Breaking Limits: Terabit Speeds on a single CPU server

Marco Chiesa

KTH Royal Institute of Technology

Breaking Limits: Terabit Speeds on a single CPU server

Marco Chiesa

KTH Royal Institute of Technology

Joint work with

Tommaso Caiazzi

Mariano Scazzariello

Hamid Ghasemirahni

Alireza Farshin

Dejan Kostić

Gerald Q. Maguire Jr

Internet Traffic is Growing in Speed and Size

Massive amount of data to be processed at **cloud/edge networks**

External internet-facing

- users, sensors, services
- **terabits** per second (Tbps)
- **millions** of connections per second

Internal machine-to-machine

– big-data, AI/ML, etc

– **thousands** of servers

– **petabits** per second (Pbps) of traffic

Processing massive amounts of traffic at **high rates** is **hard**!

Processing Massive Amounts of Traffic at High Rates is Hard!

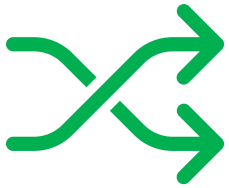
Network Functions Virtualization is an **essential** architectural paradigm of today's data centers

Processing Massive Amounts of Traffic at High Rates is Hard!

Network Functions Virtualization is an **essential** architectural paradigm of today's data centers

NFs are deployed on commodity hardware rather than specialized one

Cost-Effective



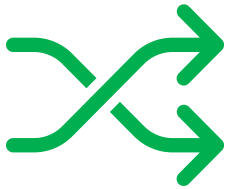
Flexible

Scalable

Processing Massive Amounts of Traffic at High Rates is Hard!

NFs are deployed on commodity hardware rather than specialized one

Cost-Effective



Flexible

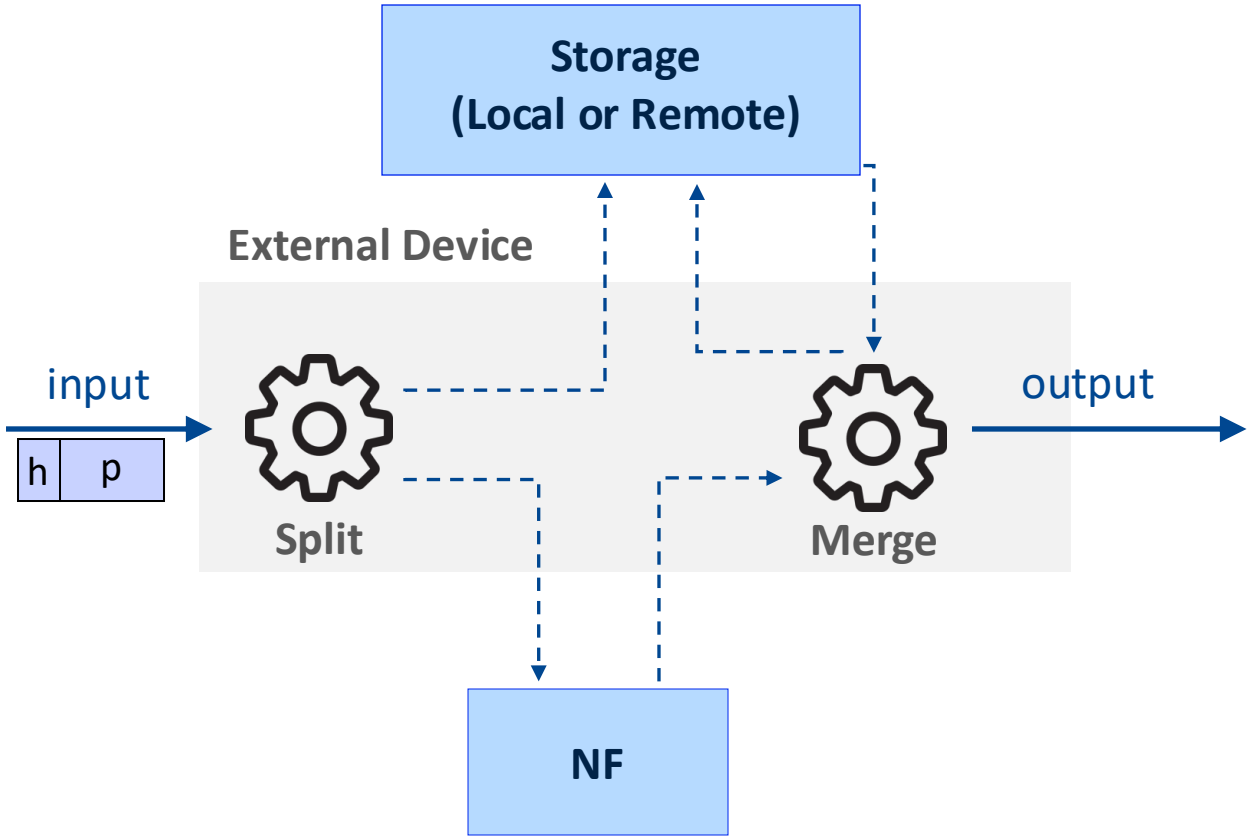
Scalable

Lower performance compared to specialized hardware

Minimize CPU Cycles per Packet

State-of-the-art **packet-splitting** solutions **increase** shallow NFs throughput

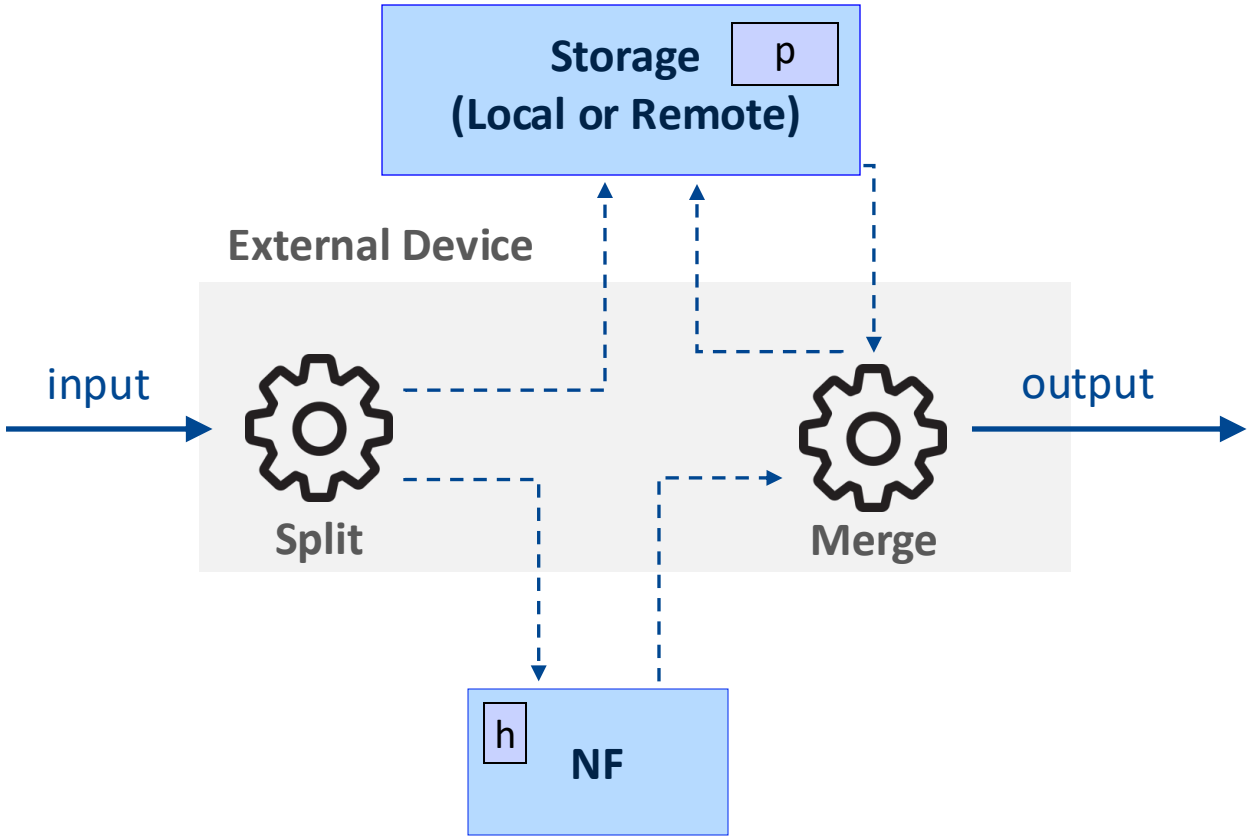
e.g., Ribosome, nicmem



Minimize CPU Cycles per Packet

State-of-the-art **packet-splitting** solutions **increase** shallow NFs throughput

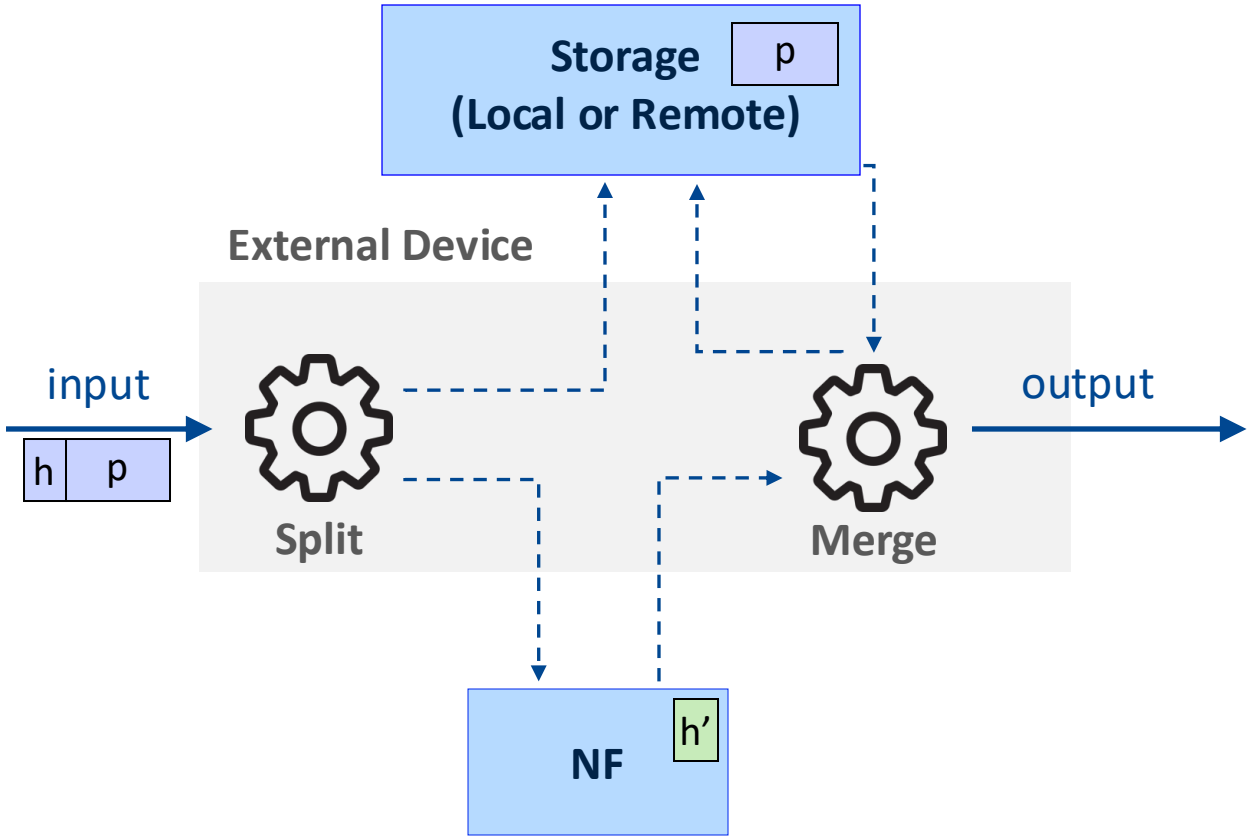
e.g., Ribosome, nicmem



Minimize CPU Cycles per Packet

State-of-the-art **packet-splitting** solutions **increase** shallow NFs throughput

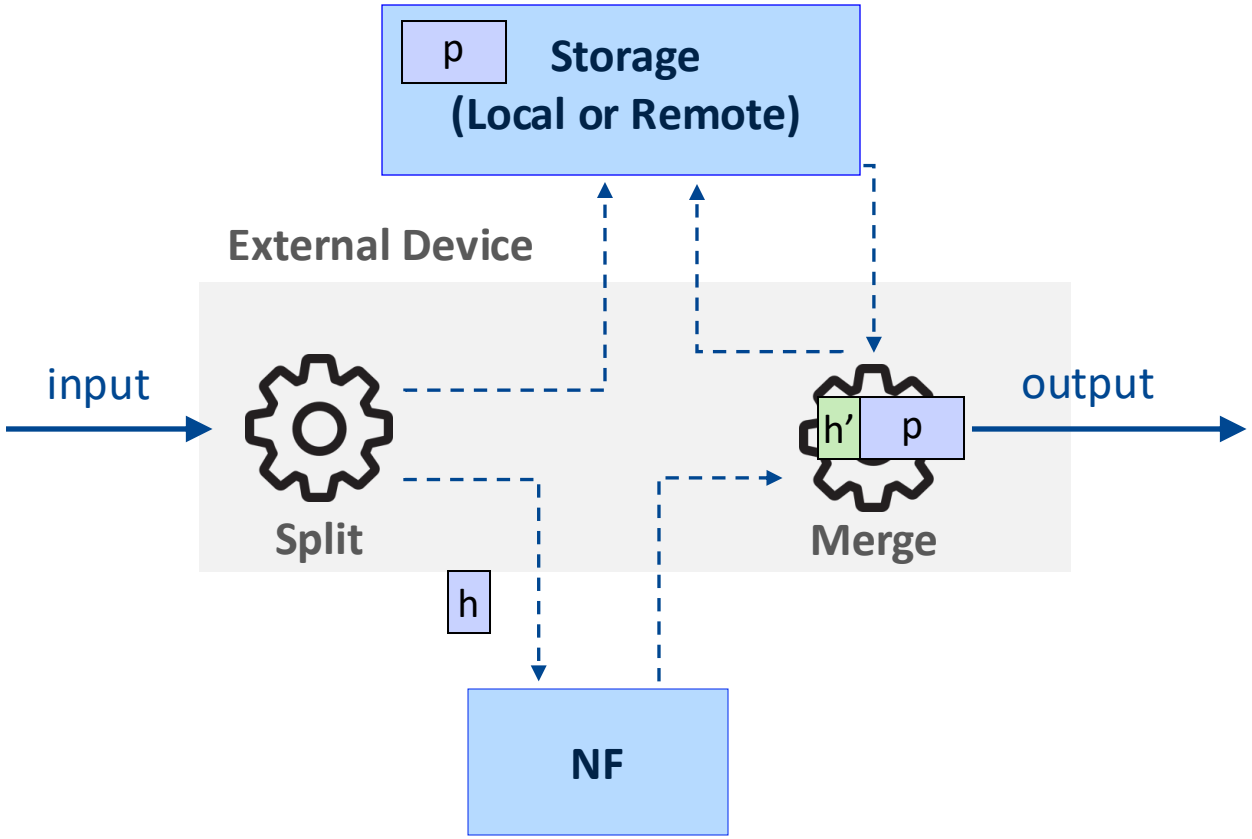
e.g., Ribosome, nicmem



Minimize CPU Cycles per Packet

State-of-the-art **packet-splitting** solutions **increase** shallow NFs throughput

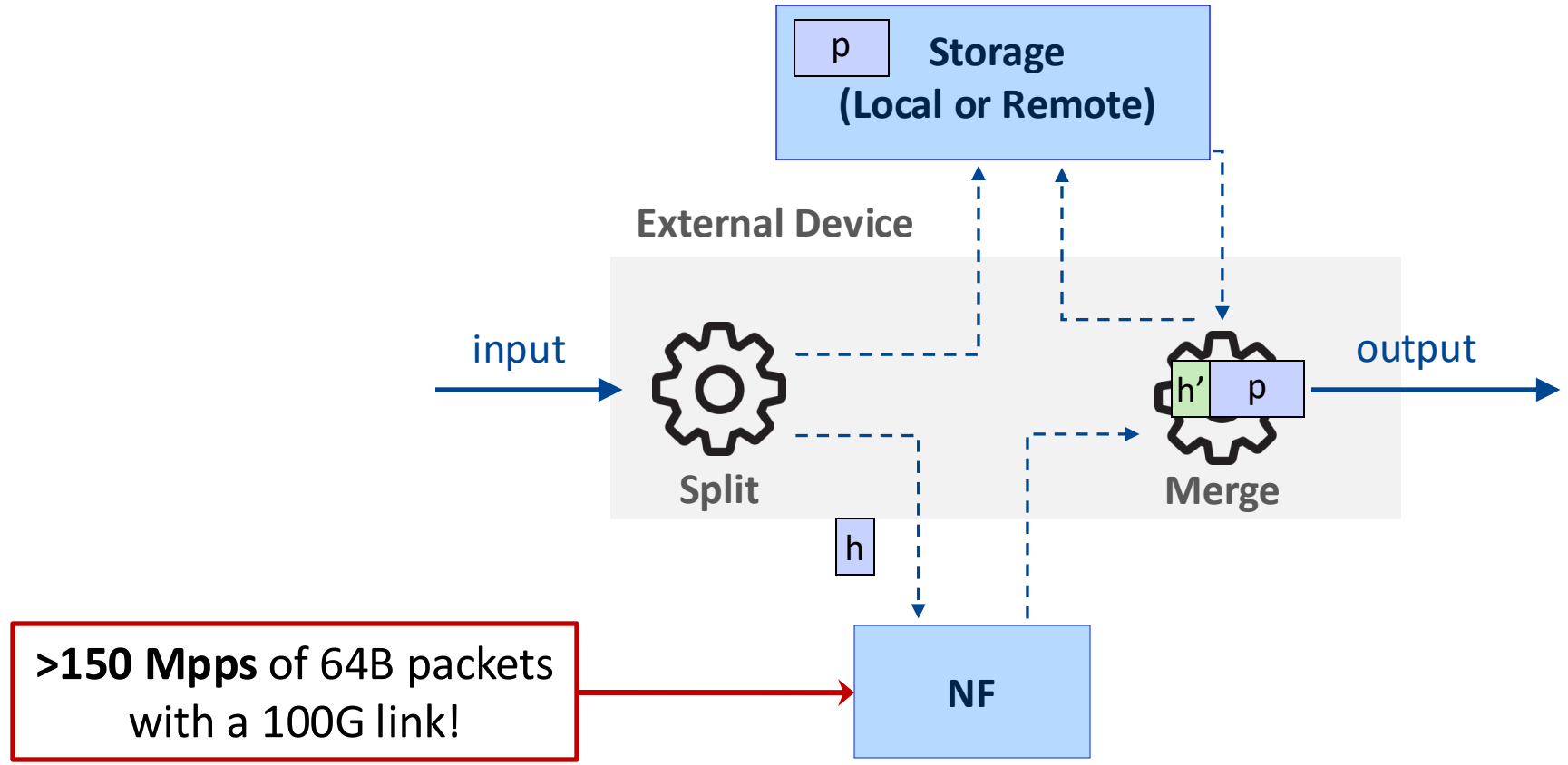
e.g., Ribosome, nicmem



Minimize CPU Cycles per Packet

State-of-the-art **packet-splitting** solutions **increase** shallow NFs throughput

e.g., Ribosome, nicmem

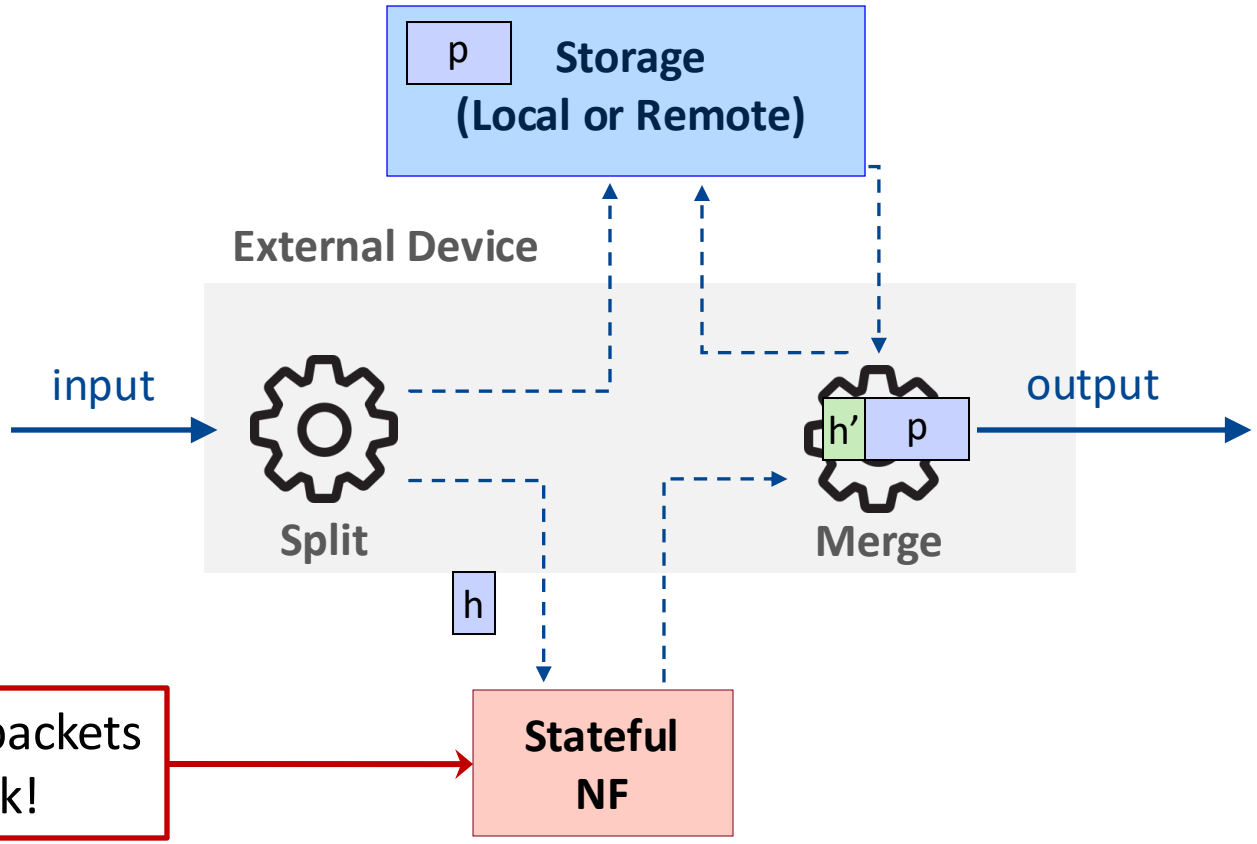


>150 Mpps of 64B packets with a 100G link!

Minimize CPU Cycles per Packet

State-of-the-art **packet-splitting** solutions **increase** shallow NFs throughput

e.g., Ribosome, nicmem



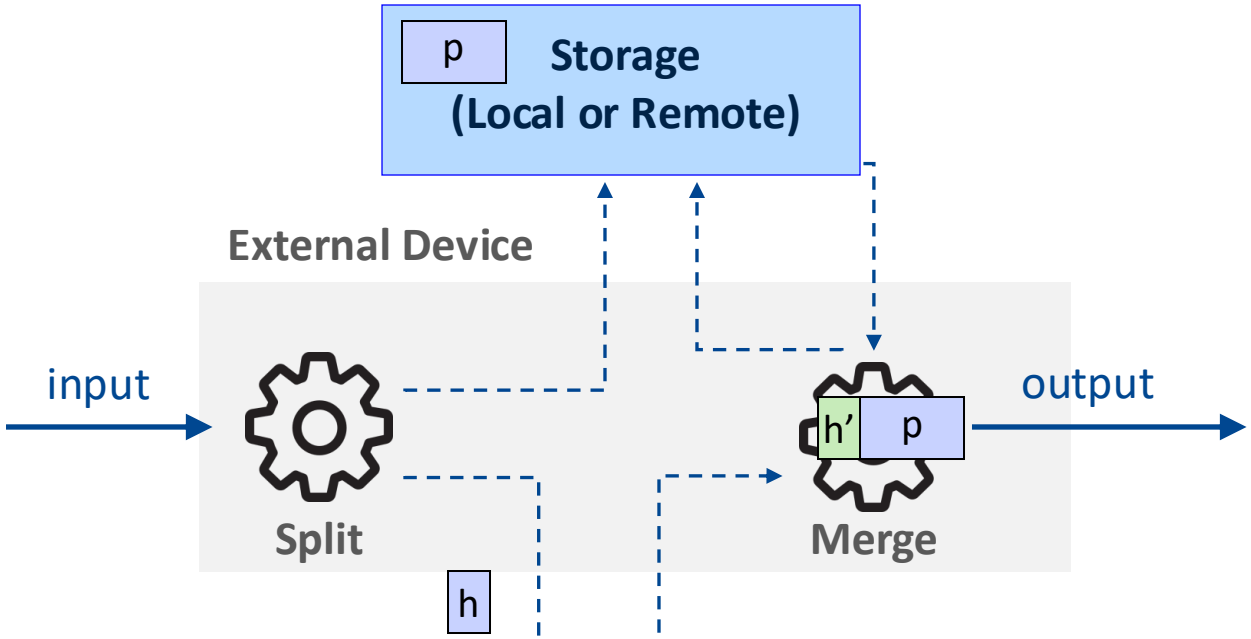
>150 Mpps of 64B packets with a 100G link!

Store **per-flow** states!

Minimize CPU Cycles per Packet

State-of-the-art **packet-splitting** solutions **increase** shallow NFs throughput

None of the existing frameworks can handle such loads!



>150 Mpps of 64B packets with a 100G link!

Store **per-flow** states**!**

Can we scale complex, stateful
packet processing chains on commodity hardware to
over 100 millions packets per second?

