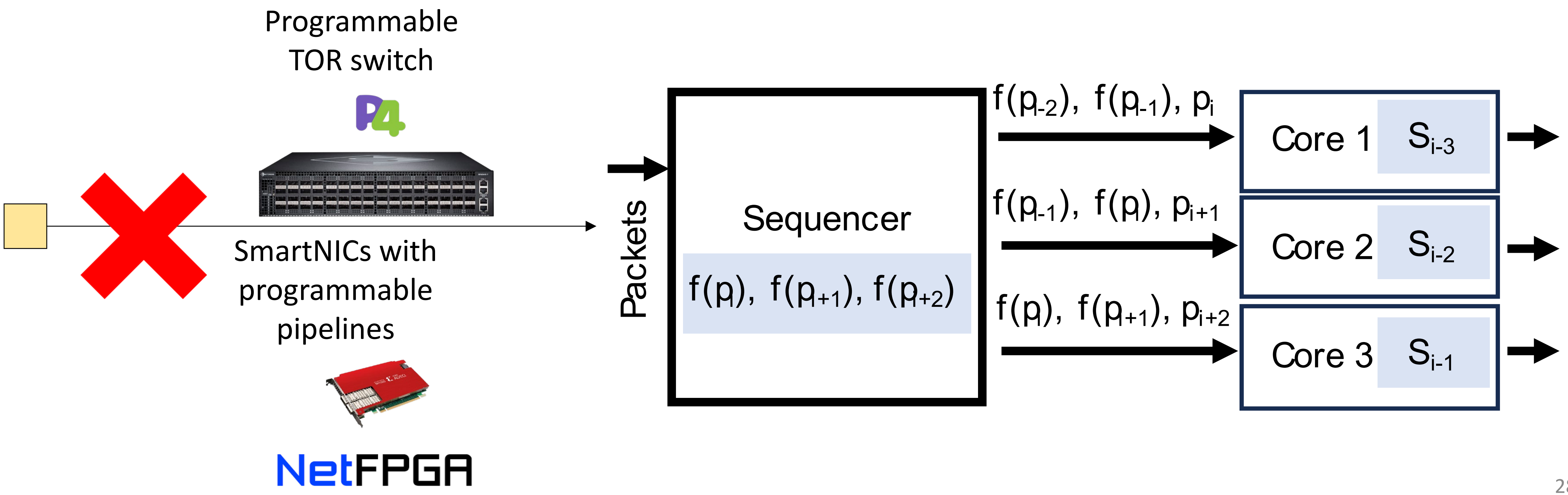
Handling Packet Loss

- Packets can be lost either



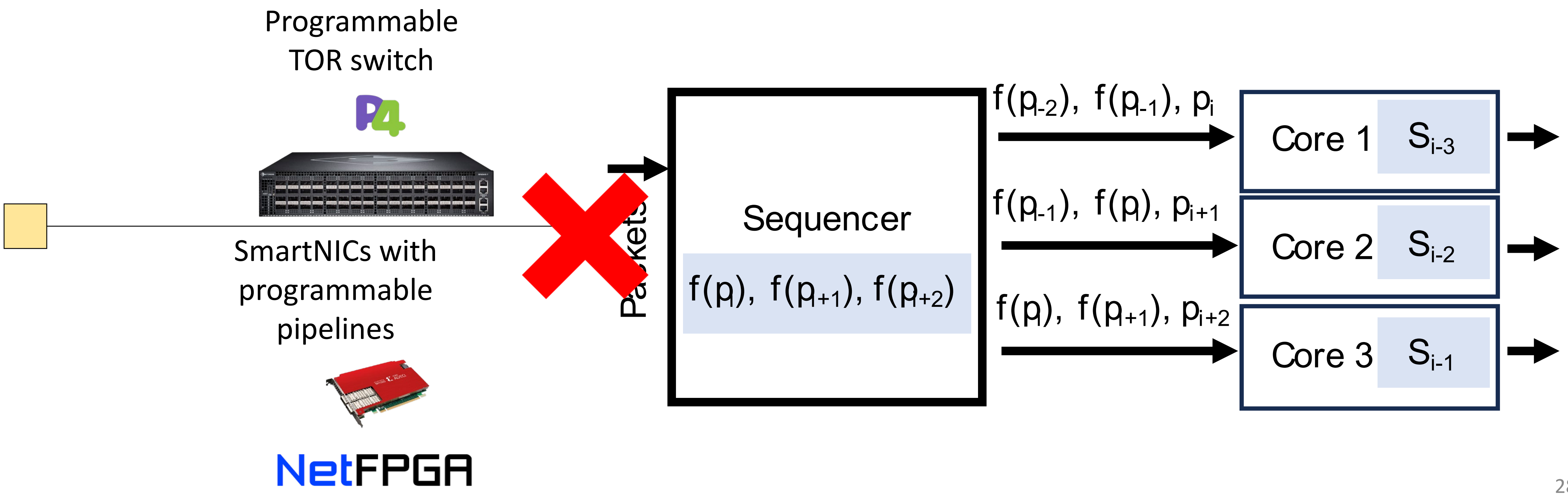
Handling Packet Loss

- Packets can be lost either
  - (1) **prior** to the sequencer