Essential Principles of Stateful Packet Processors

Batch Processing

Minimize Memory Accesses

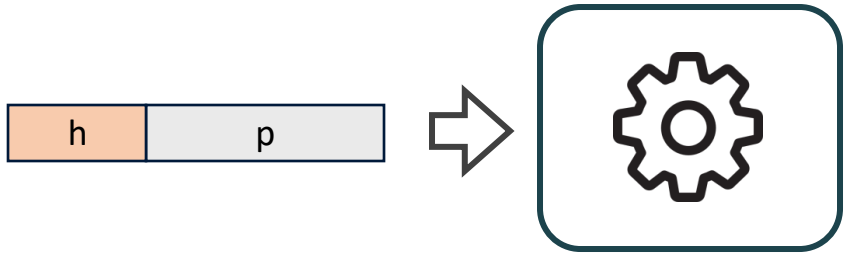
Minimize Sync. Overheads

Essential Principles of Stateful Packet Processors

- **Batch Processing**

Minimize Memory Accesses

Minimize Sync. Overheads



Essential Principles of Stateful Packet Processors

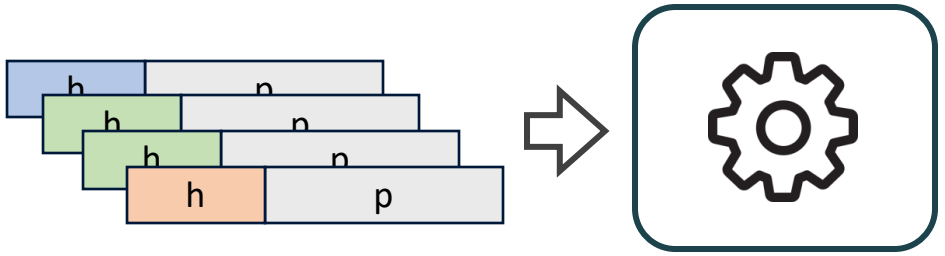
✓ VPP

› Batch Processing

Minimize Memory Accesses

Minimize Sync. Overheads

Process **batches** of packets



Maximizes instruction **cache hits**

Enables **software prefetching** of data

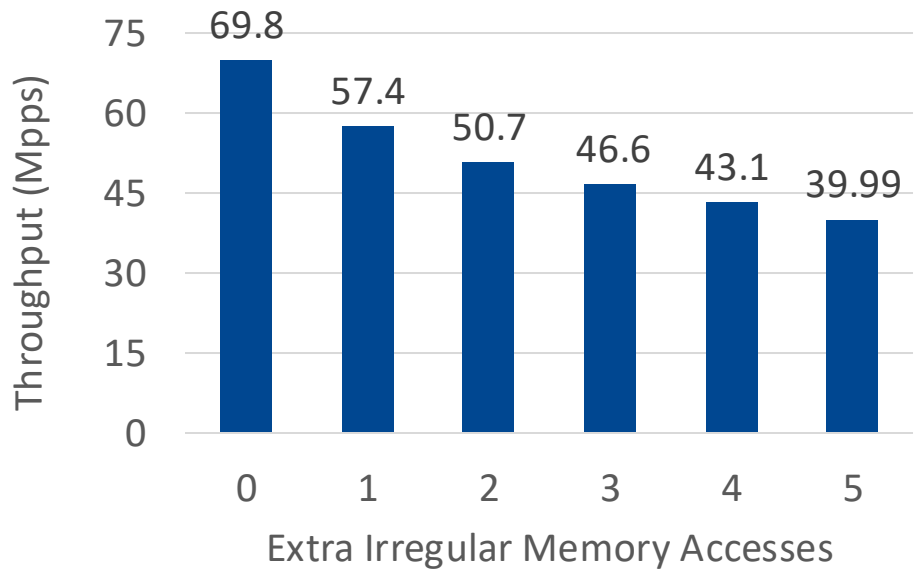
Essential Principles of Stateful Packet Processors

✓ VPP

Batch Processing > **Minimize Memory Accesses** **Minimize Sync. Overheads**

Hash tables are the essential data structure in stateful NFs

Require **at least two** random memory accesses



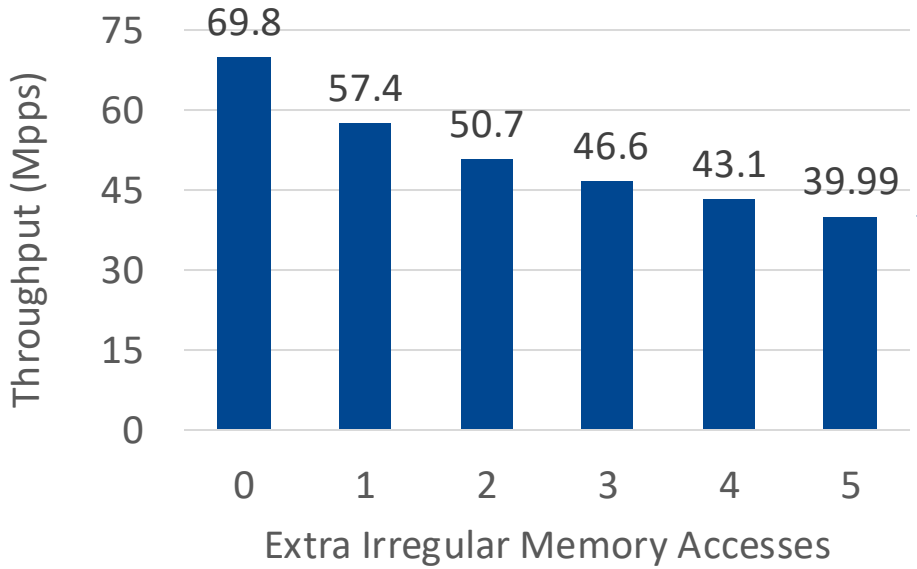
Essential Principles of Stateful Packet Processors

✓ VPP

Batch Processing > **Minimize Memory Accesses** **Minimize Sync. Overheads**

Hash tables are the essential data structure in stateful NFs

Require **at least two** random memory accesses



Each memory access **drops performance**!

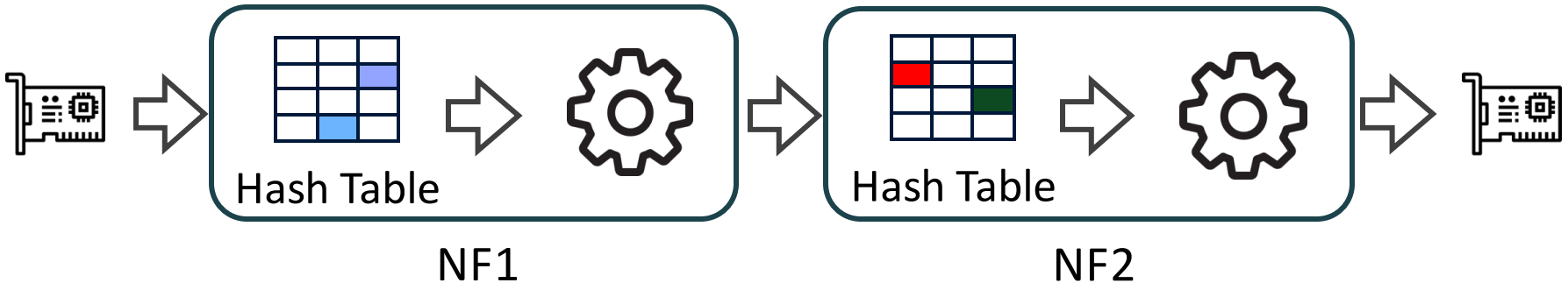
How to **minimize** memory accesses?

Essential Principles of Stateful Packet Processors

✓ VPP

Batch Processing > **Minimize Memory Accesses** **Minimize Sync. Overheads**

? How to **minimize** memory accesses? → **Aggregate** States

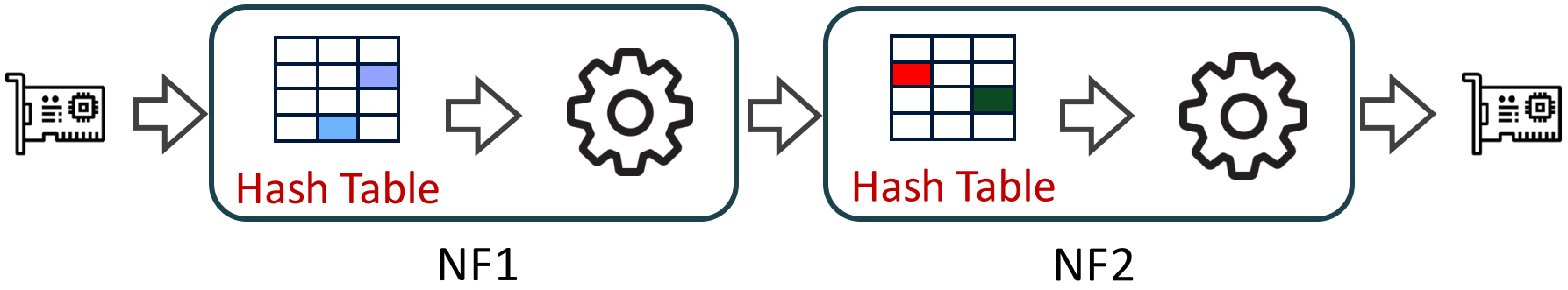


Essential Principles of Stateful Packet Processors

✓ VPP

Batch Processing > **Minimize Memory Accesses** **Minimize Sync. Overheads**

? How to **minimize** memory accesses? ➡ 💡 **Aggregate** States

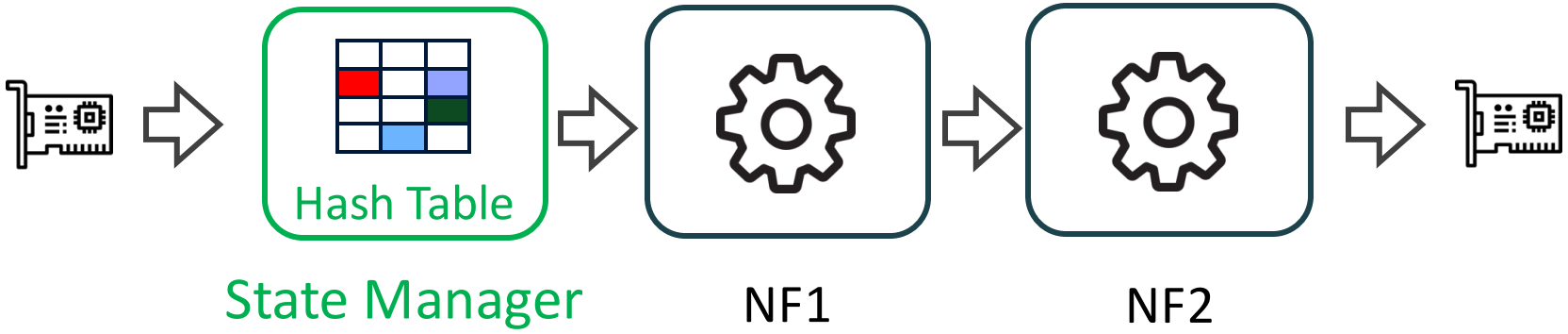


Essential Principles of Stateful Packet Processors

✓ VPP — Batch Processing

✓ FastClick — Minimize Memory Accesses > Minimize Sync. Overheads

? How to **minimize** memory accesses? → **Aggregate** States



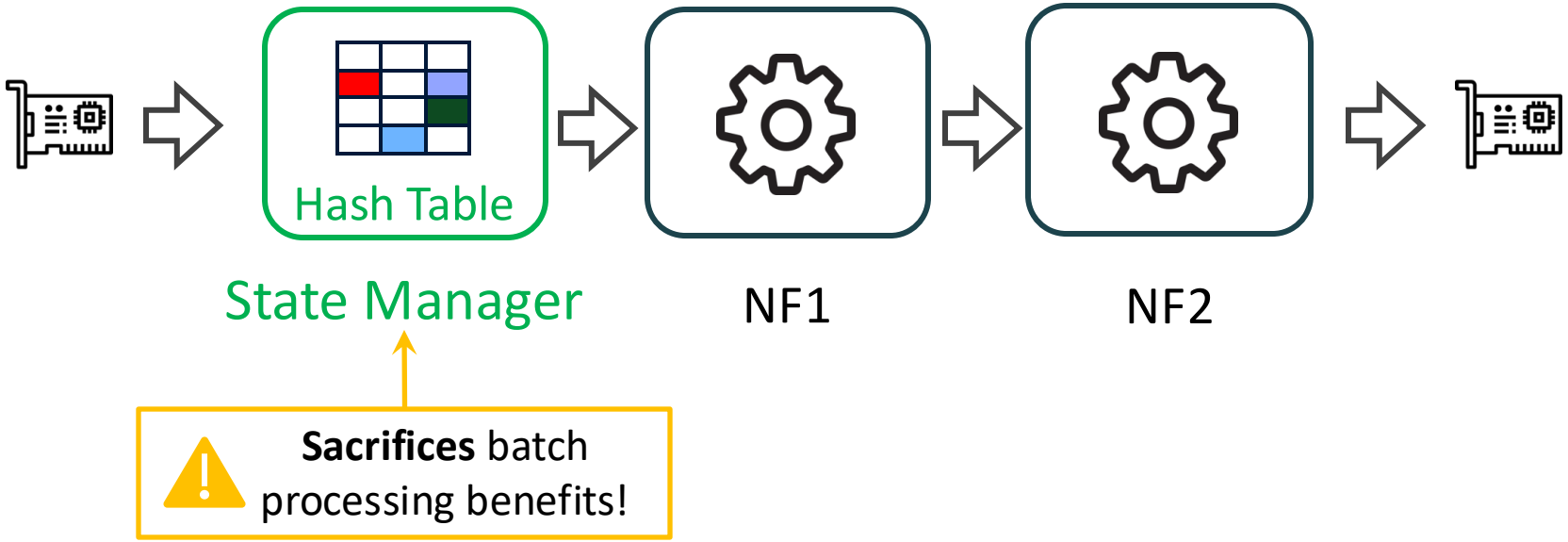
Essential Principles of Stateful Packet Processors

✓ VPP — Batch Processing

✓ FastClick — Minimize Memory Accesses

Minimize Sync. Overheads

? How to **minimize** memory accesses? → **Aggregate** States



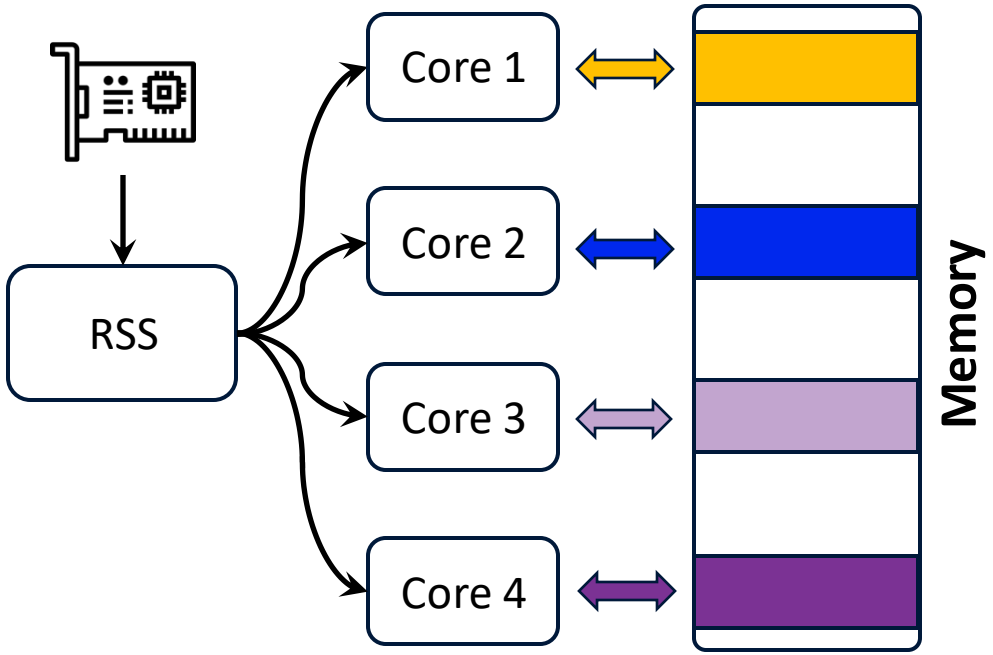
Essential Principles of Stateful Packet Processors

✓ VPP — **Batch Processing**

✓ FastClick — **Minimize Memory Accesses**

› **Minimize Sync. Overheads**

? How to **scale** NFs on **multiple cores**? ➡ **Shared-Nothing** Model



Essential Principles of Stateful Packet Processors

✓ VPP — Batch Processing

✓ FastClick — Minimize Memory Accesses

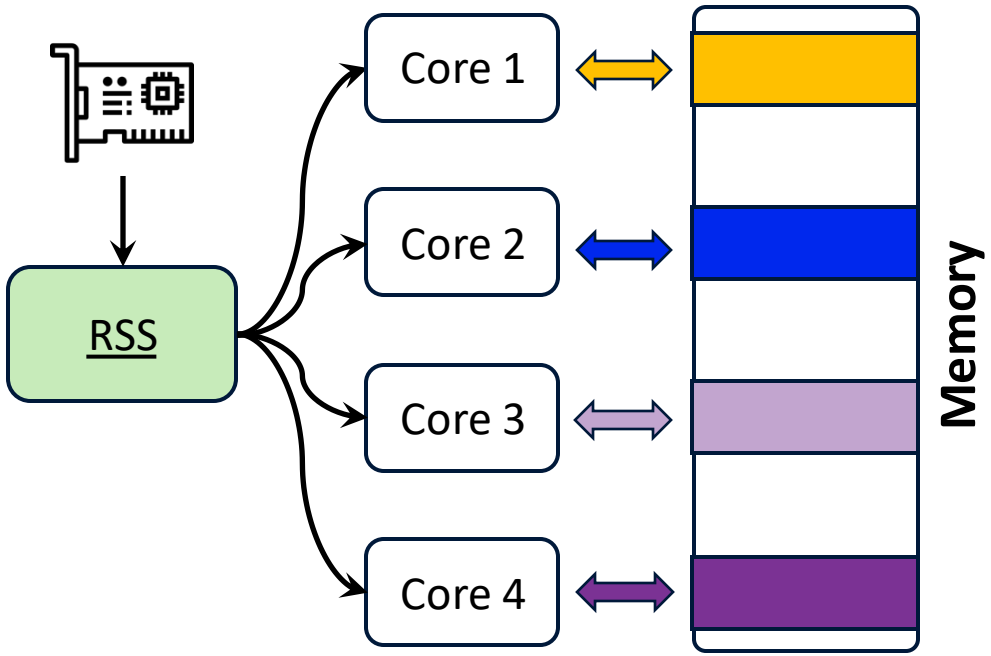
› Minimize Sync. Overheads

? How to **scale** NFs on **multiple cores**? → **Shared-Nothing** Model

👍 **High Performance**

👍 **Per-flow Consistency**

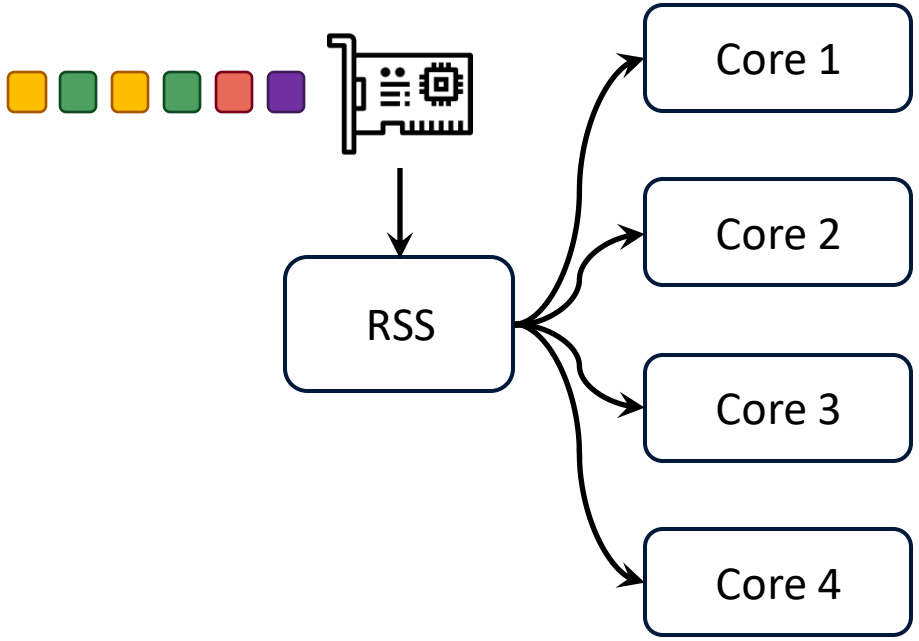
✗ **Not applicable in all scenarios!**



Shared-Nothing Model is not applicable in all scenarios!

(i) Efficient Inter-Core Load Balancing

⚠ RSS **struggles** to efficiently distribute packets among cores with **adversarial loads**

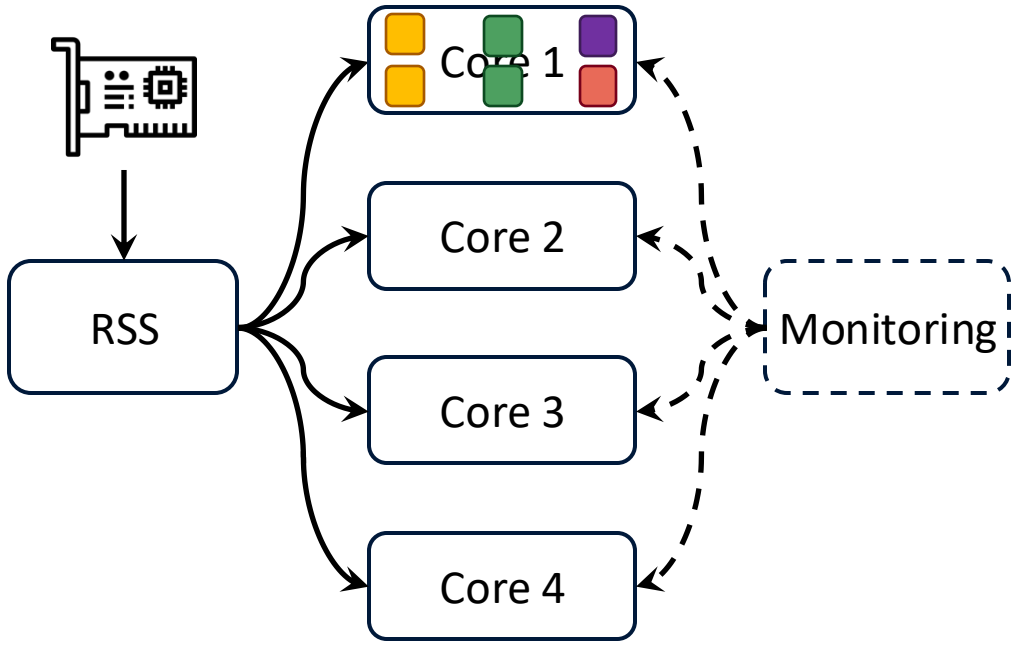


Shared-Nothing Model is not applicable in all scenarios!

(i) Efficient Inter-Core Load Balancing

⚠ RSS **struggles** to efficiently distribute packets among cores with **adversarial loads**

ℹ **Monitor** and **rebalance** flows among cores (e.g., Dyssect, RSS++)

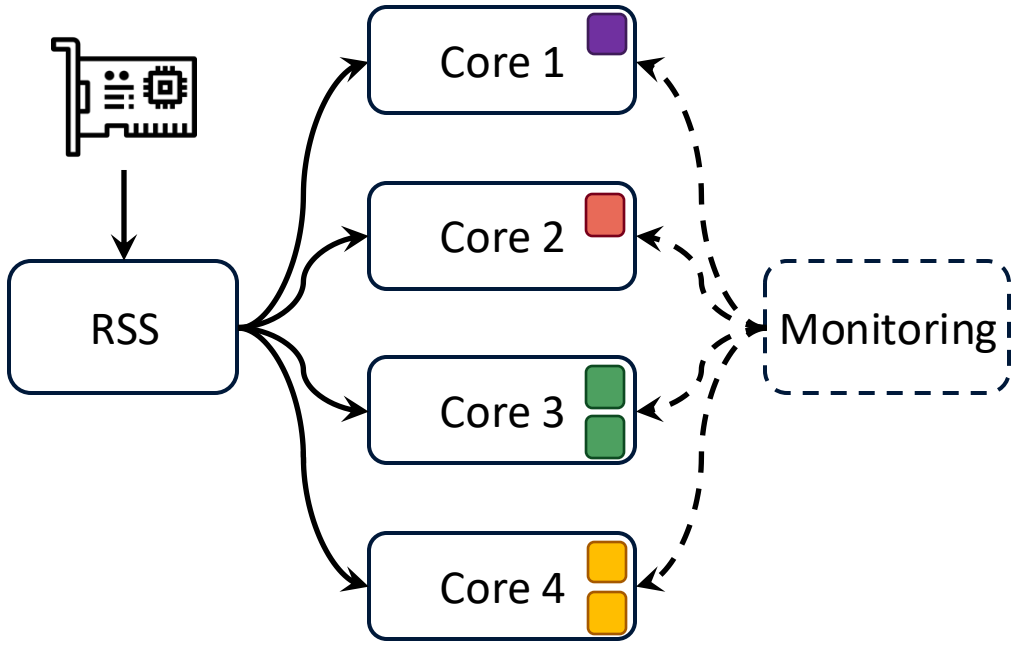


Shared-Nothing Model is not applicable in all scenarios!

(i) Efficient Inter-Core Load Balancing

⚠ RSS **struggles** to efficiently distribute packets among cores with **adversarial loads**

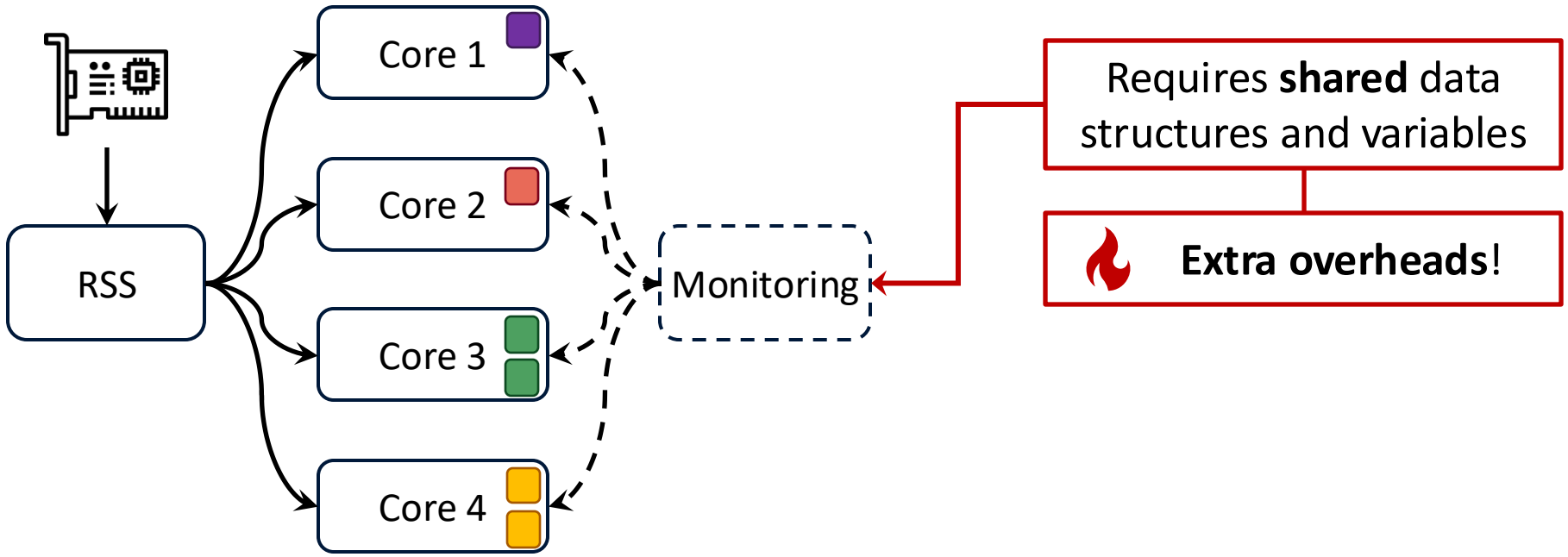
ℹ **Monitor** and **rebalance** flows among cores (e.g., Dyssect, RSS++)



Shared-Nothing Model is not applicable in all scenarios!

(i) Efficient Inter-Core Load Balancing

- RSS **struggles** to efficiently distribute packets among cores with **adversarial loads**
- **Monitor** and **rebalance** flows among cores (e.g., Dyssect, RSS++)



Shared-Nothing Model is not applicable in all scenarios!

(i) Efficient Inter-Core Load Balancing

(ii) Stateful NFs with Diverse Flow Definitions

The definition of state is **not exclusively** the 5-tuple!

What happens if they are **chained together**?

Load Balancer	Src IP	Dst IP	Src Port	Dst Port	Protocol
Policer	Src IP	Dst IP	Src Port	Dst Port	Protocol
Port Scan Detector	Src IP	Dst IP	Src Port	Dst Port	Protocol

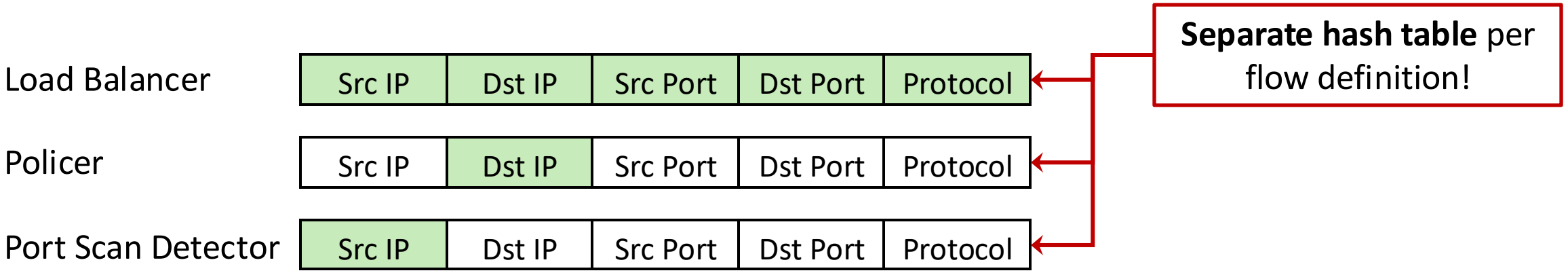
Shared-Nothing Model is not applicable in all scenarios!

(i) Efficient Inter-Core Load Balancing

(ii) Stateful NFs with Diverse Flow Definitions

The definition of state is **not exclusively** the 5-tuple!

What happens if they are **chained together**?



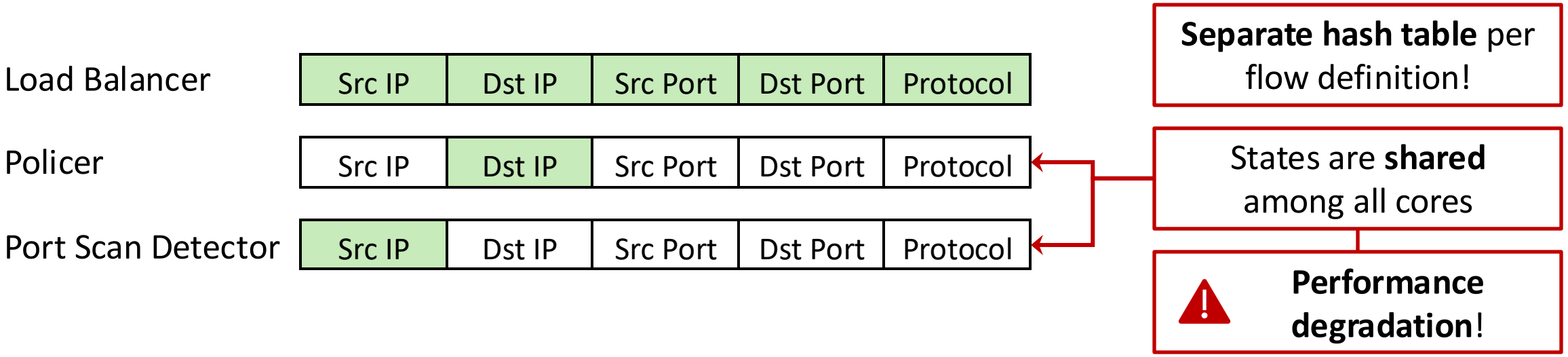
Shared-Nothing Model is not applicable in all scenarios!

(i) Efficient Inter-Core Load Balancing

(ii) Stateful NFs with Diverse Flow Definitions

The definition of state is **not exclusively** the 5-tuple!

What happens if they are **chained together**?



Essential Principles of Stateful Packet Processors

✓ **VPP**
Batch Processing

✓ **FastClick**
Minimize Memory Accesses

?
Minimize Sync. Overheads

Overlooking one of these principles **hinders**
Stateful NF frameworks to process >100Mpps

We **must** support the three principles **together**!

FAJITA

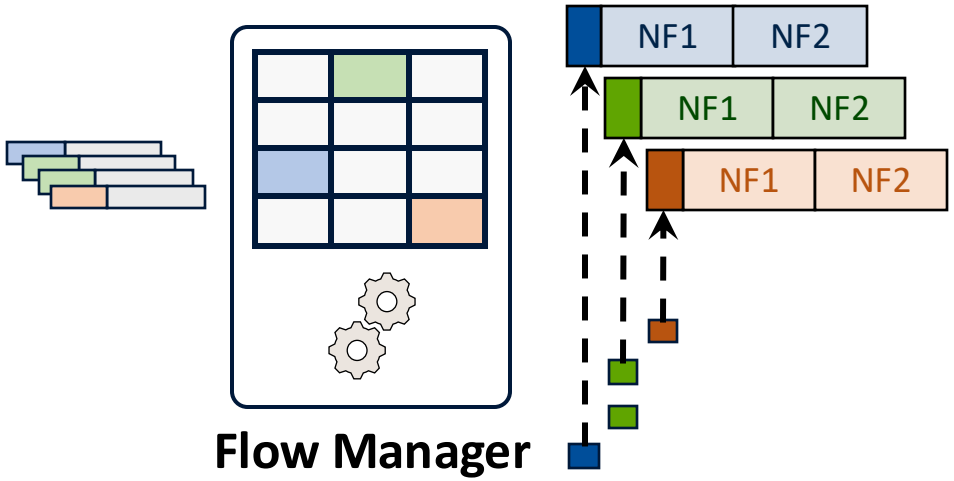
FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

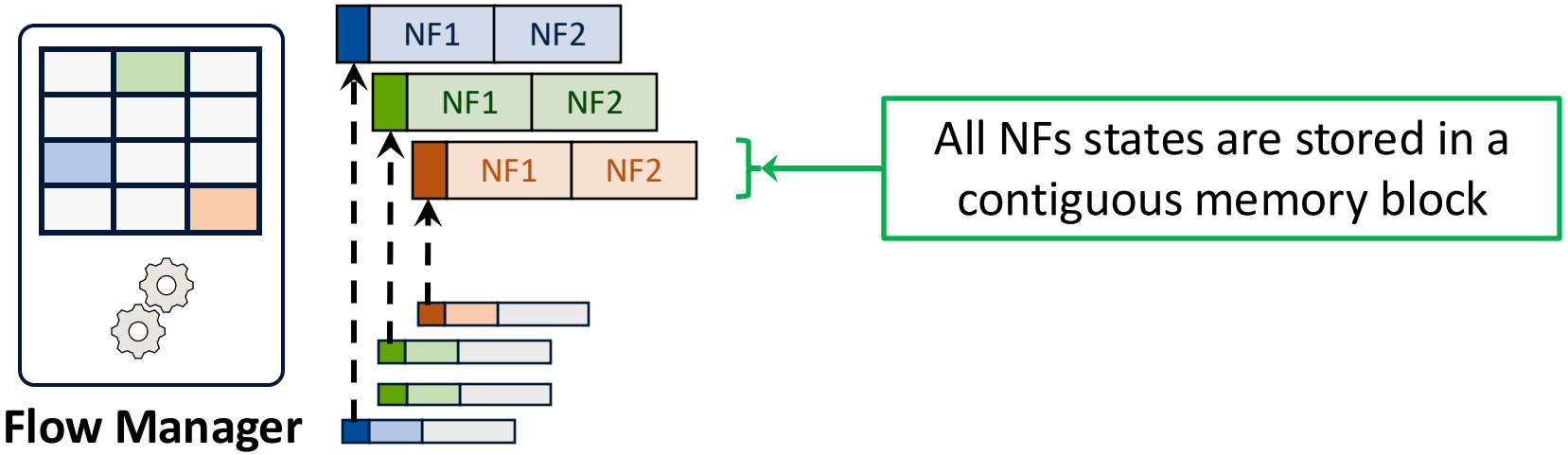
- **Optimized** state aggregation
- **Bulk Lookup** for **all packets** in the batch



FAJITA

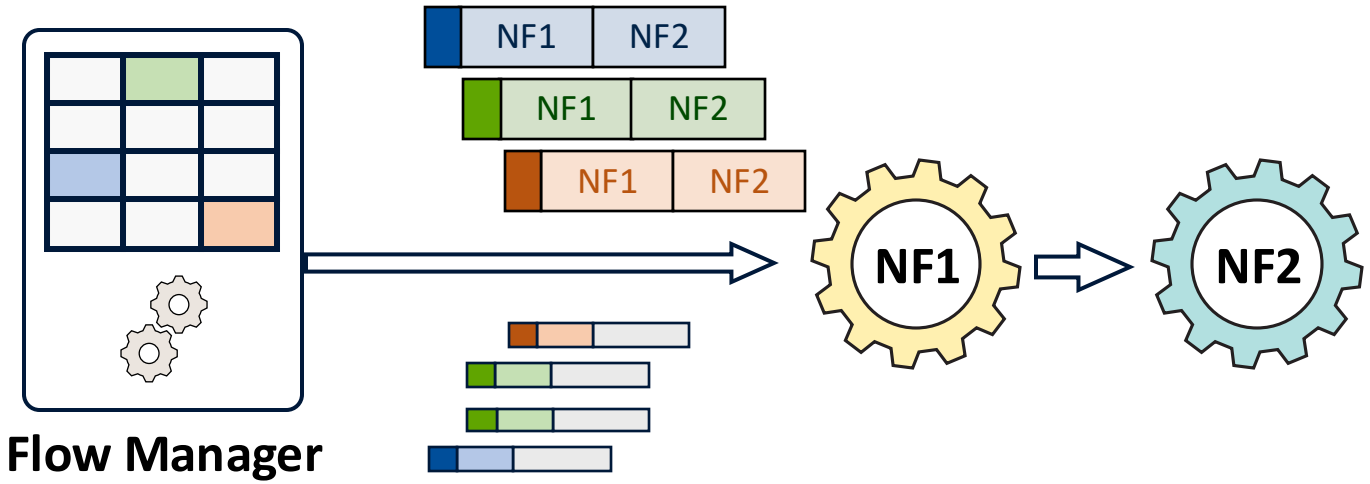
Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

- **Optimized** state aggregation
- **Bulk Lookup** for **all packets** in the batch → **Batches are preserved!**



FAJITA

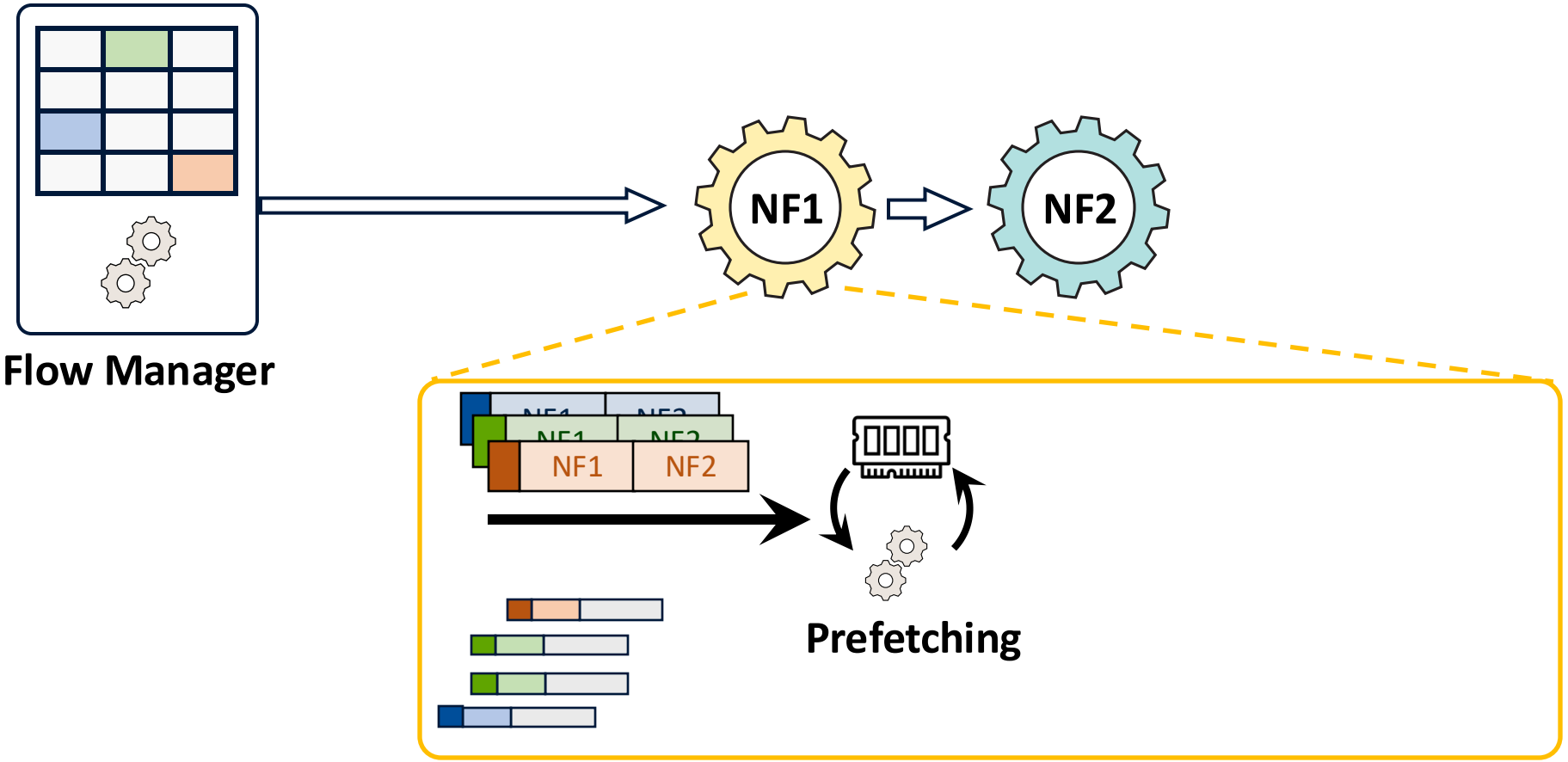
Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**



FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

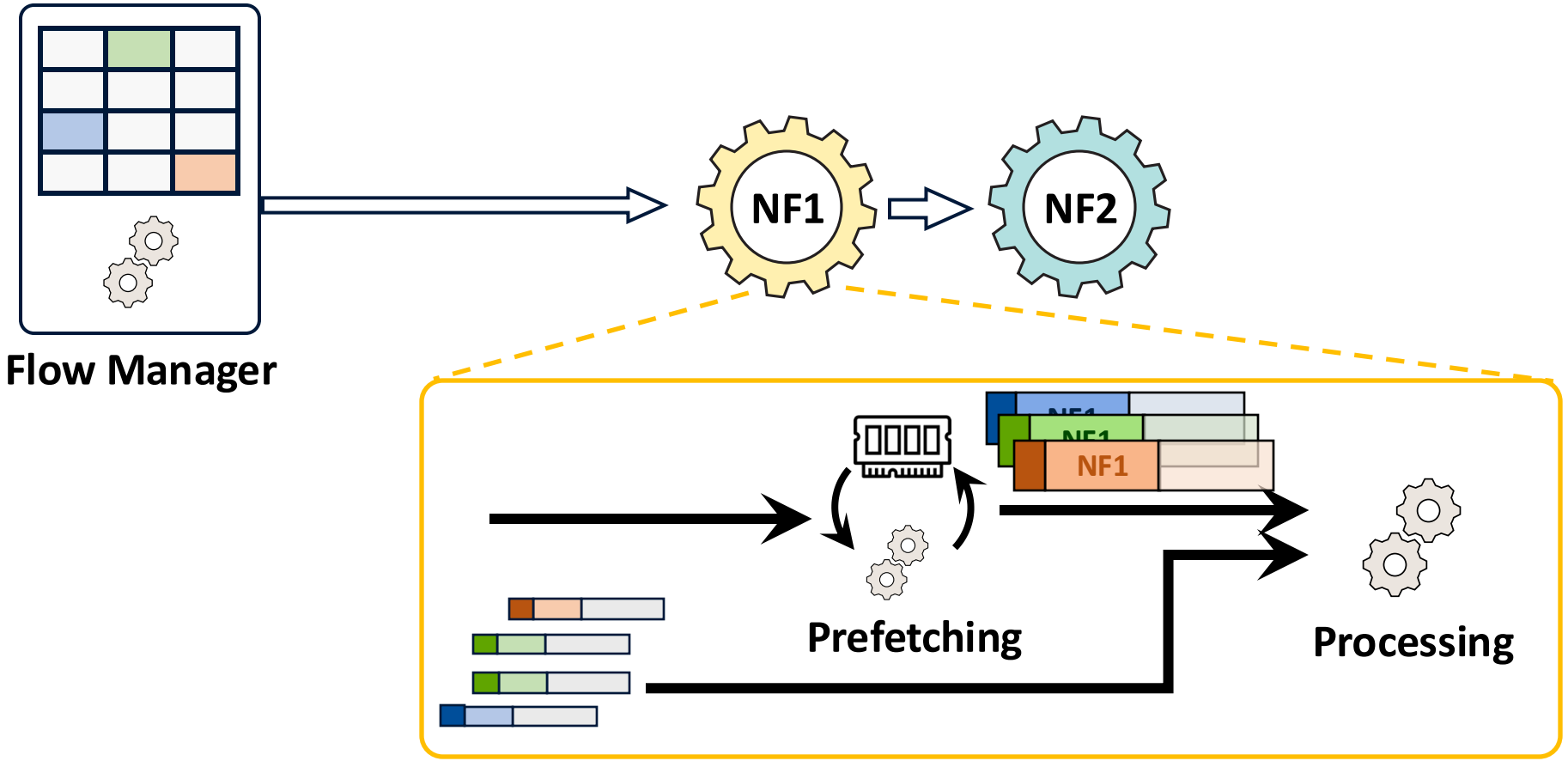
✓ Each NF **prefetches** its state data at the **right time**



FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

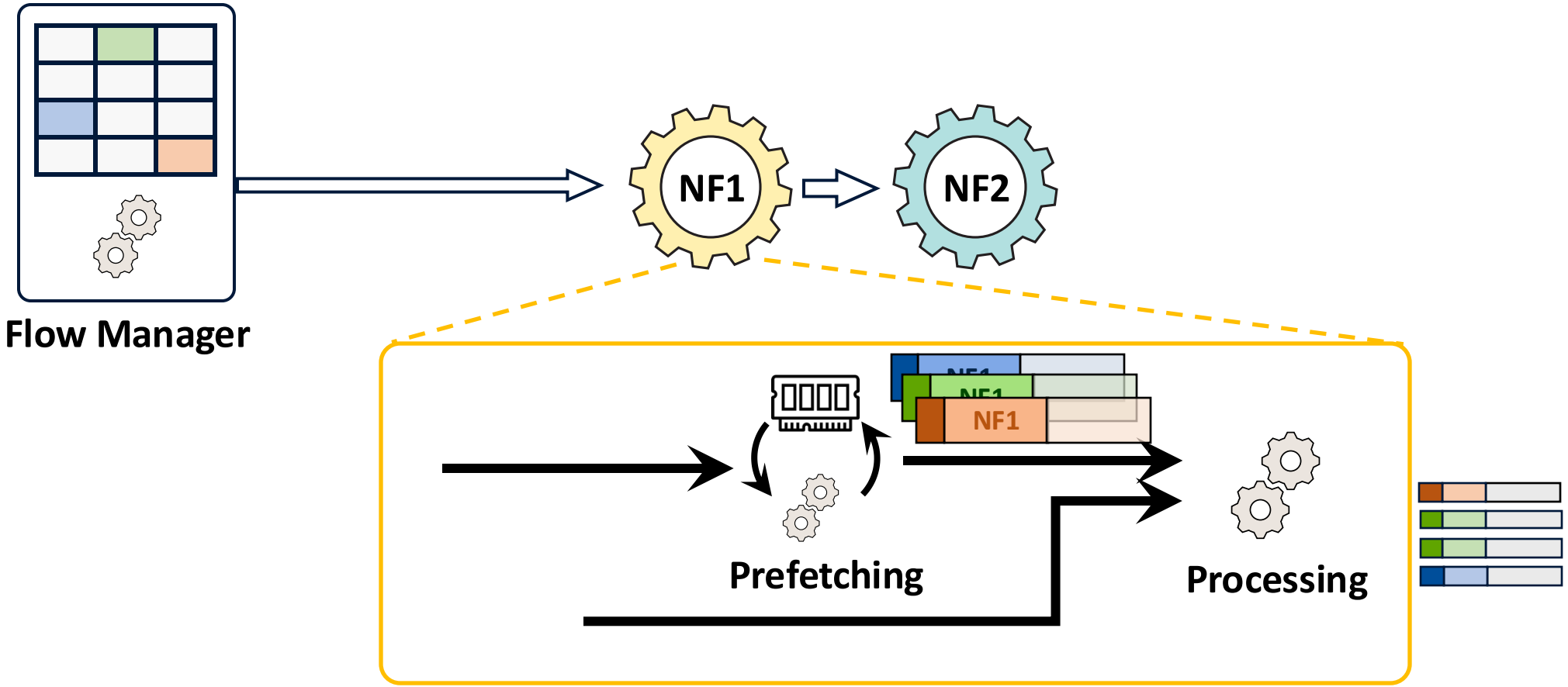
✓ Each NF **prefetches** its state data at the **right time** ➡ **Minimize memory loads overheads**



FAJITA

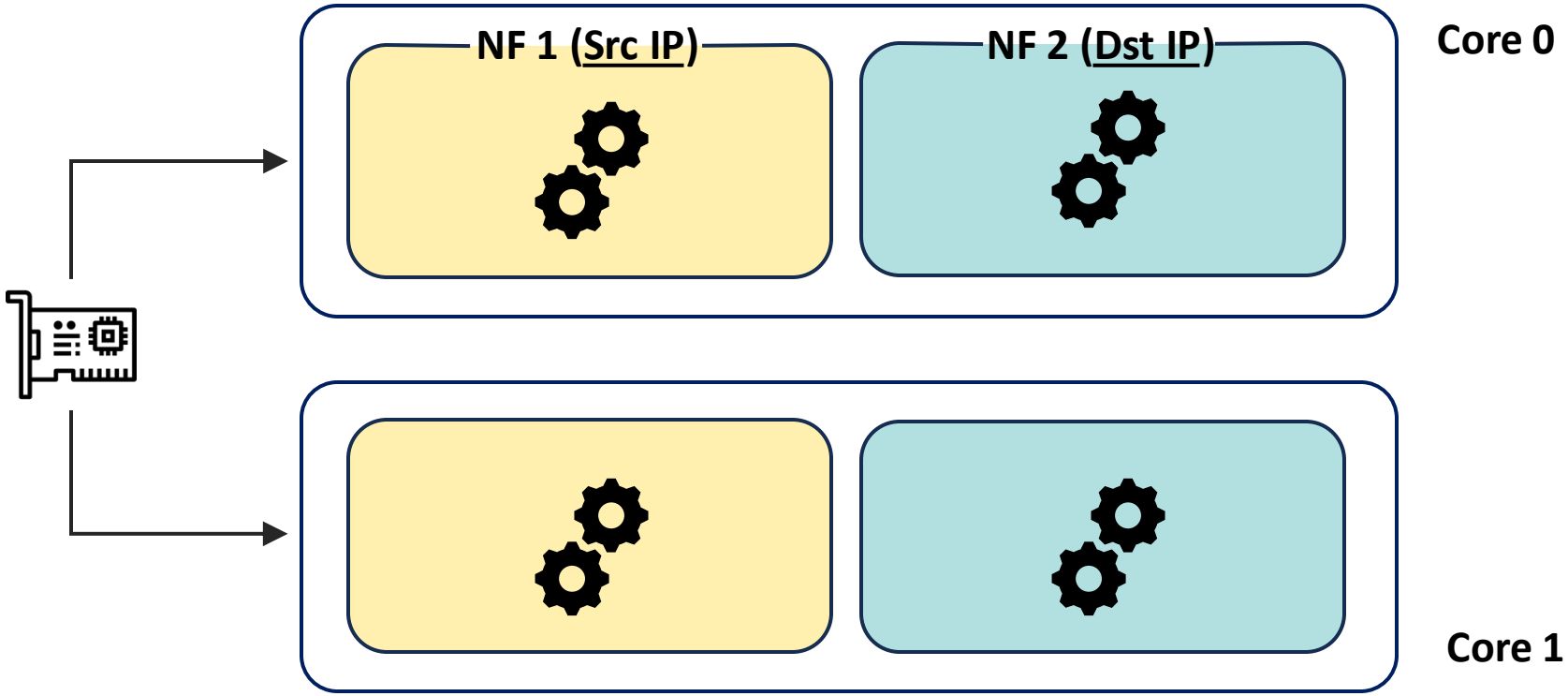
Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

⚠ Only considers 5-tuple flows definition!