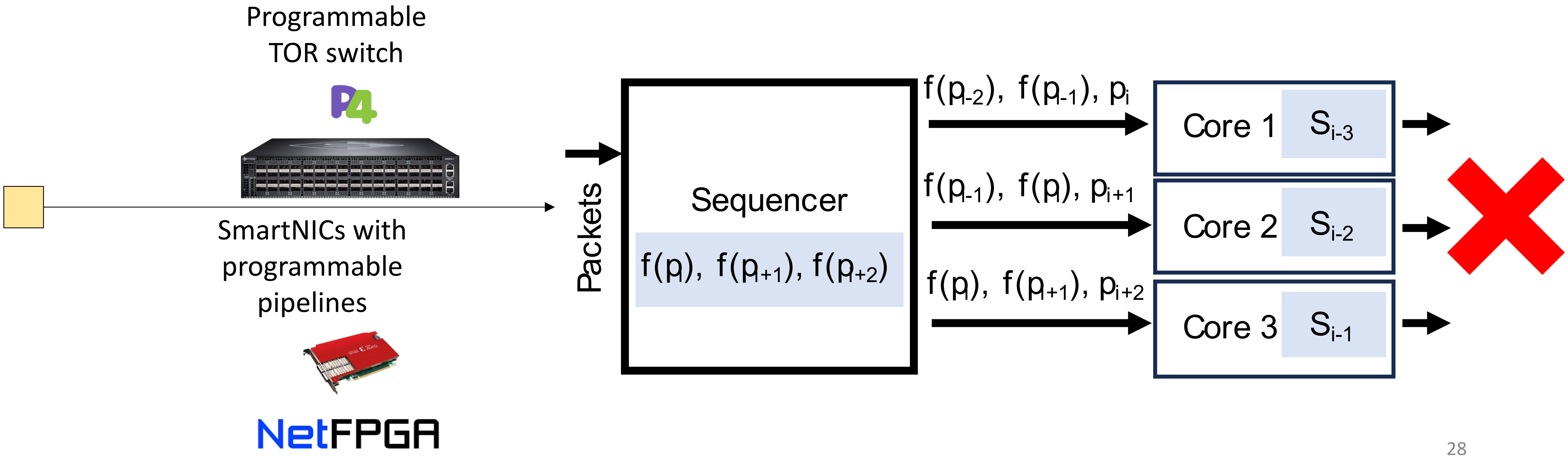
Handling Packet Loss

- Packets can be lost either
  - (1) **prior** to the sequencer
  - (2) **after** the sequencer but prior to processing at a CPU core