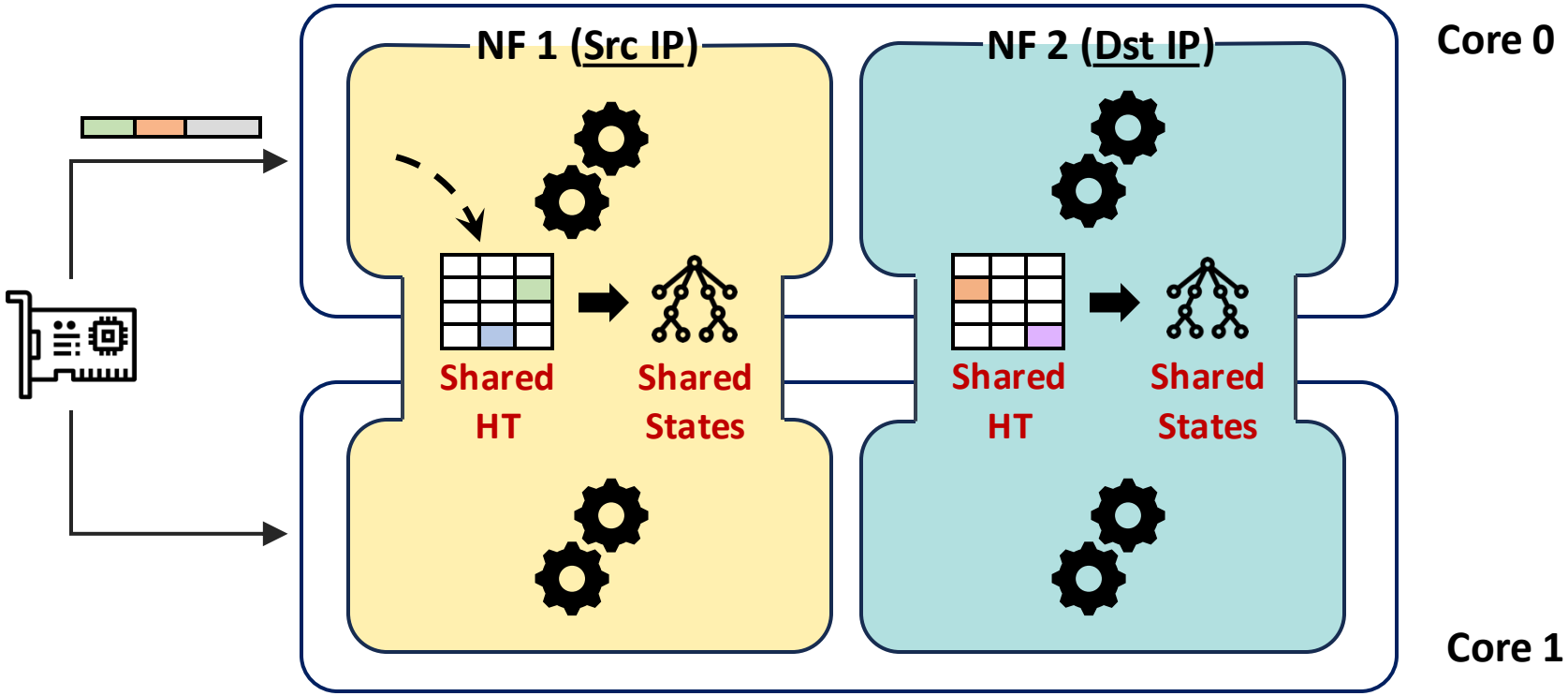
FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**



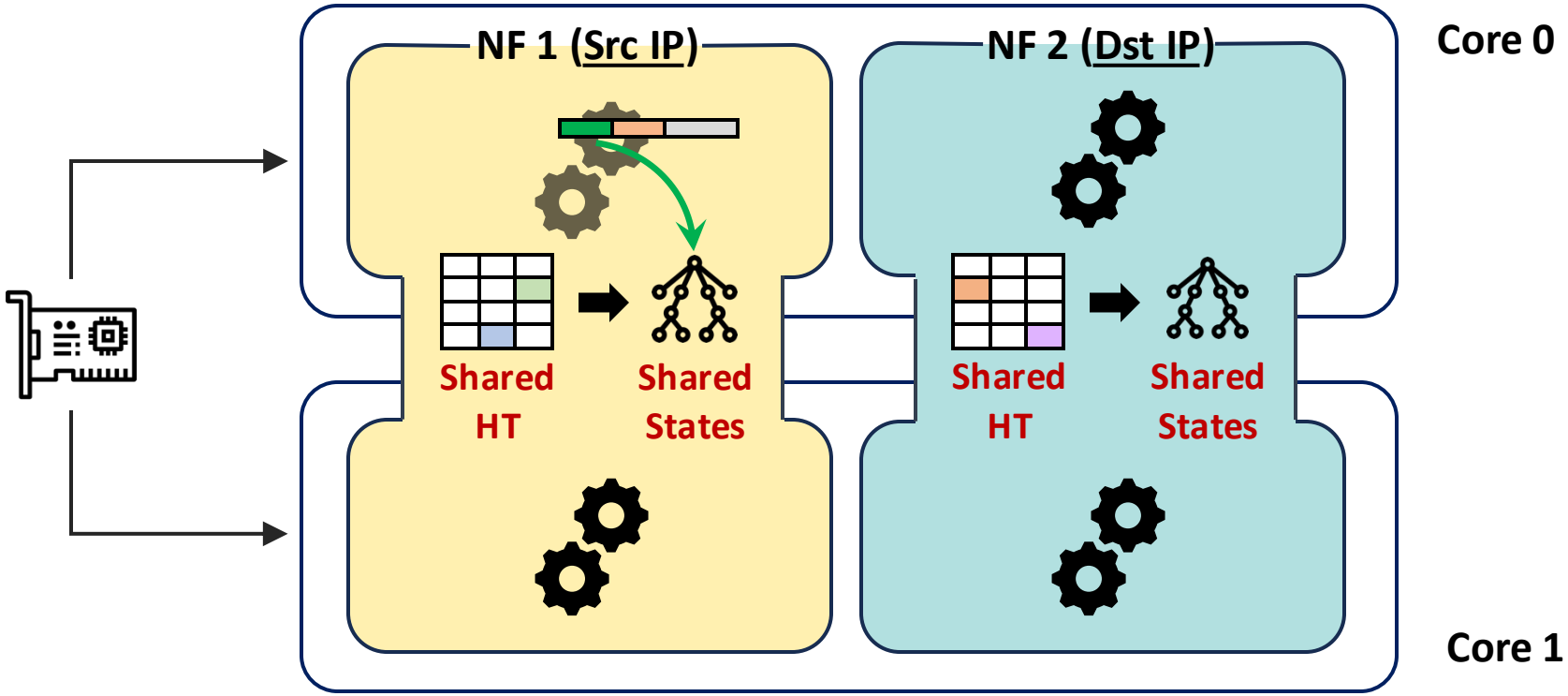
FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**



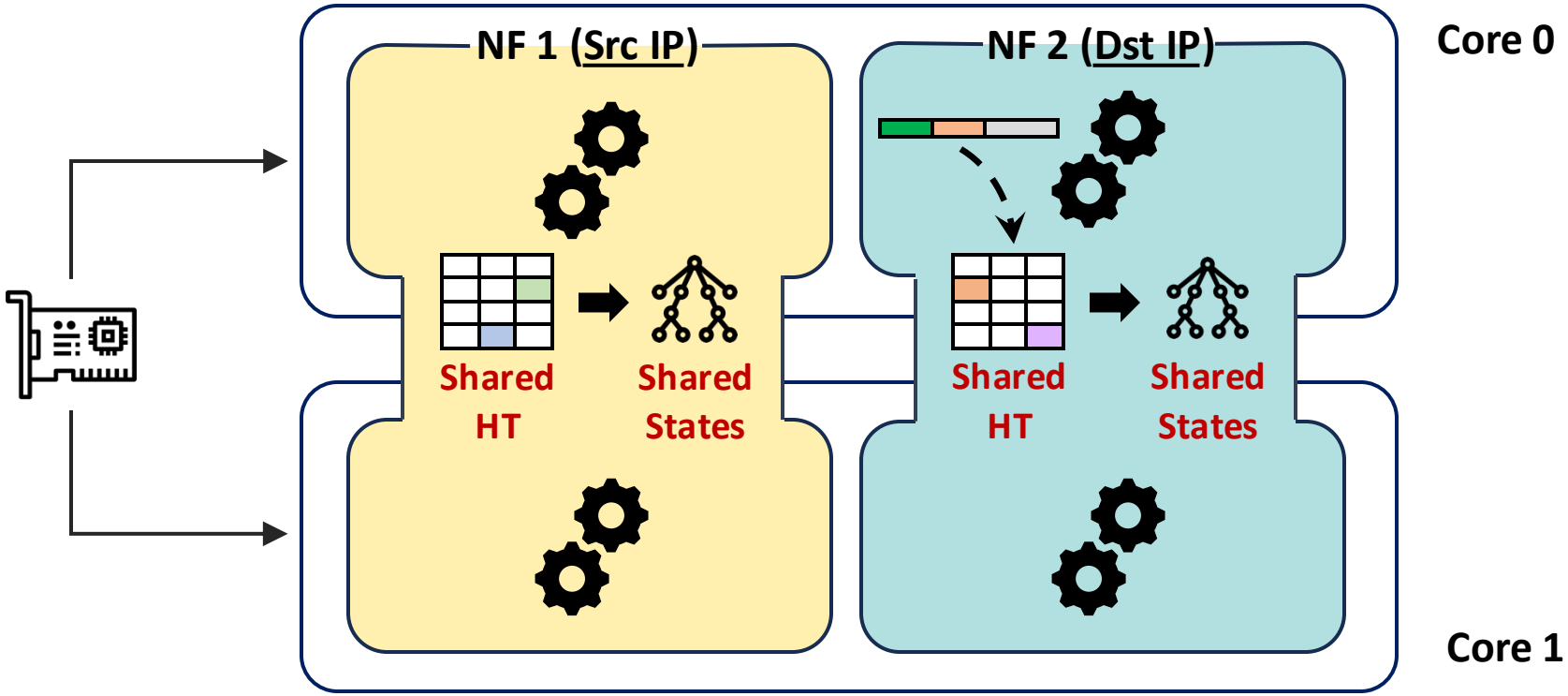
FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**



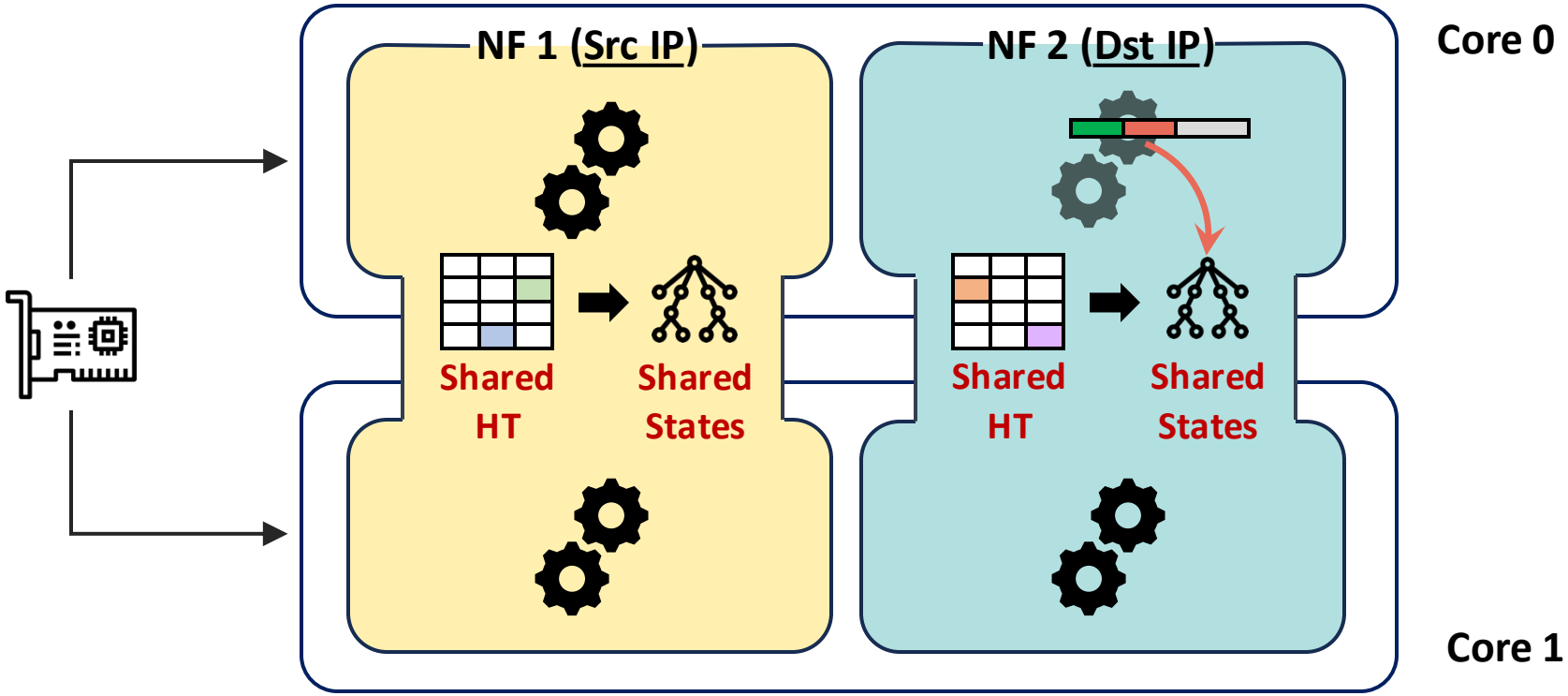
FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**



FAJITA

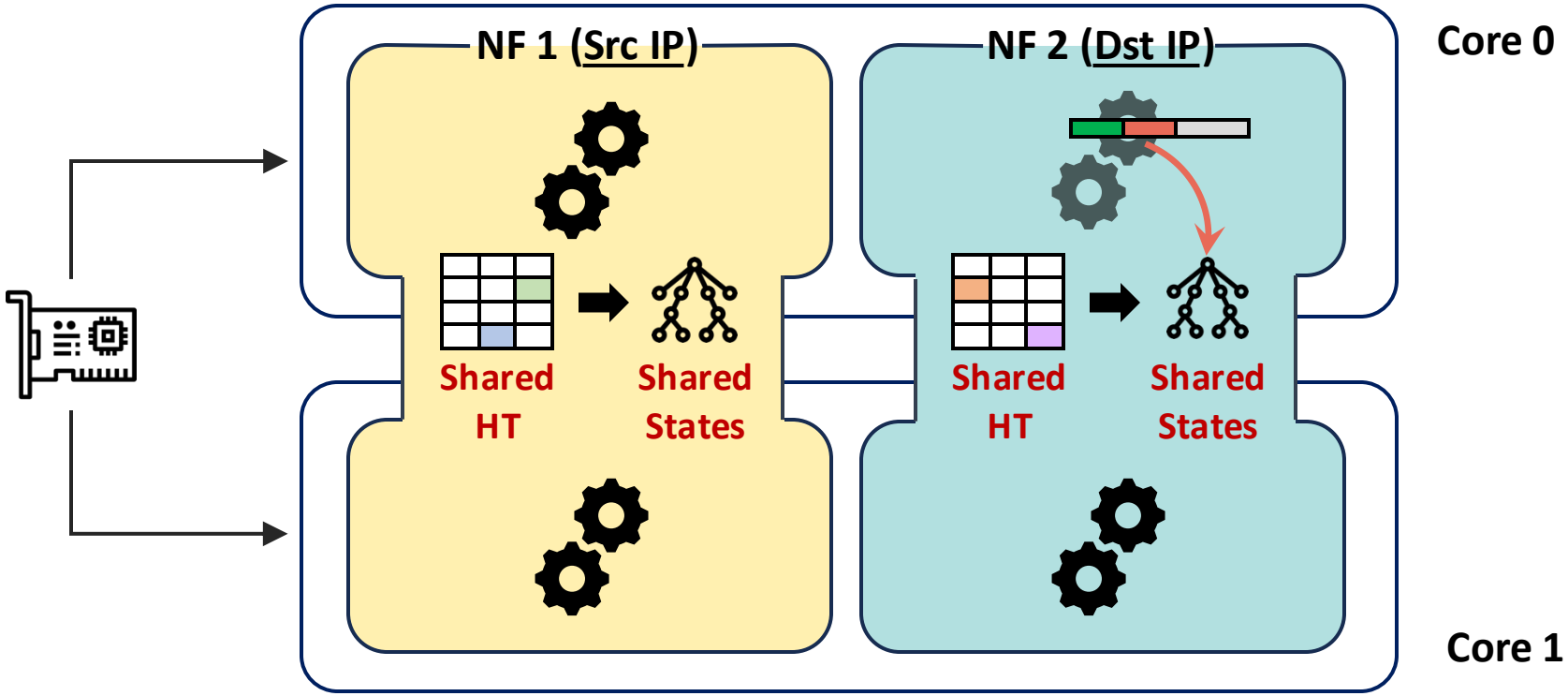
Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**



FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

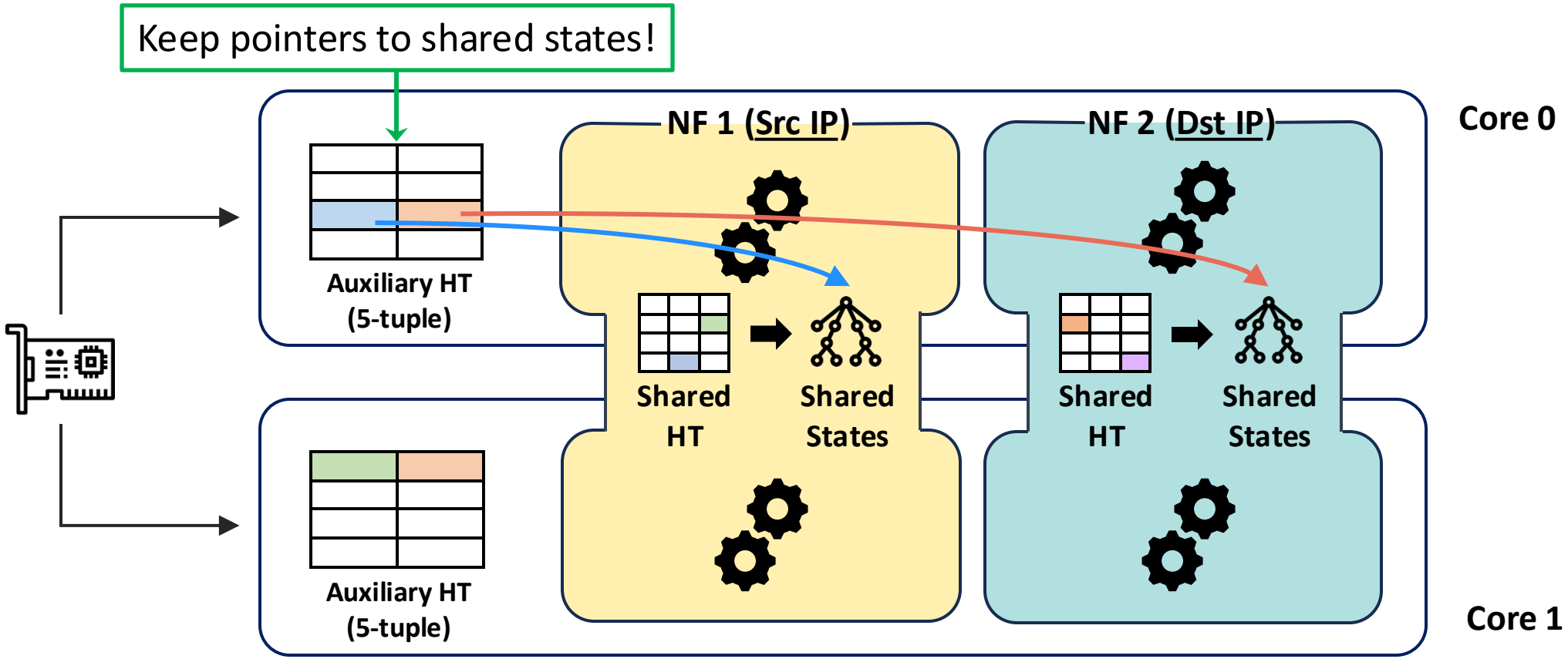
How can we **reduce** the **inevitable** sync. overhead with different flow definitions?



FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

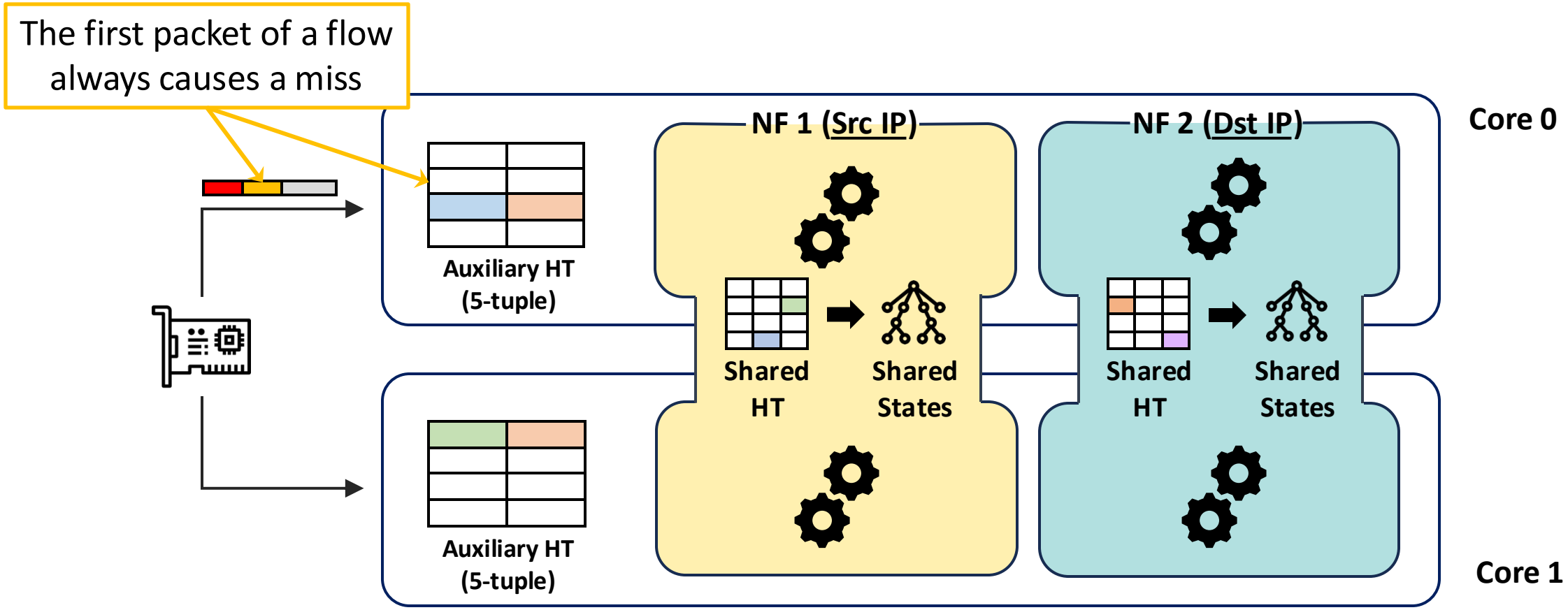
How can we **reduce** the **inevitable** sync. overhead with different flow definitions?



FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

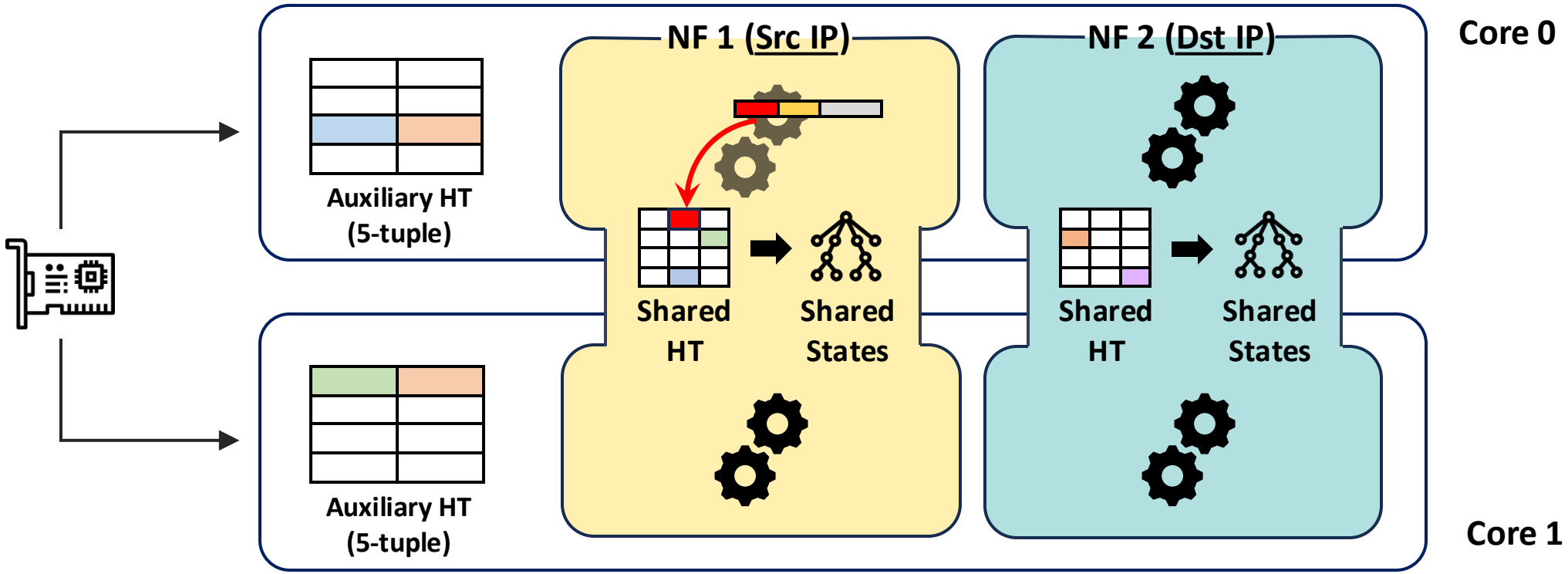
How can we **reduce** the **inevitable** sync. overhead with different flow definitions?



FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

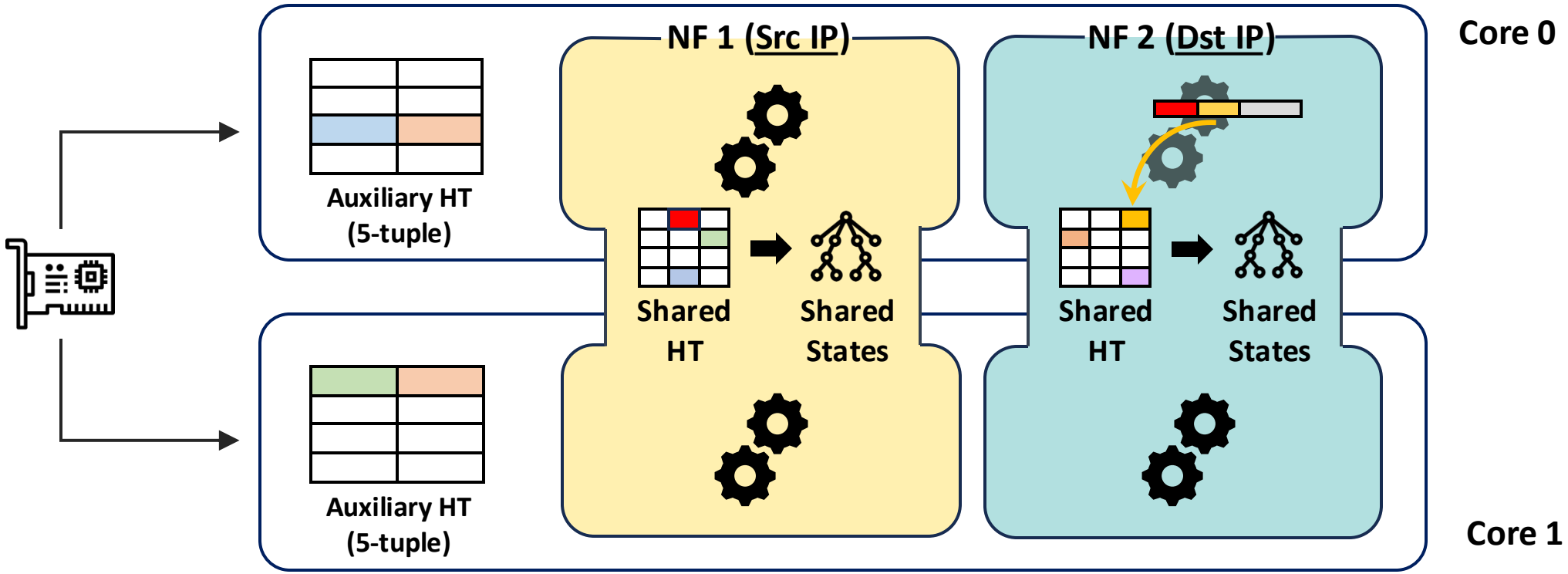
How can we **reduce** the **inevitable** sync. overhead with different flow definitions?



FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

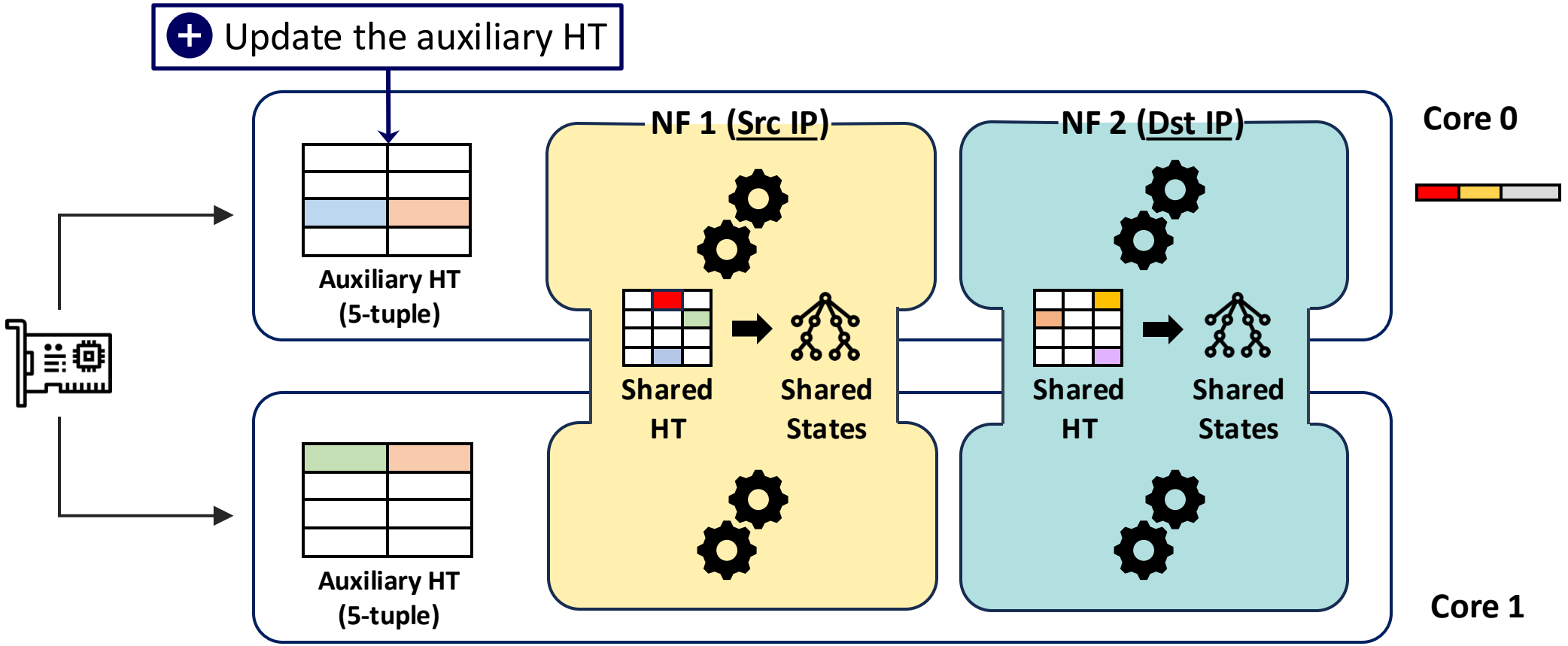
How can we **reduce** the **inevitable** sync. overhead with different flow definitions?



FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

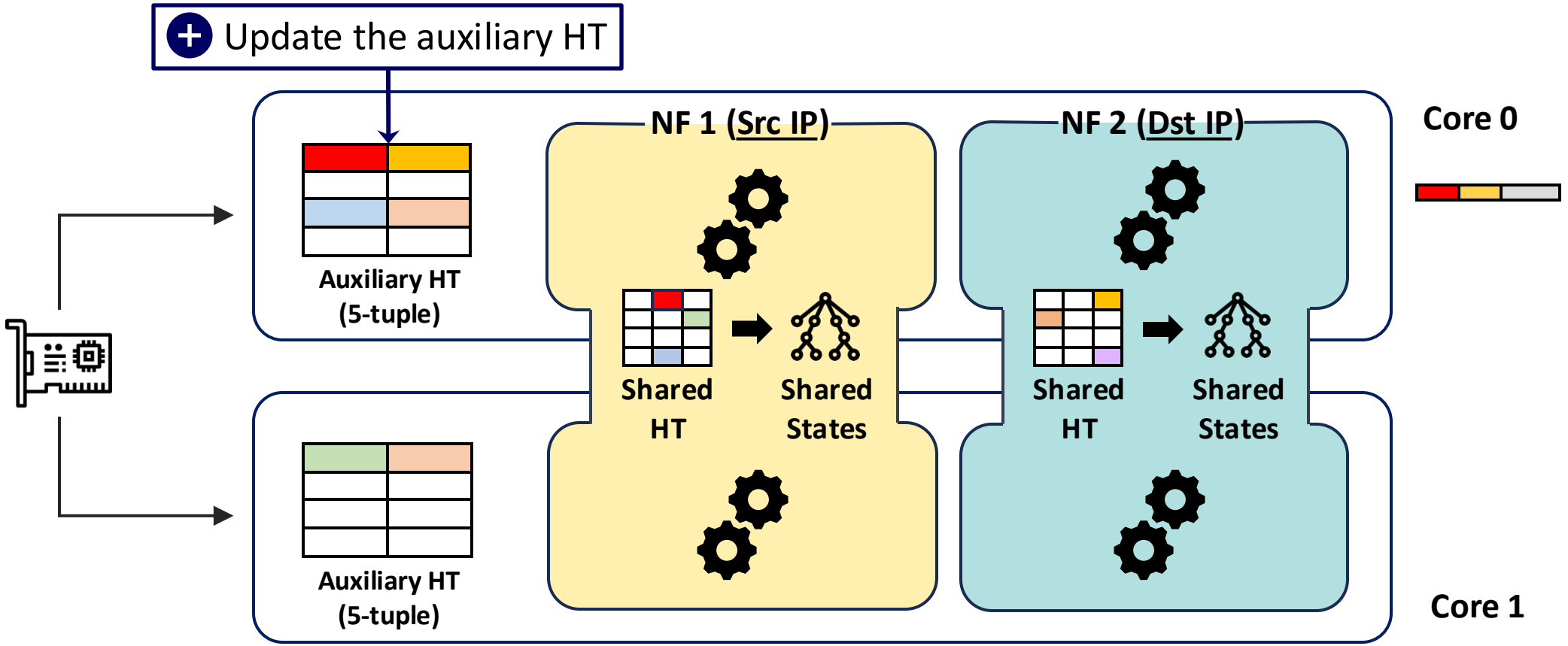
How can we **reduce** the **inevitable** sync. overhead with different flow definitions?



FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

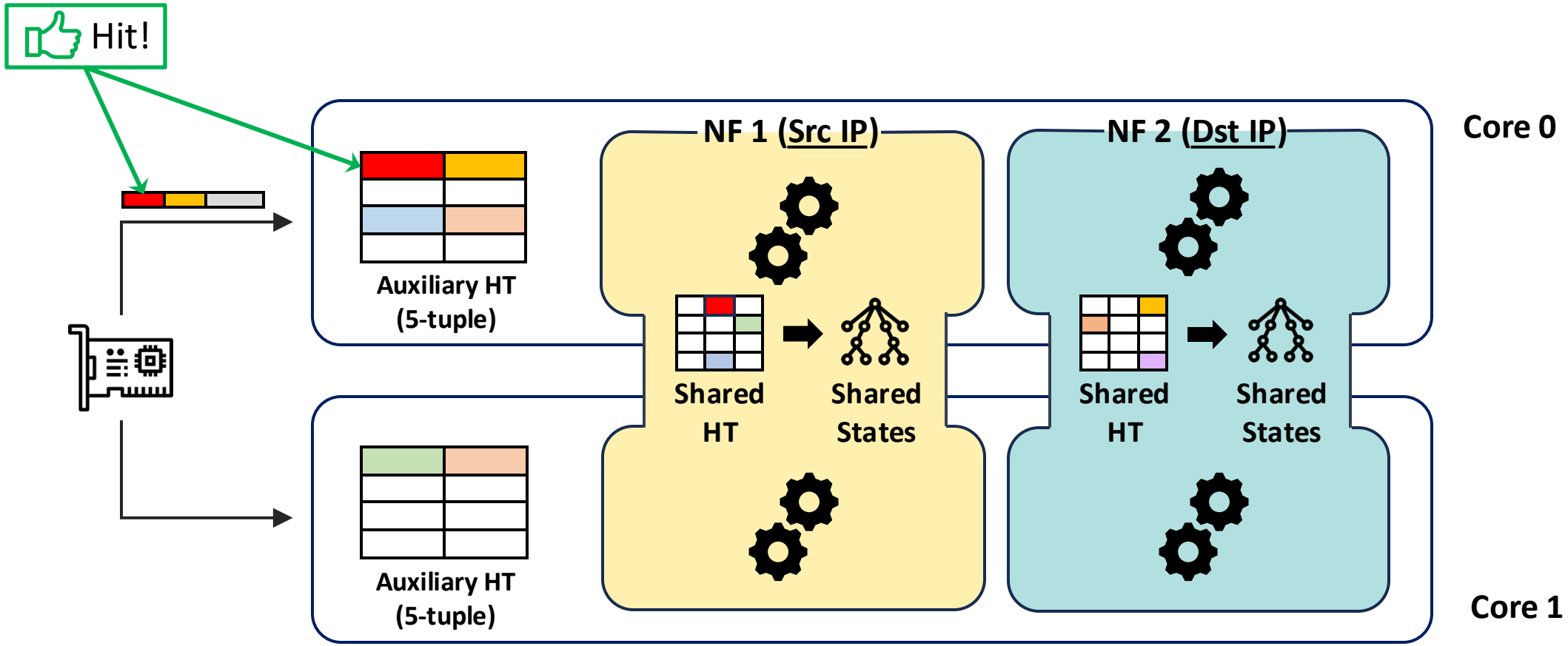
How can we **reduce** the **inevitable** sync. overhead with different flow definitions?



FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

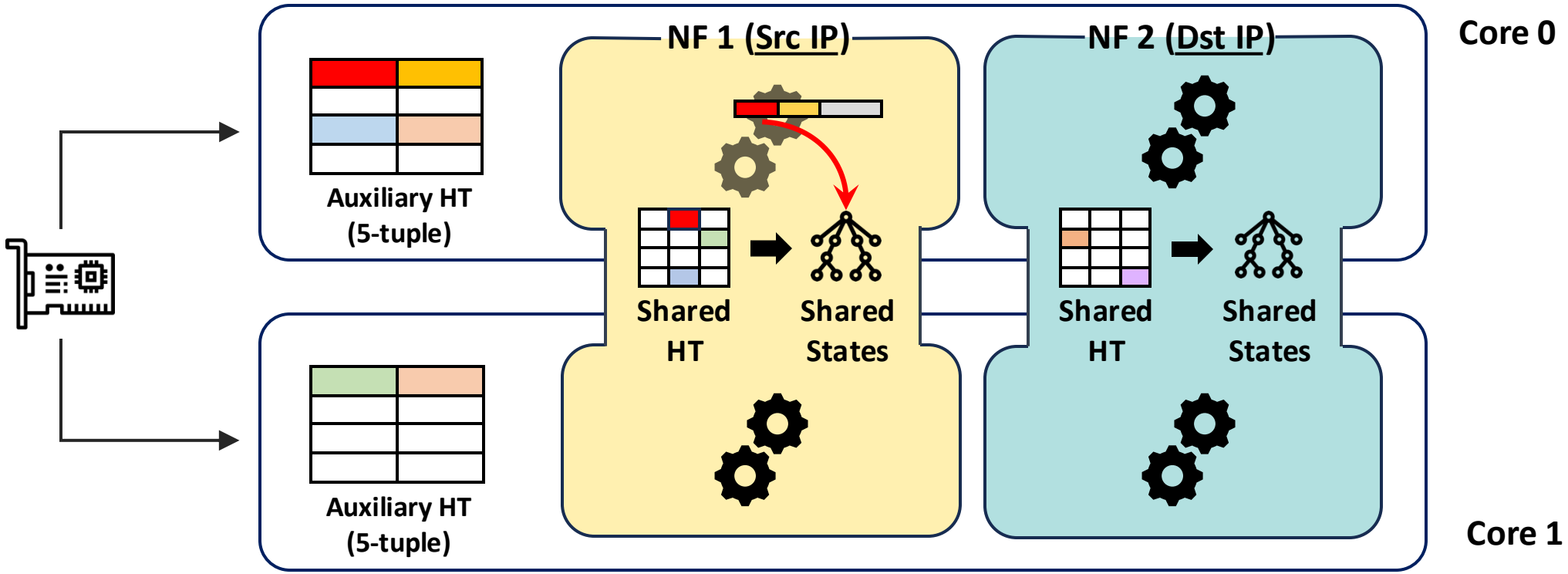
? How can we **reduce** the **inevitable** sync. overhead with different flow definitions?



FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

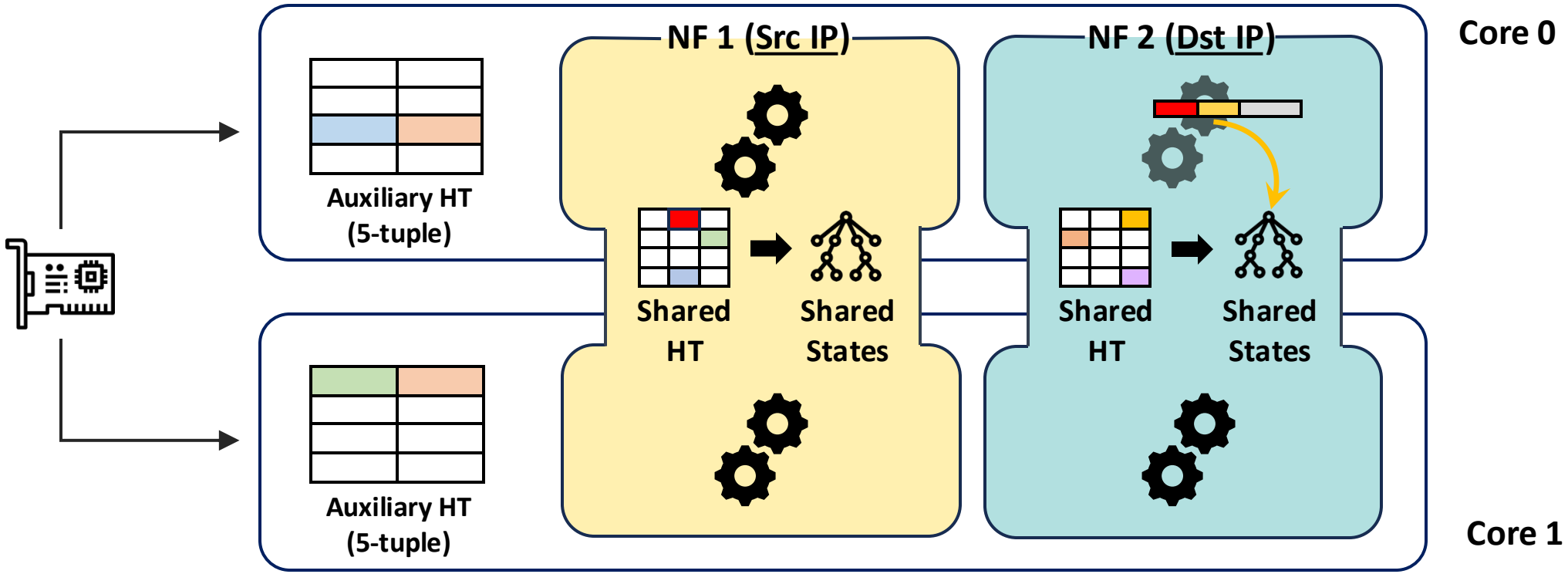
How can we **reduce** the **inevitable** sync. overhead with different flow definitions?



FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

How can we **reduce** the **inevitable** sync. overhead with different flow definitions?

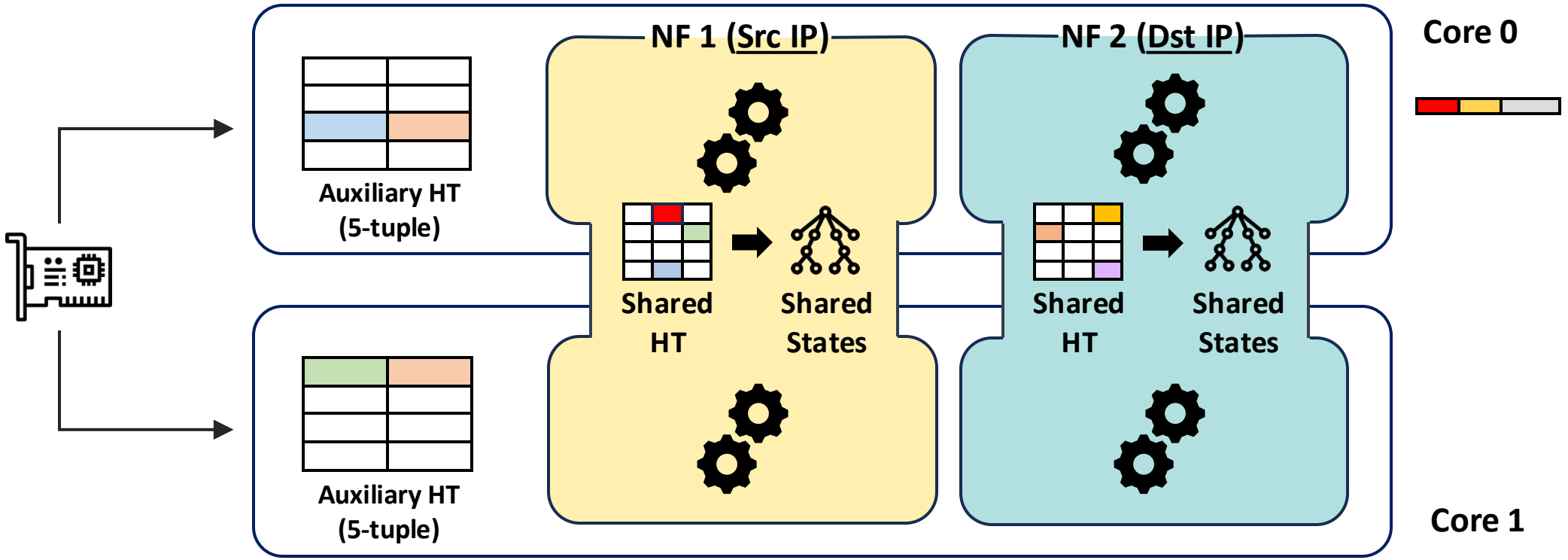


FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

How can we **reduce** the **inevitable** sync. overhead with different flow definitions?

Only a **single expensive** insert/lookup into/from shared HT → **Minimize sync. overheads**

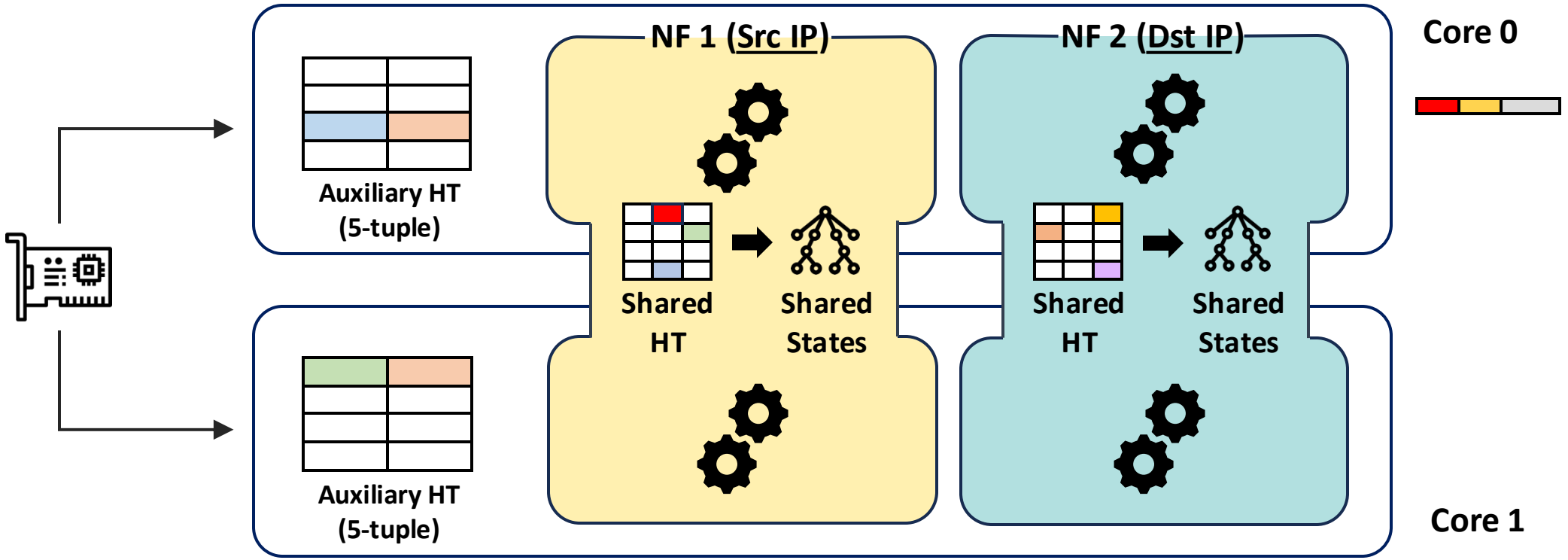


FAJITA

Batch Processing **Minimize Memory Accesses** **Minimize Sync. Overheads**

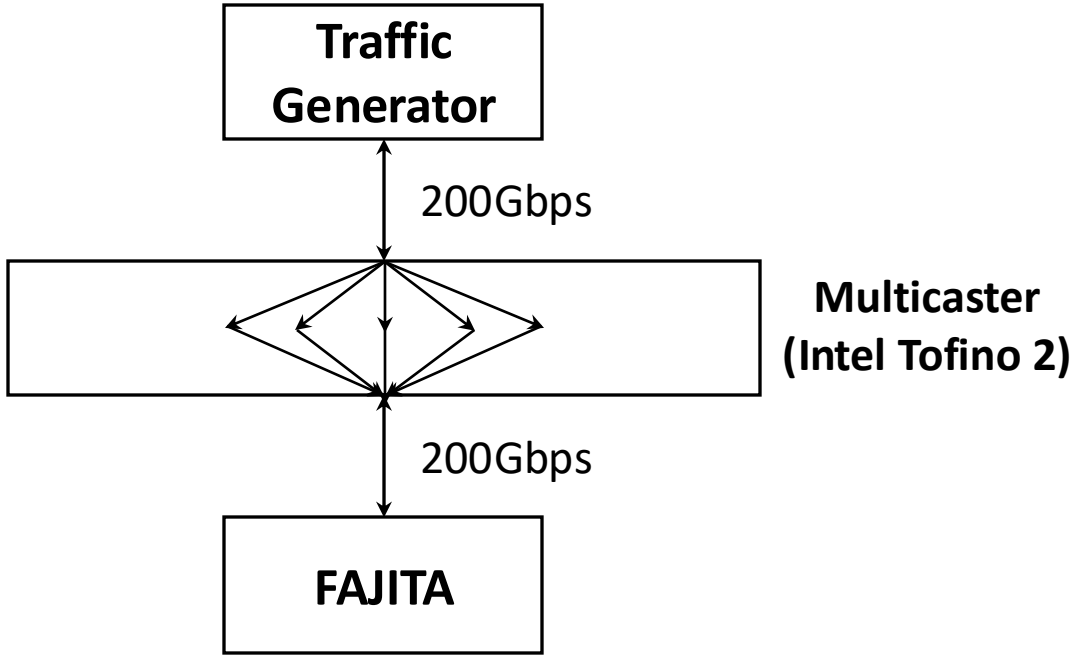
How can we **reduce** the **inevitable** sync. overhead with different flow definitions?

Only a **single expensive** insert/lookup into/from shared HT



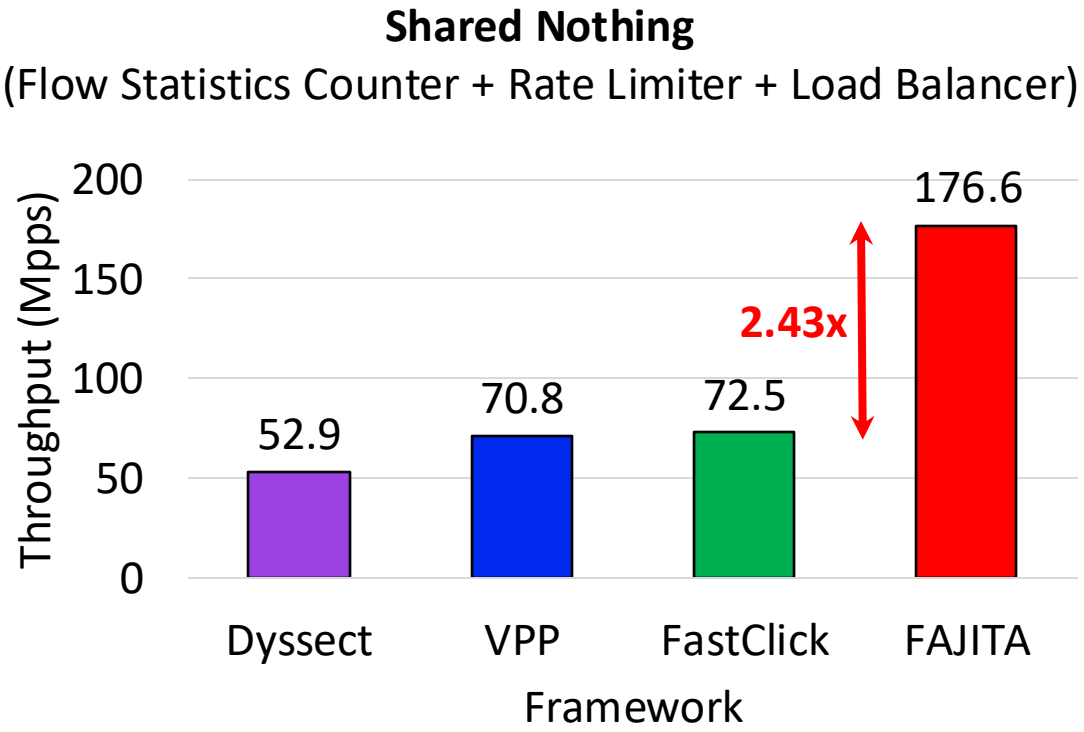
Evaluation

Testbed and Workloads



Evaluation

Does FAJITA Improve Performance?

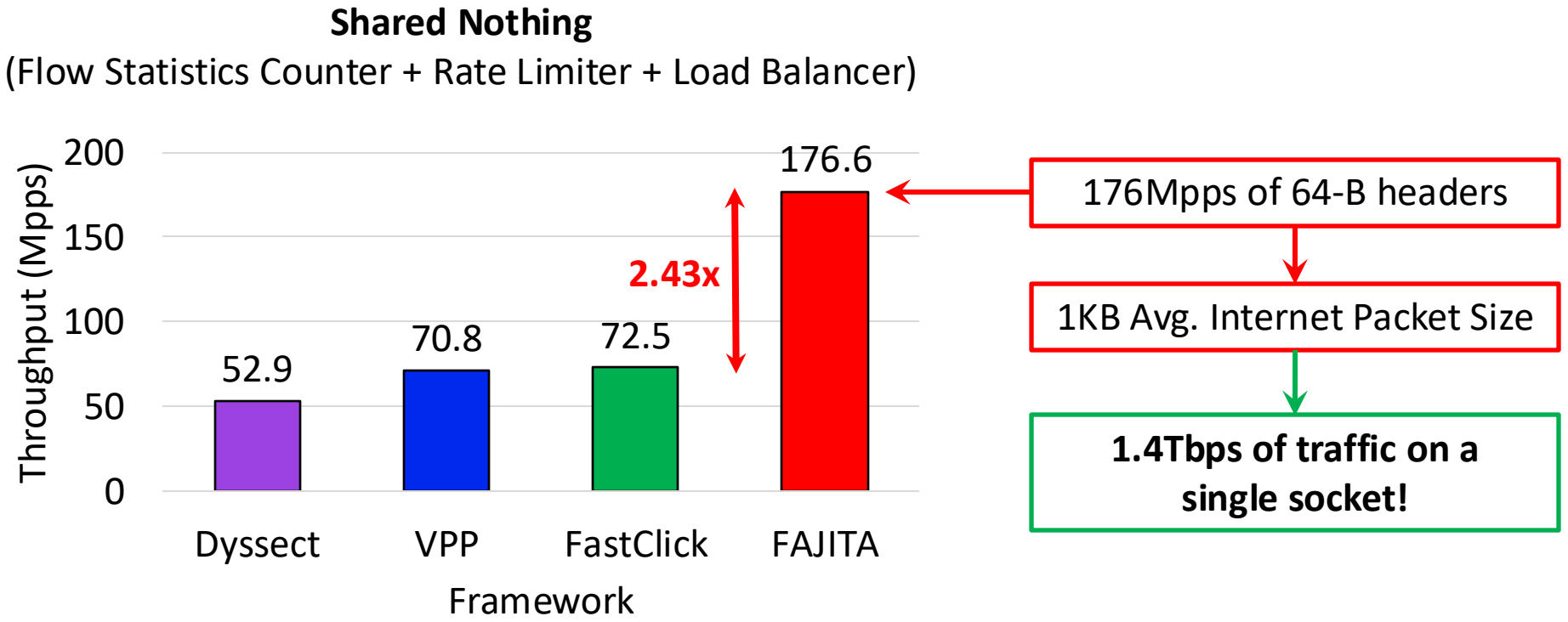


16 CPU cores

Synthetic 64-B trace w/ 2 million flows

Evaluation

Does FAJITA Improve Performance?



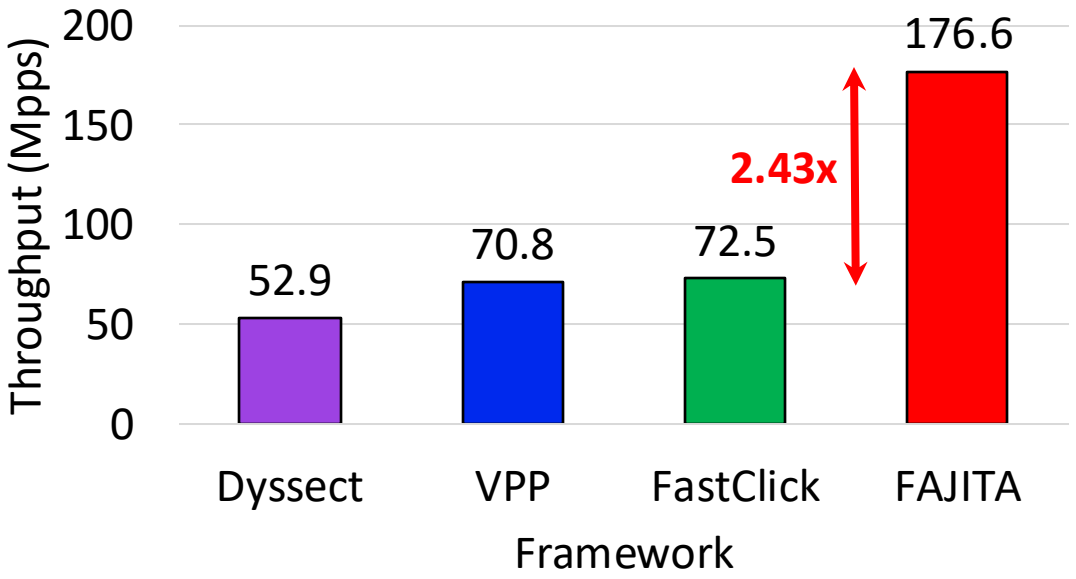
16 CPU cores

Synthetic 64-B trace w/ 2 million flows

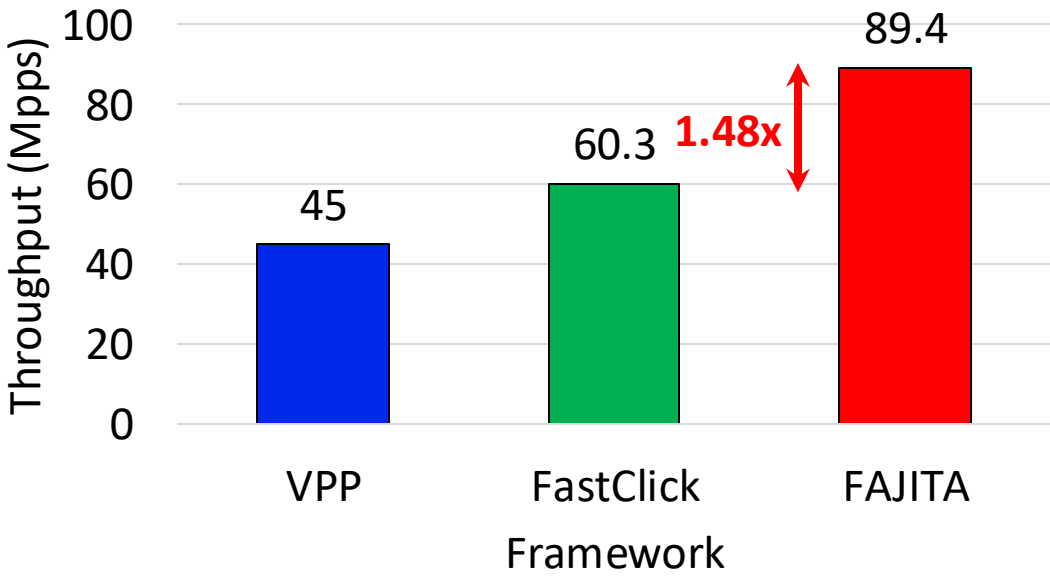
Evaluation

Does FAJITA Improve Performance?

Shared Nothing
(Flow Statistics Counter + Rate Limiter + Load Balancer)



Shared
(Src IP Counter + Policer + Load Balancer)



16 CPU cores

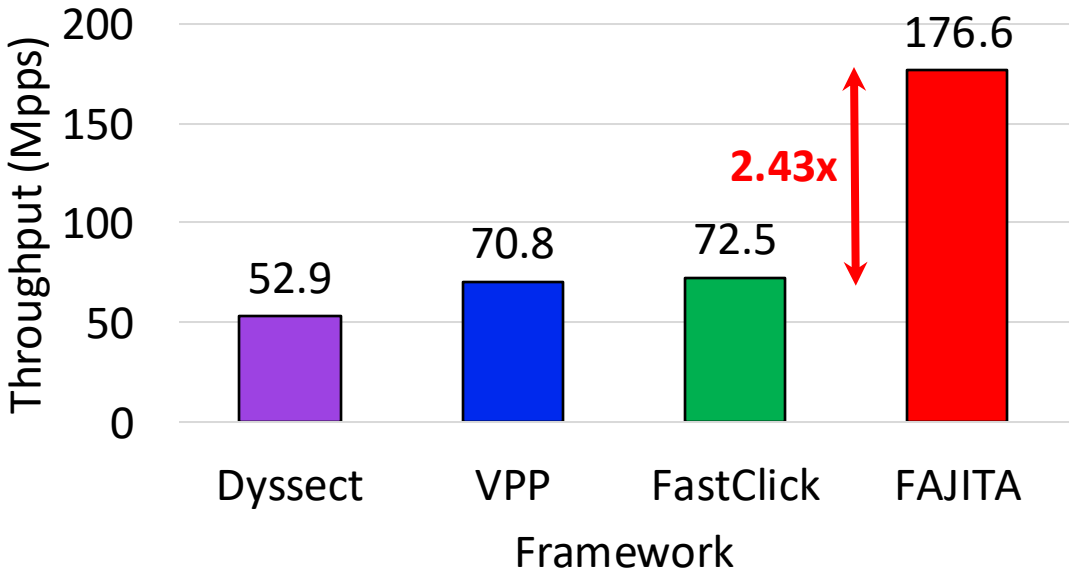
Synthetic 64-B trace w/ 2 million flows

Evaluation

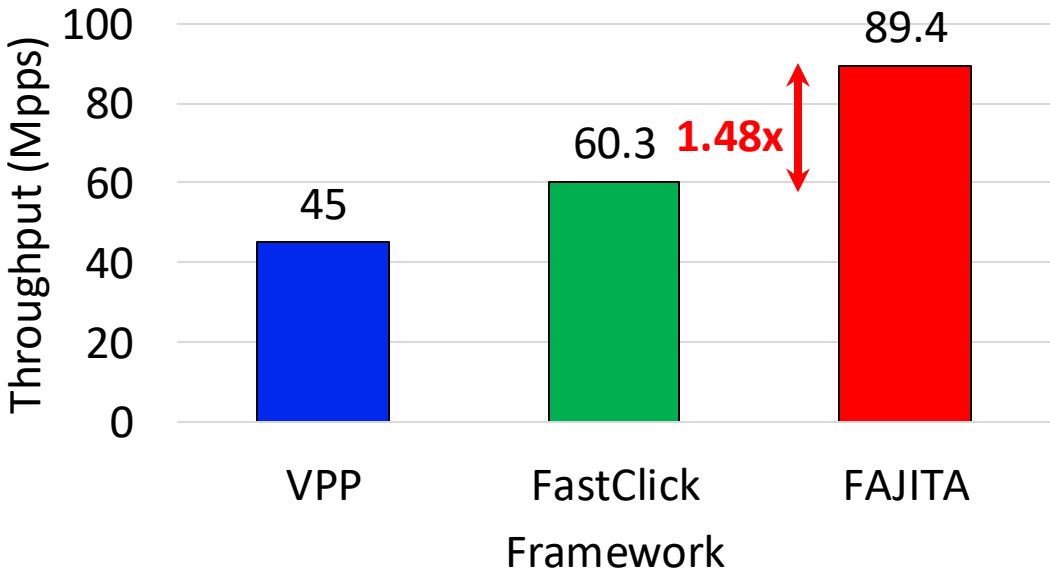
Does FAJITA Improve Performance?

FAJITA is the first system to enable **>100Mpps stateful NF chains** on a **single socket**!

Shared Nothing
(Flow Statistics Counter + Rate Limiter + Load Balancer)

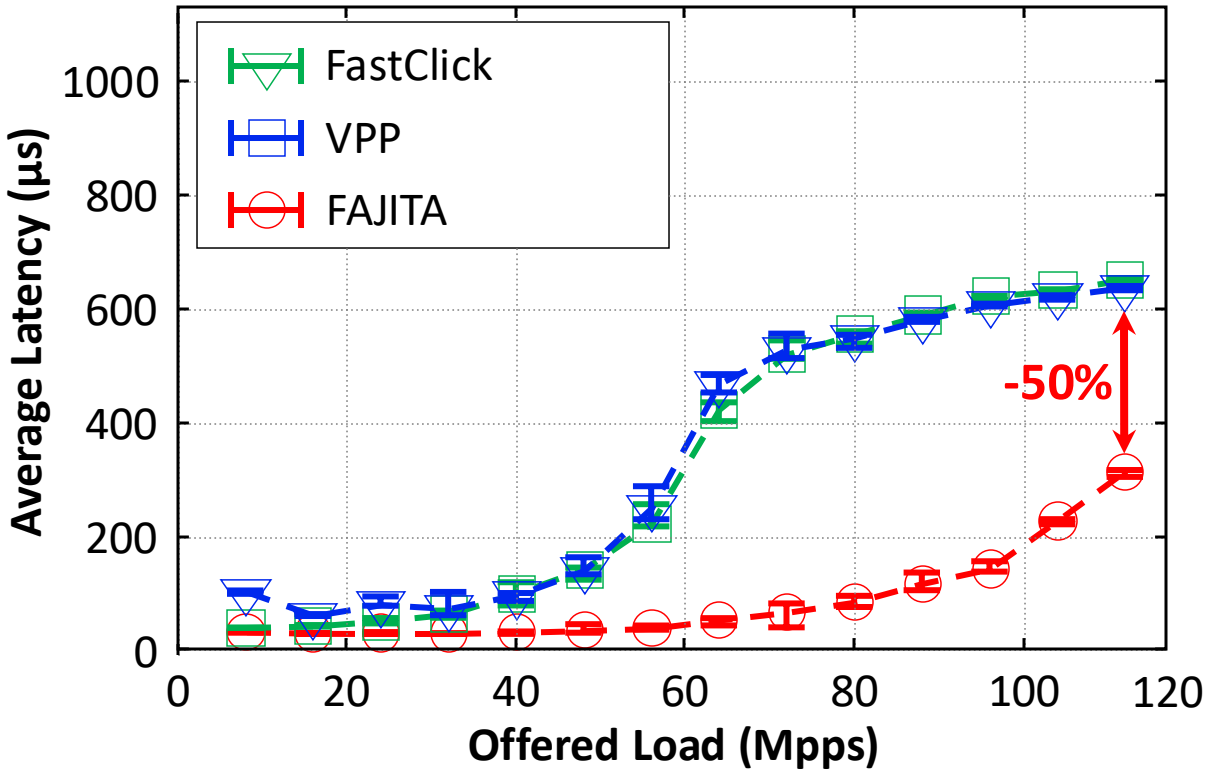


Shared
(Src IP Counter + Policer + Load Balancer)



Evaluation

Does FAJITA Improve End-to-End Latency?

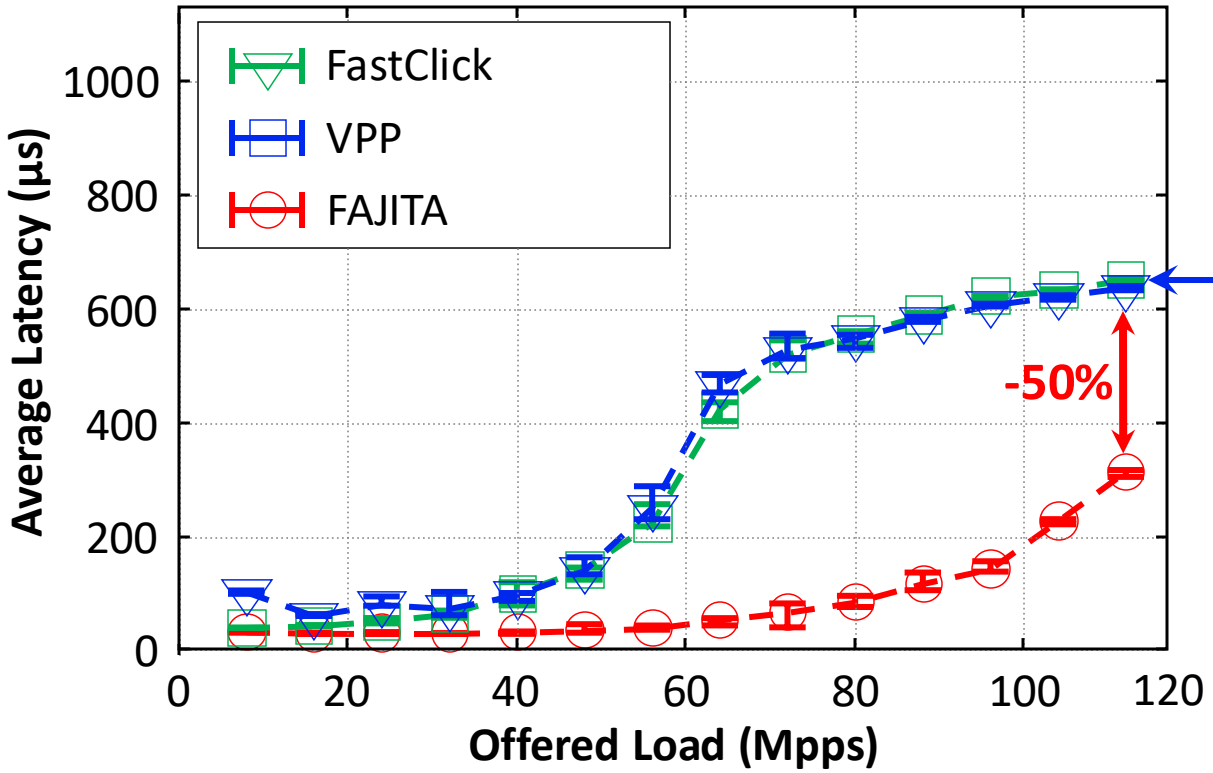


8 CPU cores

Load Balancer + Flow Statistics Counter

Evaluation

Does FAJITA Improve End-to-End Latency?



One of the three principles is **overlooked**

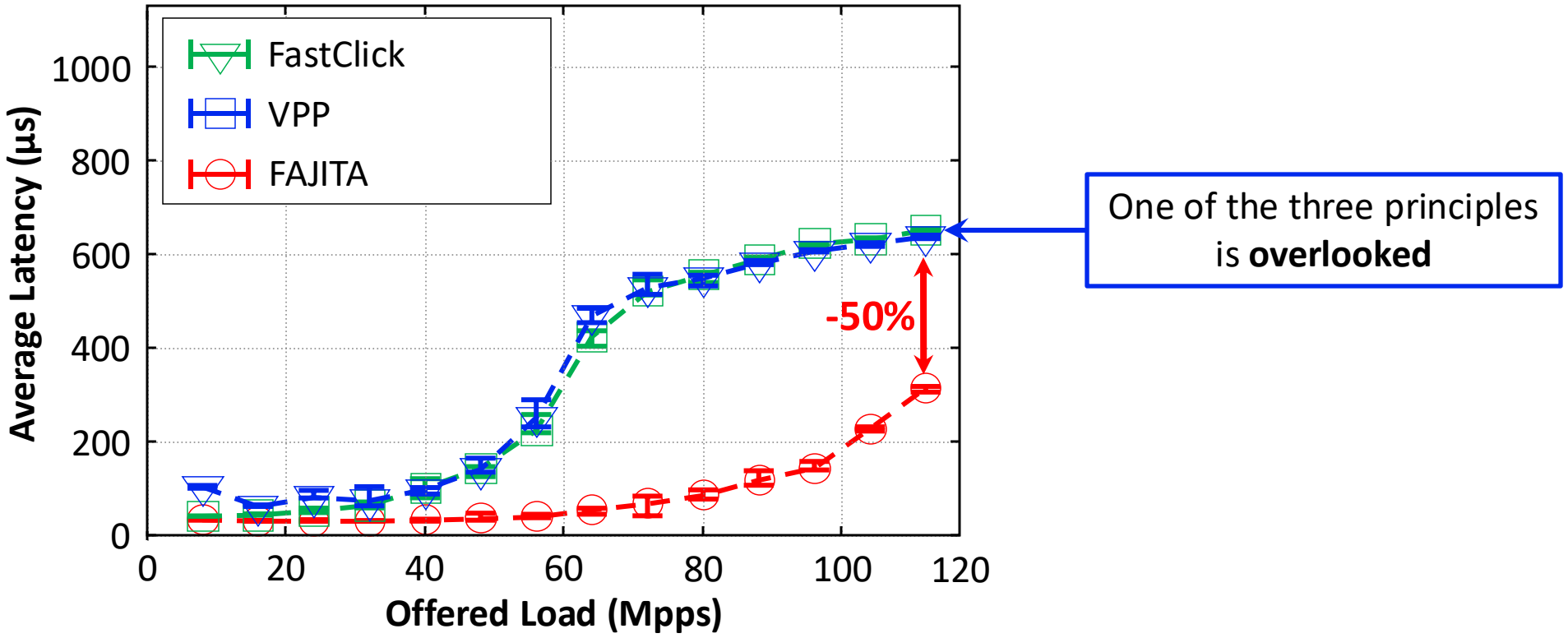
8 CPU cores

Load Balancer + Flow Statistics Counter

Evaluation

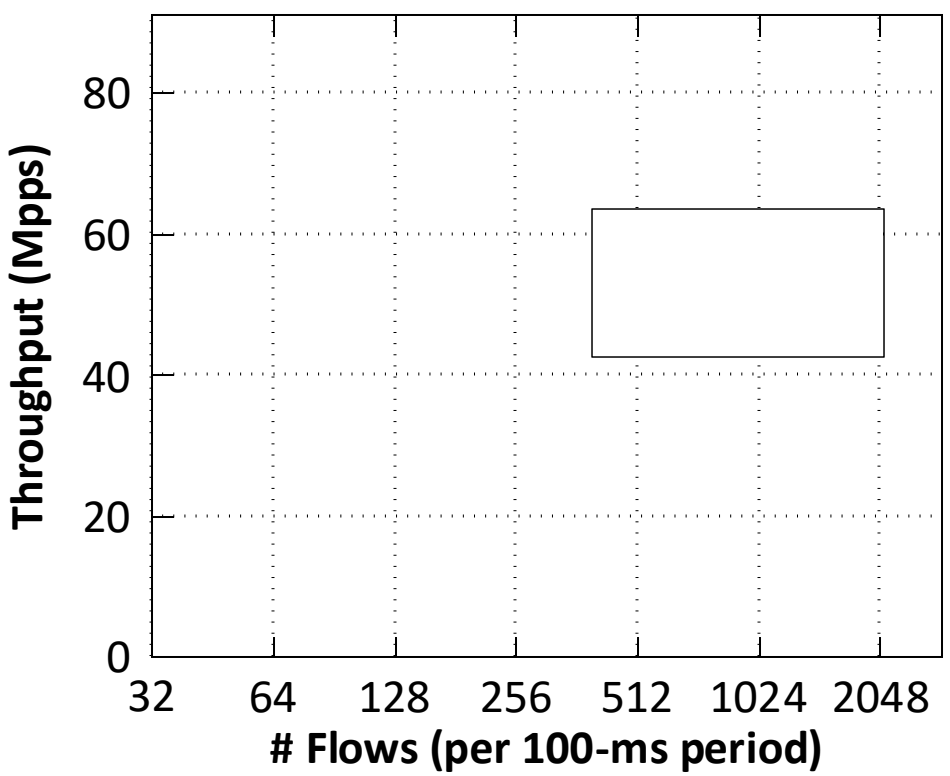
Does FAJITA Improve End-to-End Latency?