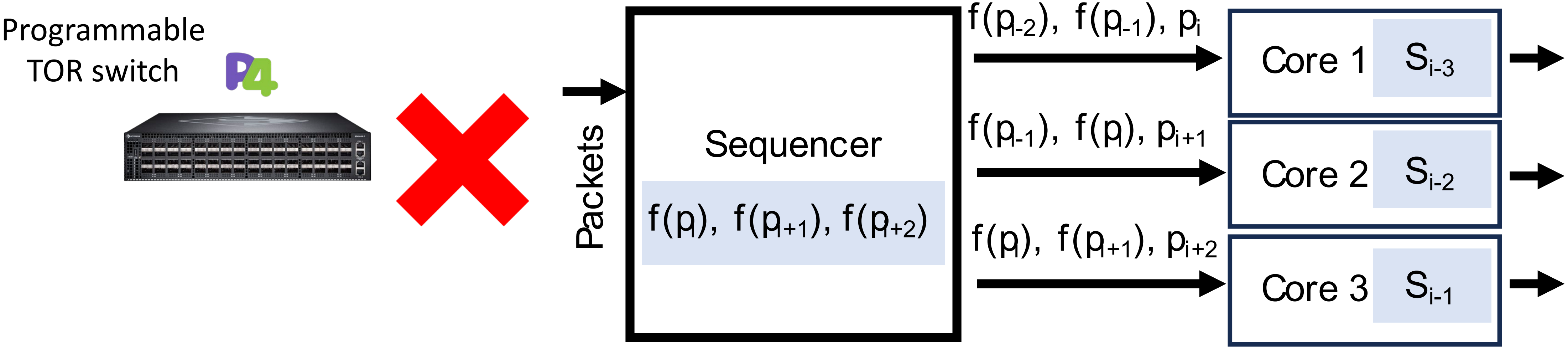


Handling Packet Loss

- Packets can be lost either
  - (1) **prior** to the sequencer
  - (2) **after** the sequencer but prior to processing at a CPU core
  - (3) after processing at a core.