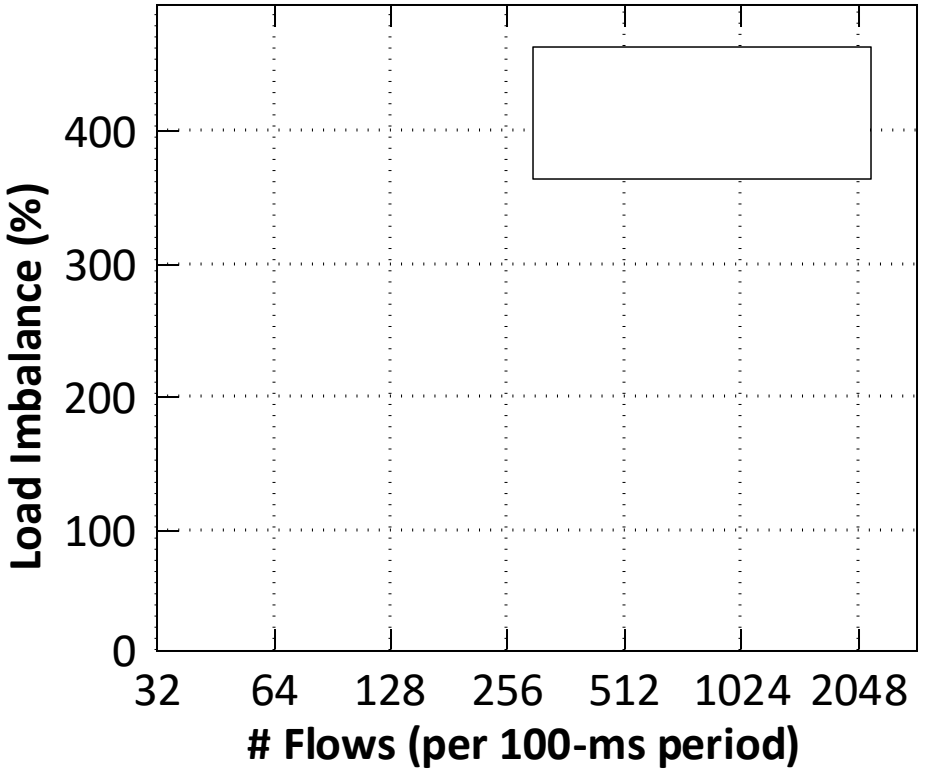
FAJITA performs stateful packet processing in **half the time** of other frameworks



Evaluation

Is RSS Sufficient for FAJITA?



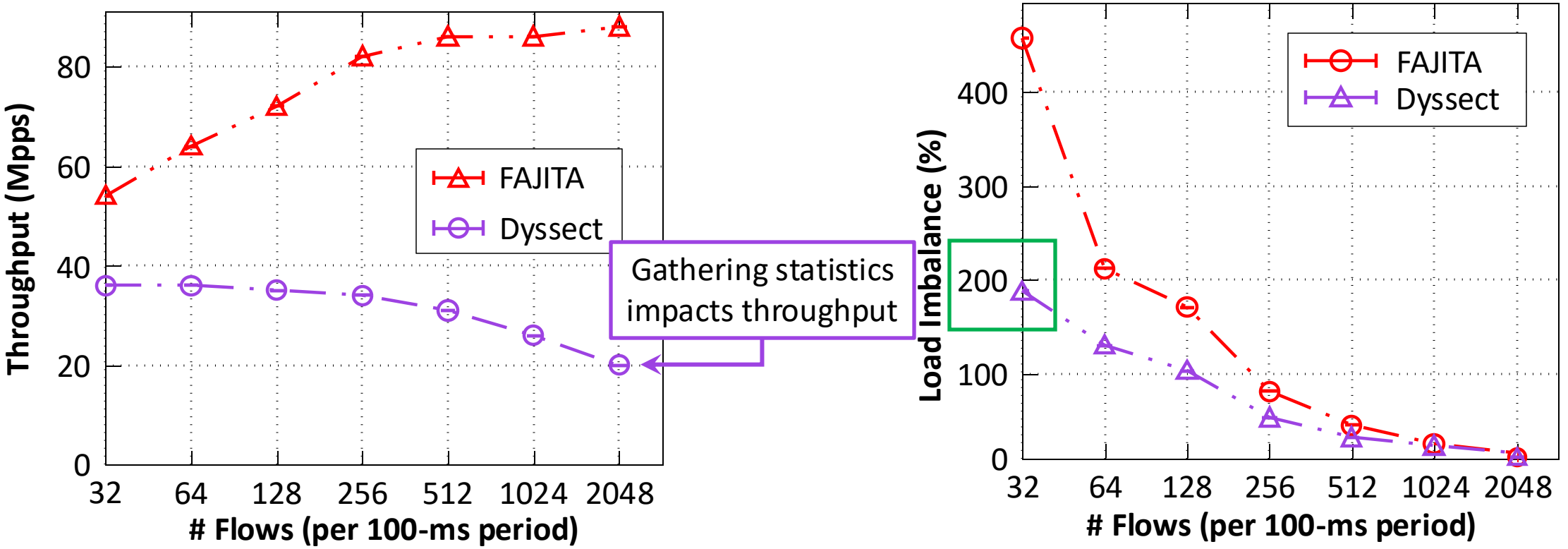


8 CPU cores – Synthetic Trace w/ Elephants Only

Flow Statistics Counter – 100ms Monitoring Interval

Evaluation

Is RSS Sufficient for FAJITA?

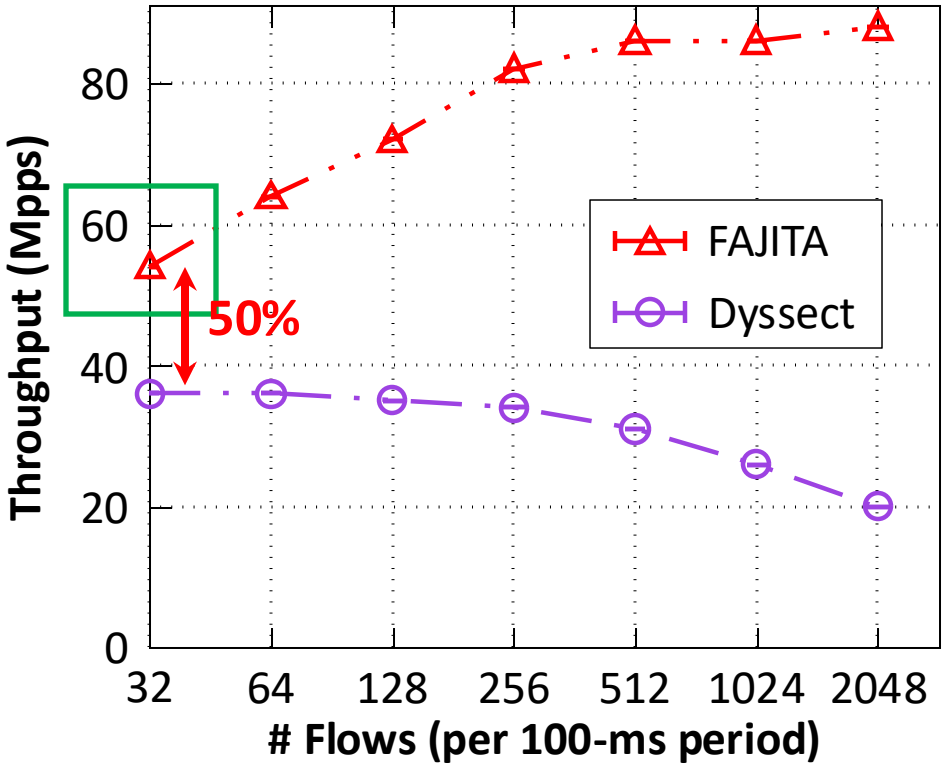


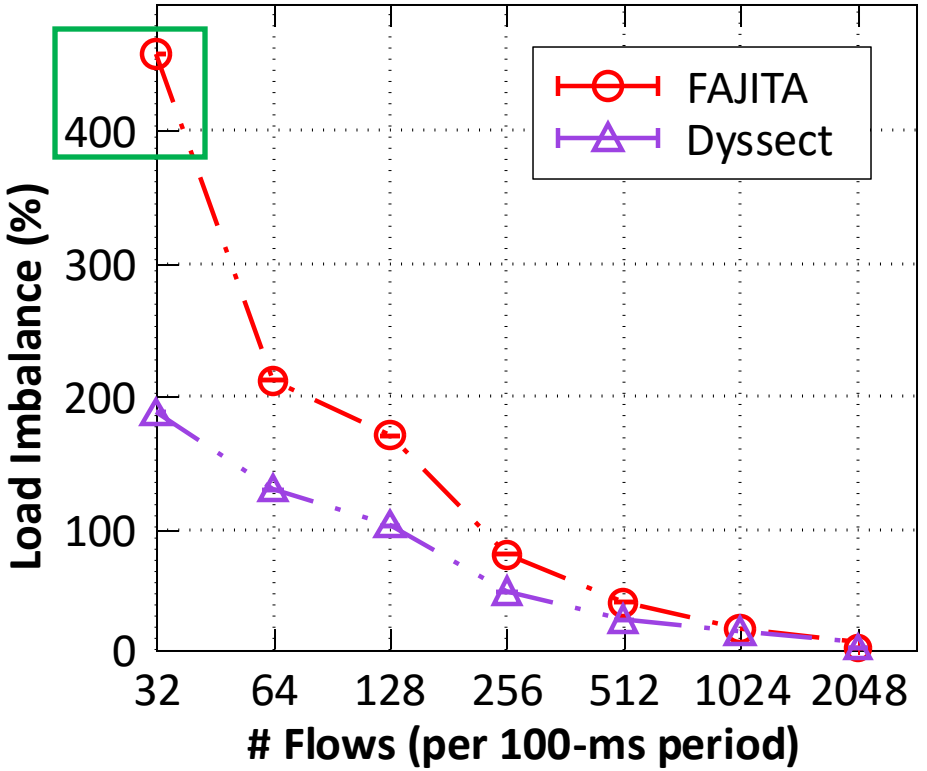
8 CPU cores – Synthetic Trace w/ Elephants Only

Flow Statistics Counter – 100ms Monitoring Interval

Evaluation

Is RSS Sufficient for FAJITA?





8 CPU cores – Synthetic Trace w/ Elephants Only

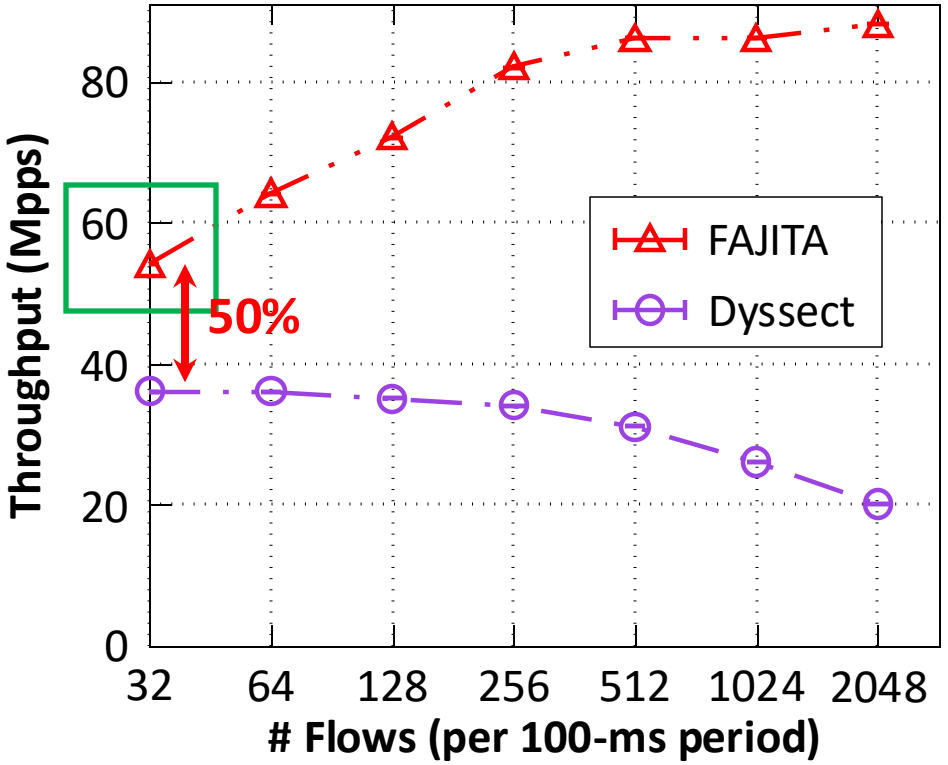
Flow Statistics Counter – 100ms Monitoring Interval

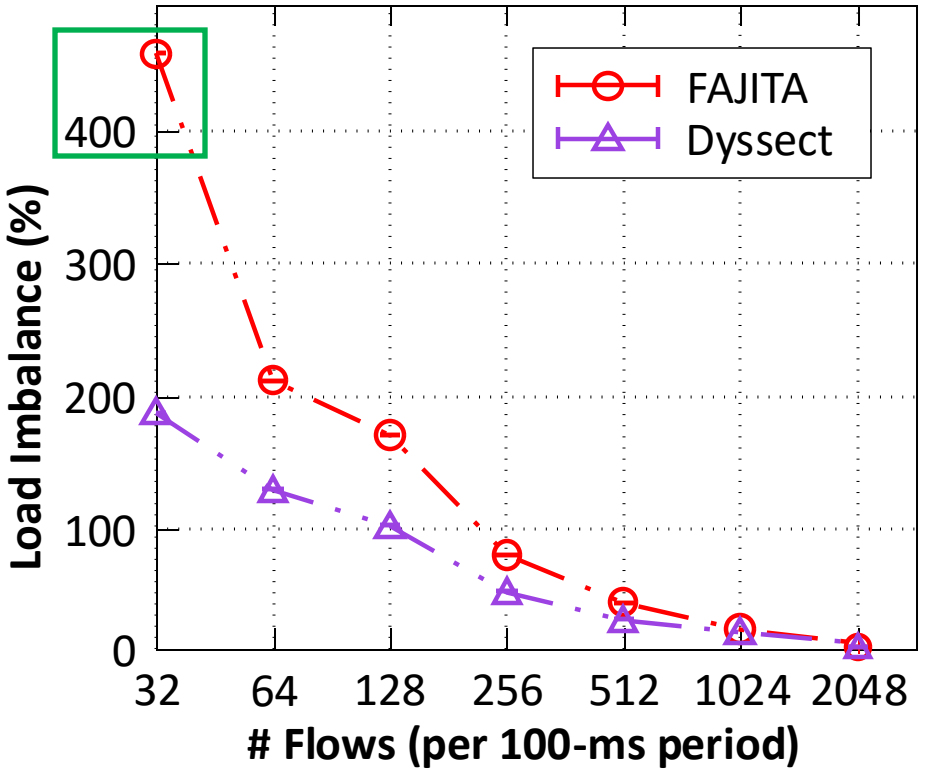
Evaluation

Is RSS Sufficient for FAJITA?

FAJITA achieves better performance **solely relying on RSS**

Load imbalance **only** depends on the number of flows, that are usually **>2K at high rates**!





FAJITA take-away

The **FAJITA** system:

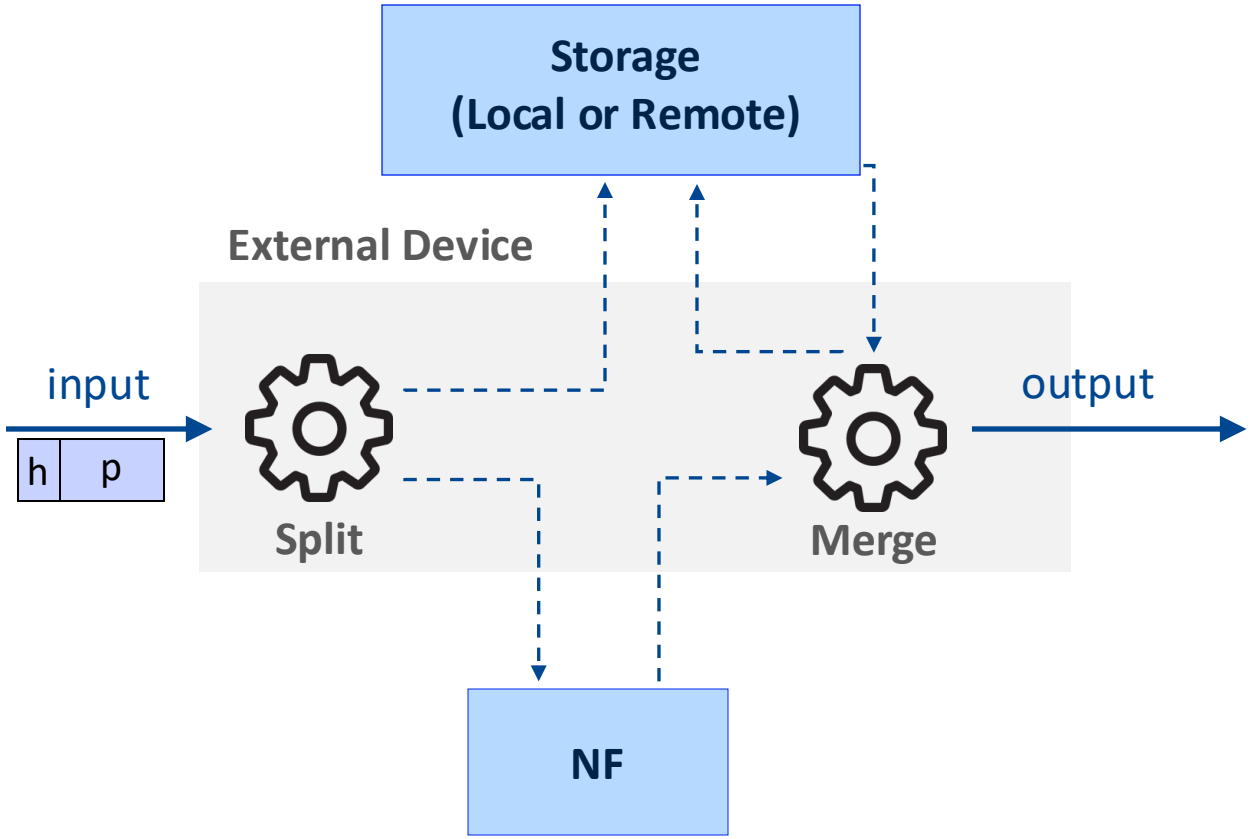
- **incorporates** the three essential principles for stateful packet processing **together**
- **minimizes** memory access overheads by **exploiting batching** & **software prefetching**
- **alleviates** overheads of accessing shared data structures **by introducing auxiliary HTs**
- achieves **performance 2x higher** than existing stateful packet processors
- demonstrates that dynamic inter-core load balancing is **detrimental** at high rates
- only processes packet headers

How to split packets?

How to split packets?

State-of-the-art **packet-splitting** solutions **increase** shallow NFs throughput

e.g., Ribosome, nicmem



Packet-Splitting Approaches

State-of-the-art packet-splitting solutions increase NFs throughput

Improve NF Performance

Better CPU cache exploitation

Free up Bandwidth

Reduce the number of dedicated NF servers

Lower energy consumption

Complex Deployment

Require end-host modifications

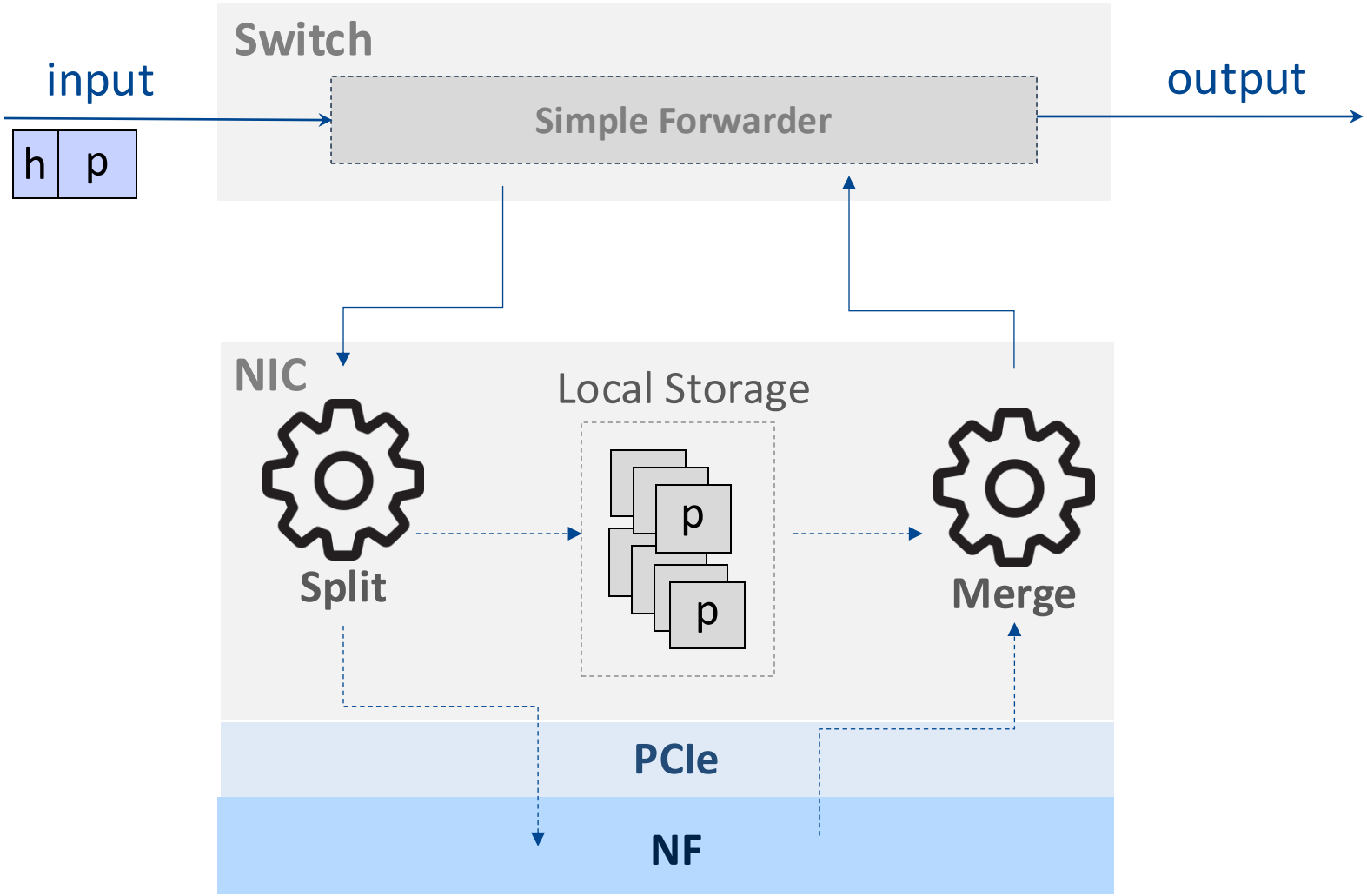
Require available shared resources

Packet-Splitting Approaches

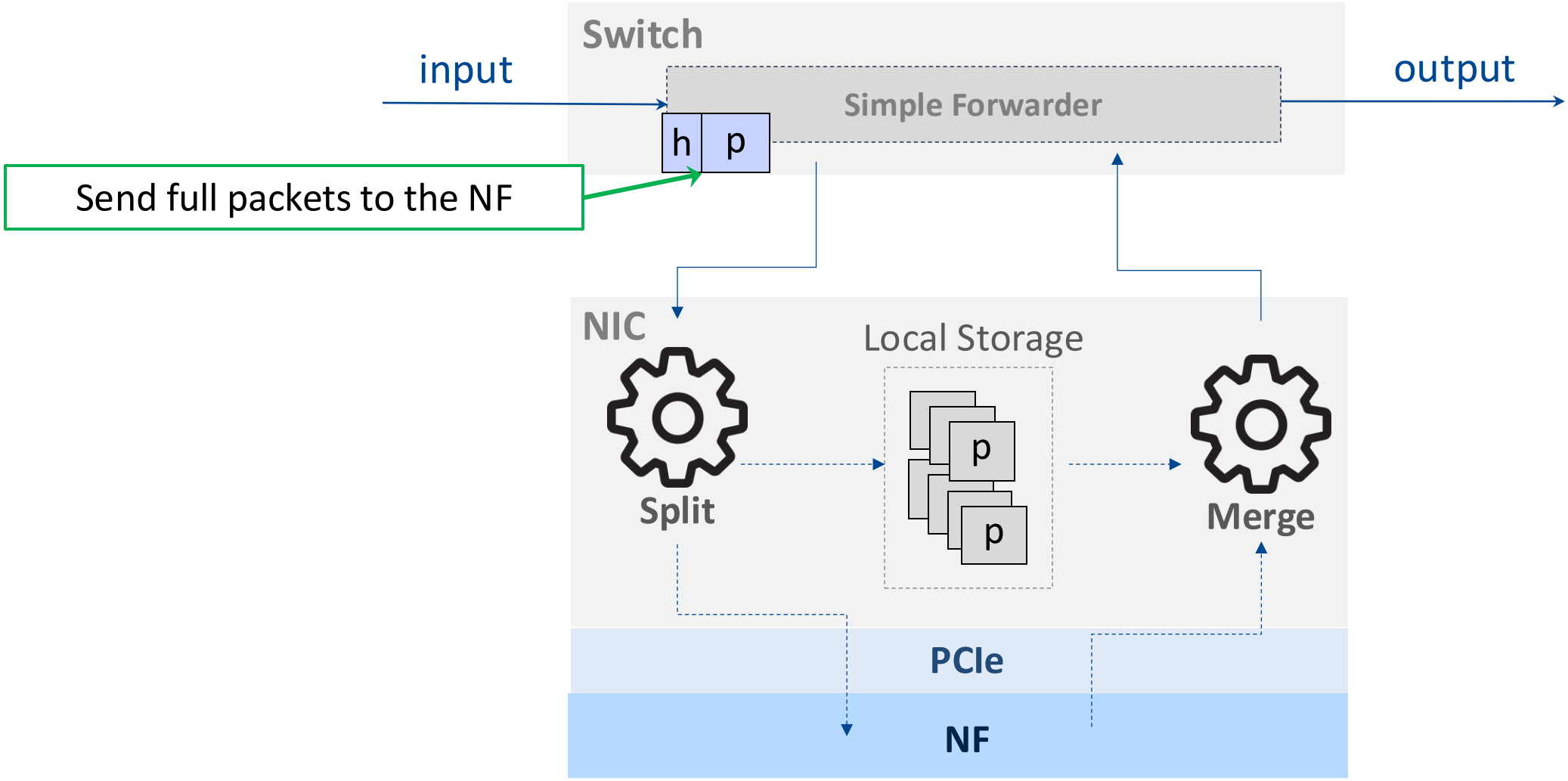
State-of-the-art packet-splitting solutions increase NFs throughput

Storing Payloads in the On-NIC Memory (nicmem) 