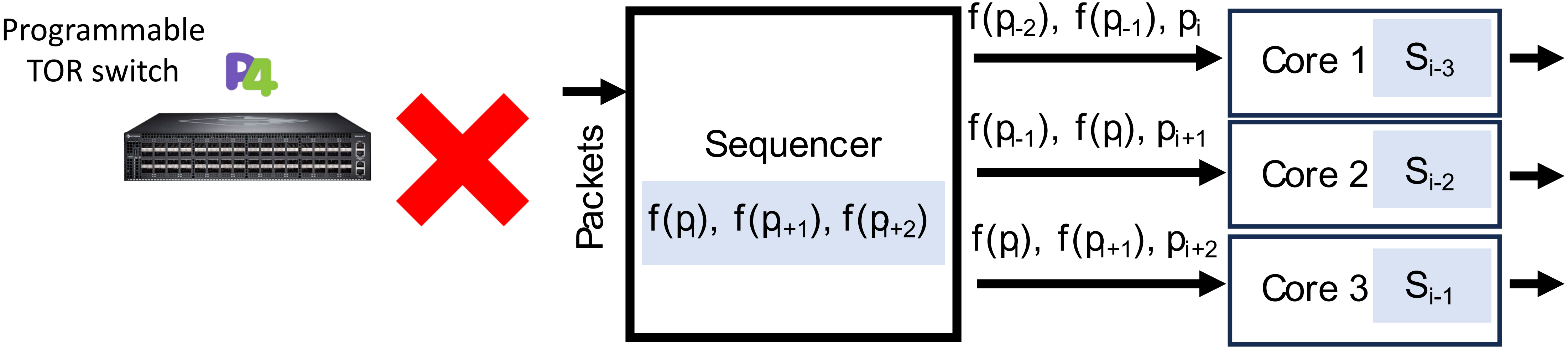


Handling Packet Loss



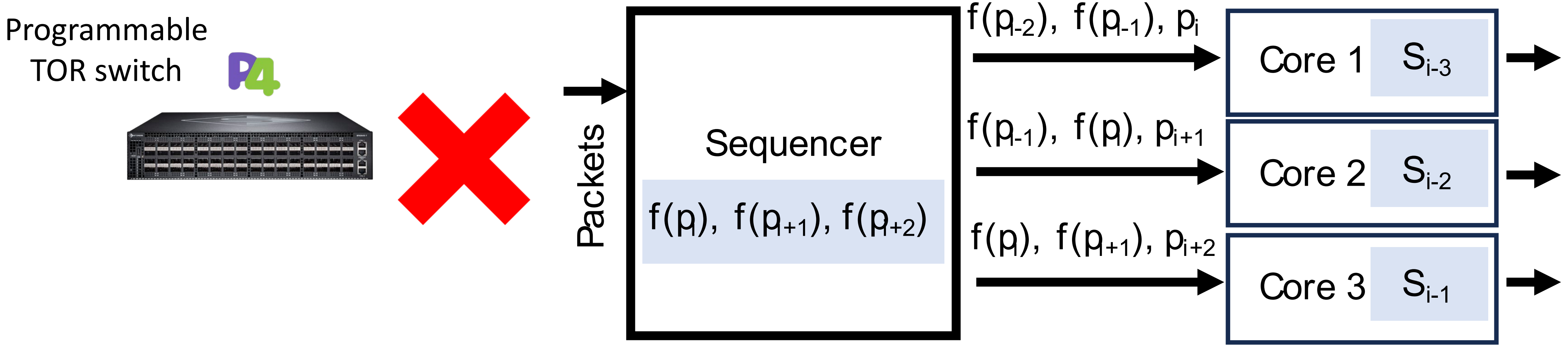
Handling Packet Loss

- **The only one that represents a problem is (2)**



Handling Packet Loss

- **The only one that represents a problem is (2)**
- Only in the case where the sequence is deployed on a top-of-the-rack switch
  - We can run link-level flow control mechanism like PFC to prevent packet loss between the switch and server cores.



Handling Packet Loss - Solution

Handling Packet Loss - Solution

- A core can either read the full flow state from a more up-to-date core

Handling Packet Loss - Solution

- A core can either read the full flow state from a more up-to-date core
- …or it can read the packet history from either the sequencer or a log written by a more up-to-date core

Handling Packet Loss - Solution

- A core can either read the full flow state from a more up-to-date core
- …or it can read the packet history from either the sequencer or a log written by a more up-to-date core

- To achieve this, we:
  1. Have the sequence attach an **incrementing sequence number** to each packet
  2. Use a **per-core, lockless, single-writer multiple-reader log**
  3. Introduce an algorithm to **catch up the flow state** on each core upon detection of loss

Conclusions

Conclusions

- **State Compute Replication (SCR)** is a principle that enables **scaling stateful** packet processing programs across multiple cores

Conclusions

- **State Compute Replication (SCR)** is a principle that enables **scaling stateful** packet processing programs across multiple cores
- It leverages a packet history sequencer to collect the history of the packets and propagate it to CPU cores
  - Can run on a **NIC** or **TOR** switch

Conclusions

- **State Compute Replication (SCR)** is a principle that enables **scaling stateful** packet processing programs across multiple cores
- It leverages a packet history sequencer to collect the history of the packets and propagate it to CPU cores
  - Can run on a **NIC** or **TOR** switch
- Applications using SCR need to be modified to replicate the program state and keep private copies per core
  - A **compiler** can do it automatically!

Conclusions

- **State Compute Replication (SCR)** is a principle that enables **scaling stateful** packet processing programs across multiple cores
- It leverages a packet history sequencer to collect the history of the packets and propagate it to CPU cores
  - Can run on a **NIC** or **TOR** switch
- Applications using SCR need to be modified to replicate the program state and keep private copies per core
  - A **compiler** can do it automatically!
- Our experiment show that SCR can scale total packet processing throughput **linearly** with cores, **deterministically** and **independent of flow size distribution**