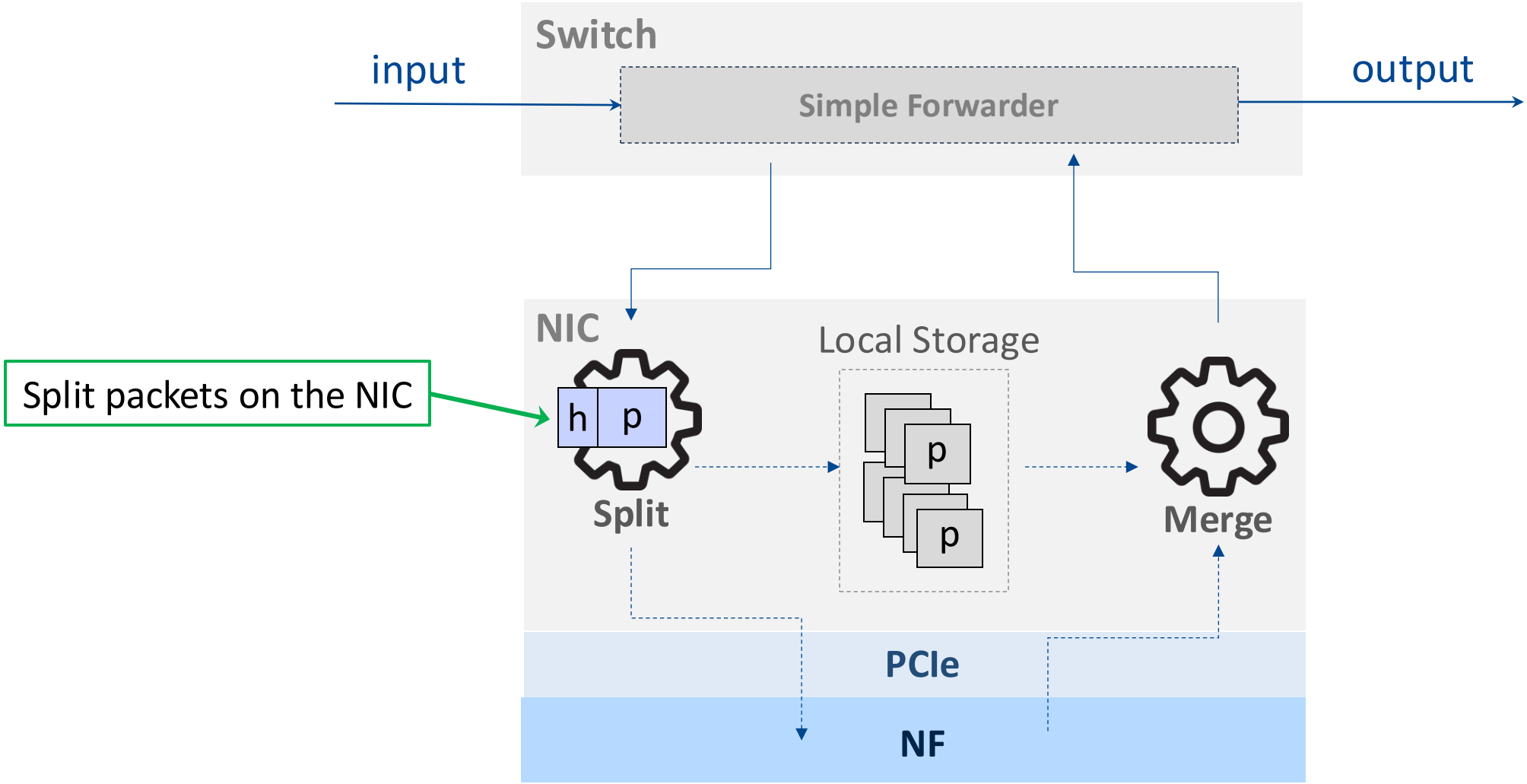
Storing Payloads in the On-NIC Memory (nicmem)



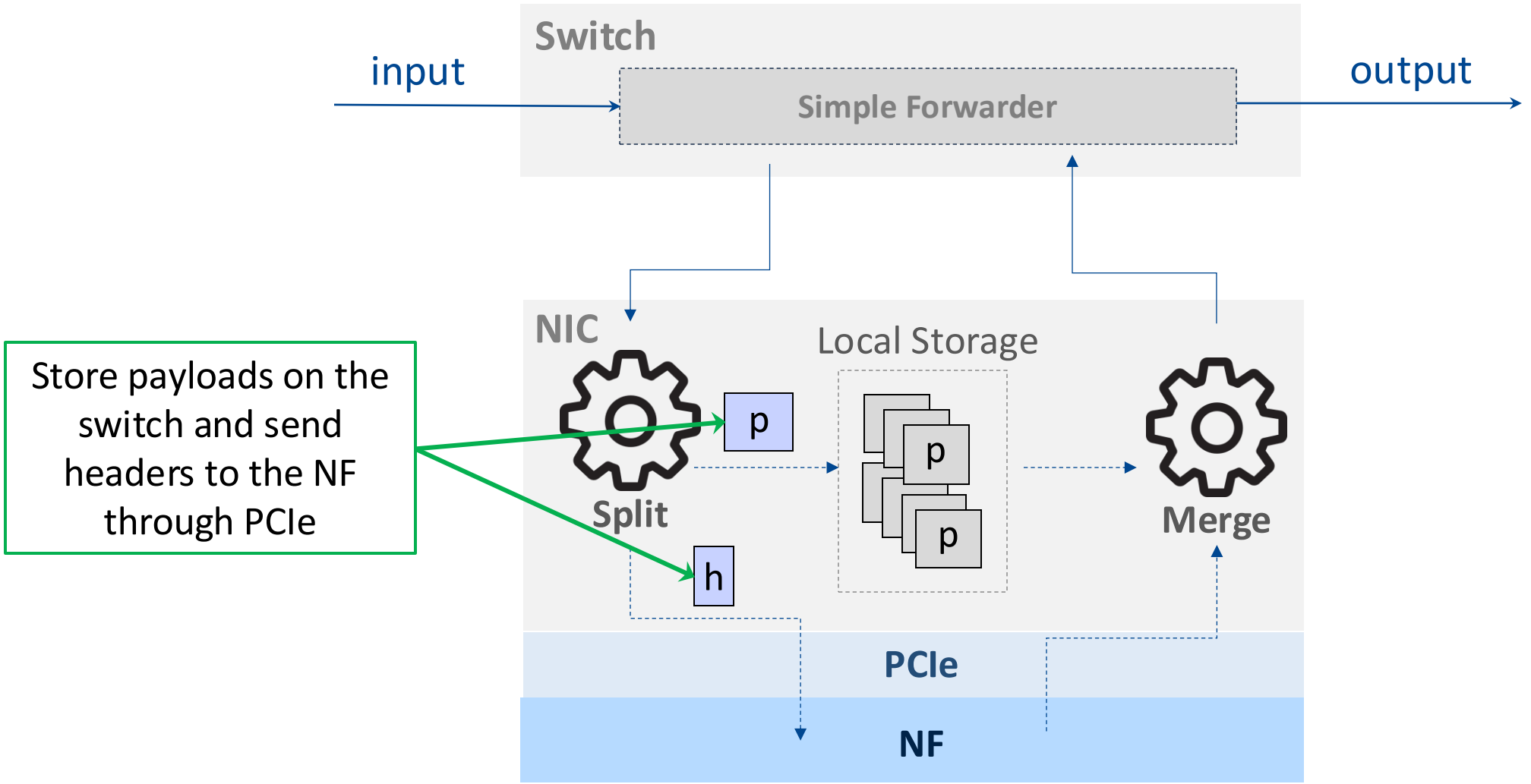
Storing Payloads in the On-NIC Memory (nicmem)



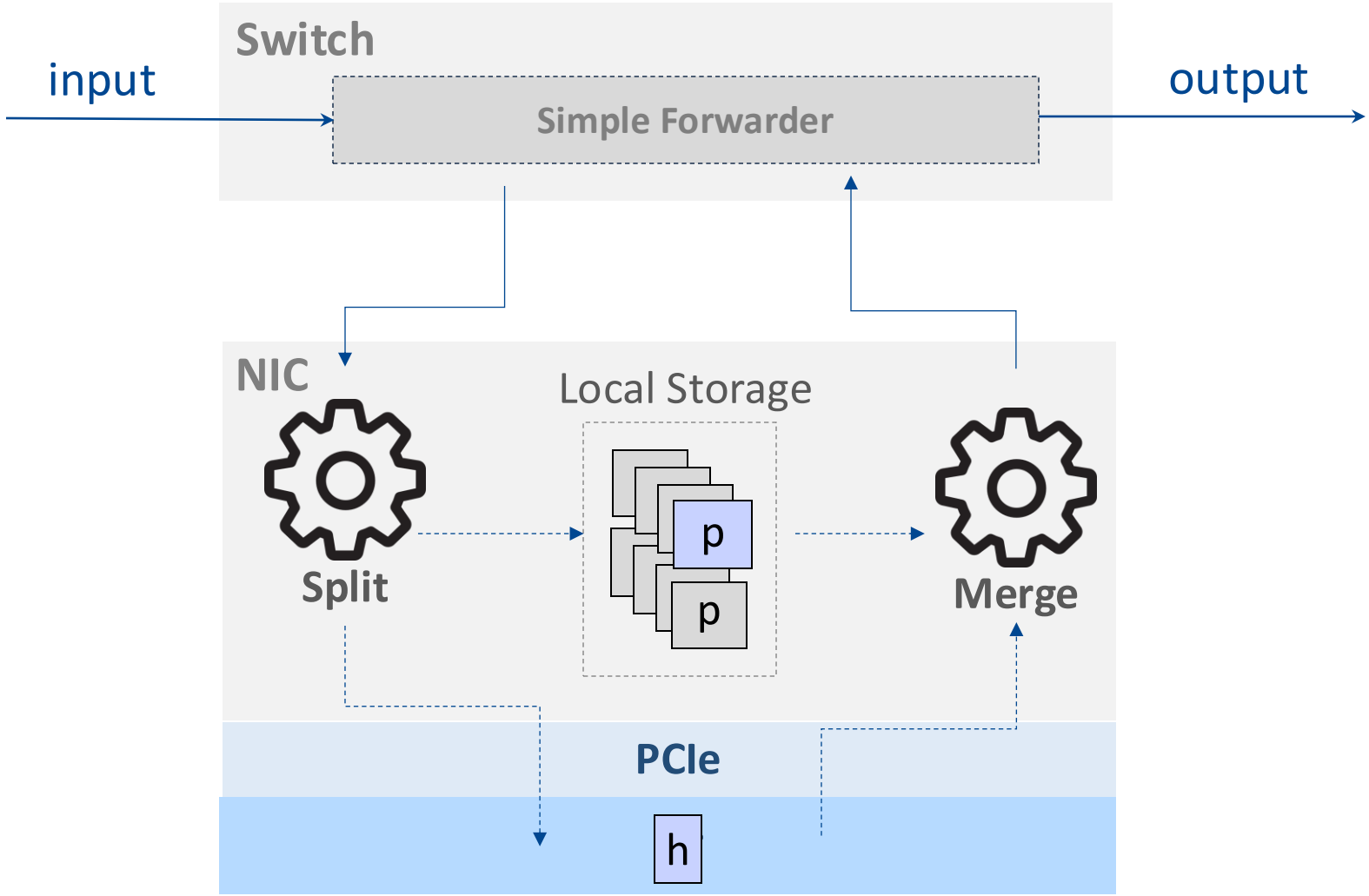
Storing Payloads in the On-NIC Memory (nicmem)



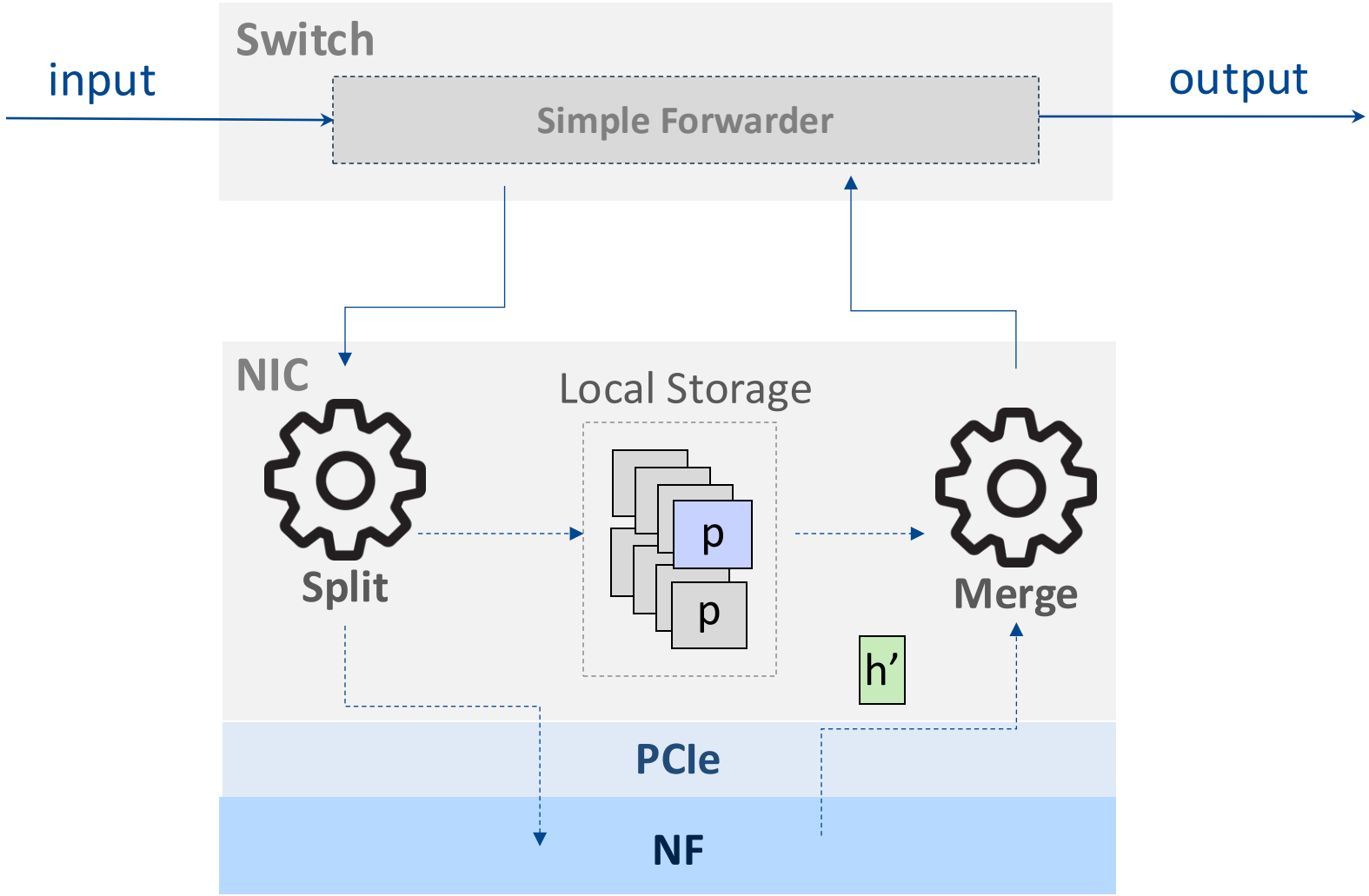
Storing Payloads in the On-NIC Memory (nicmem)



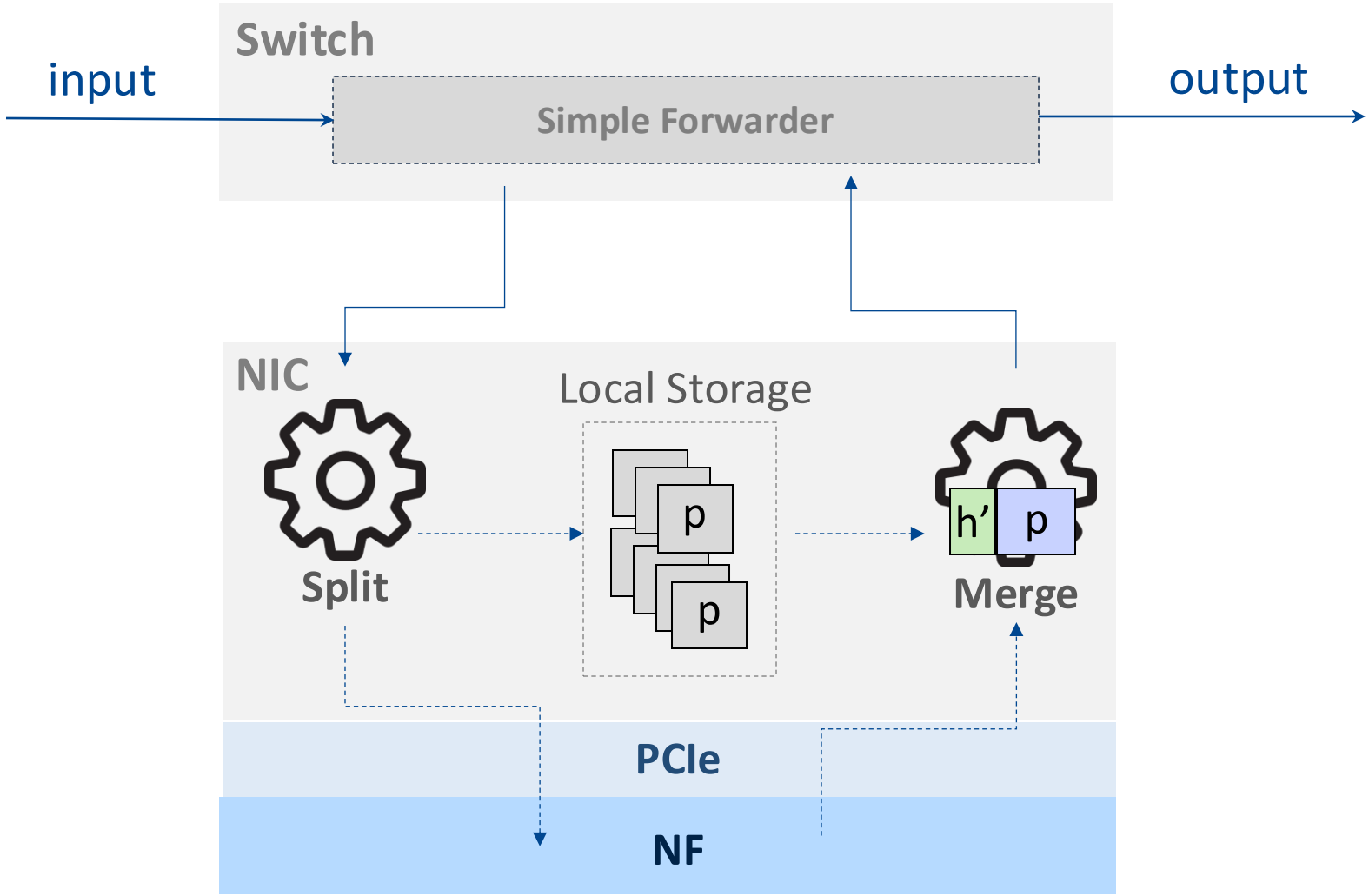
Storing Payloads in the On-NIC Memory (nicmem)



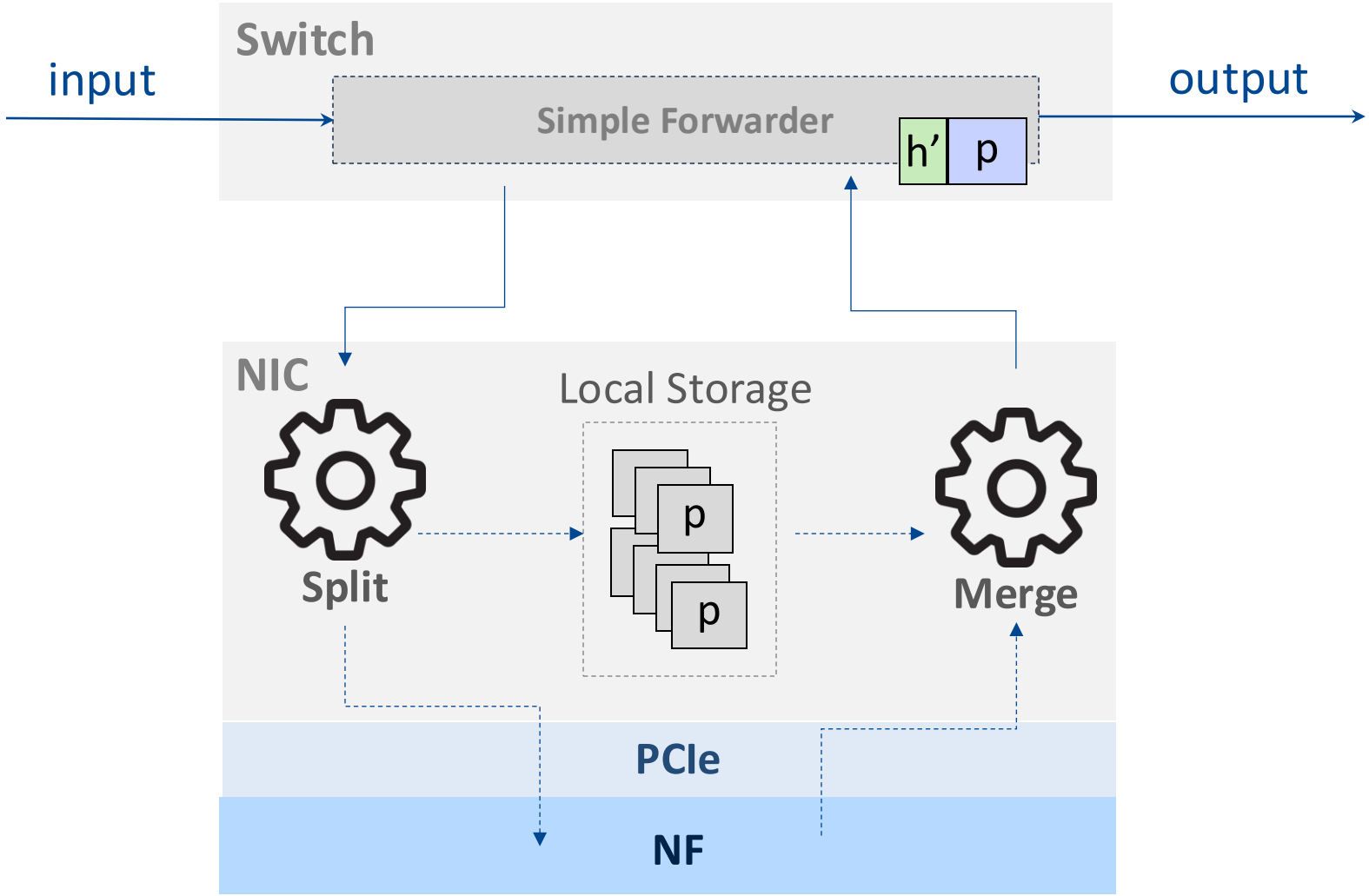
Storing Payloads in the On-NIC Memory (nicmem)



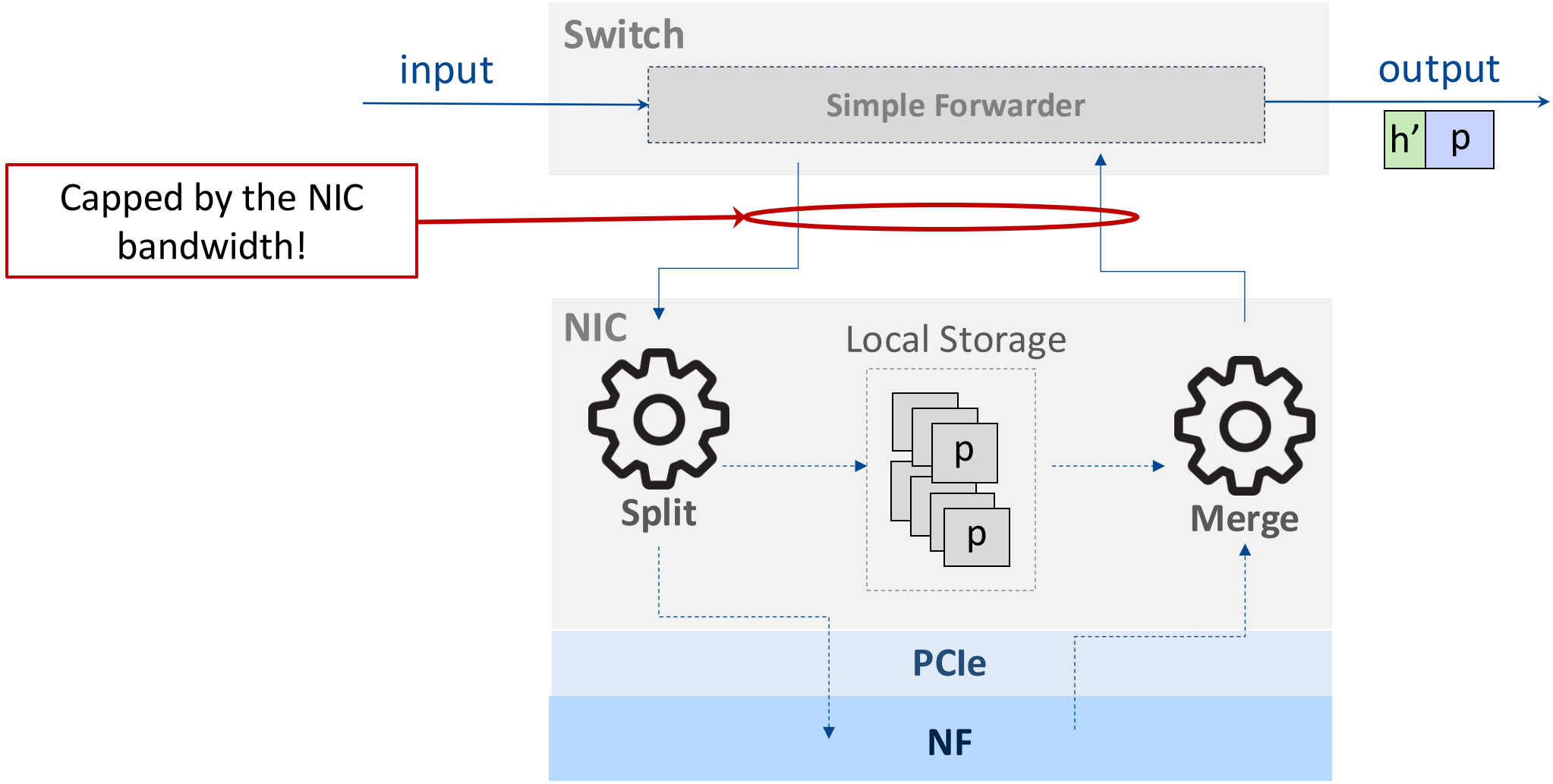
Storing Payloads in the On-NIC Memory (nicmem)



Storing Payloads in the On-NIC Memory (nicmem)



Storing Payloads in the On-NIC Memory (nicmem)



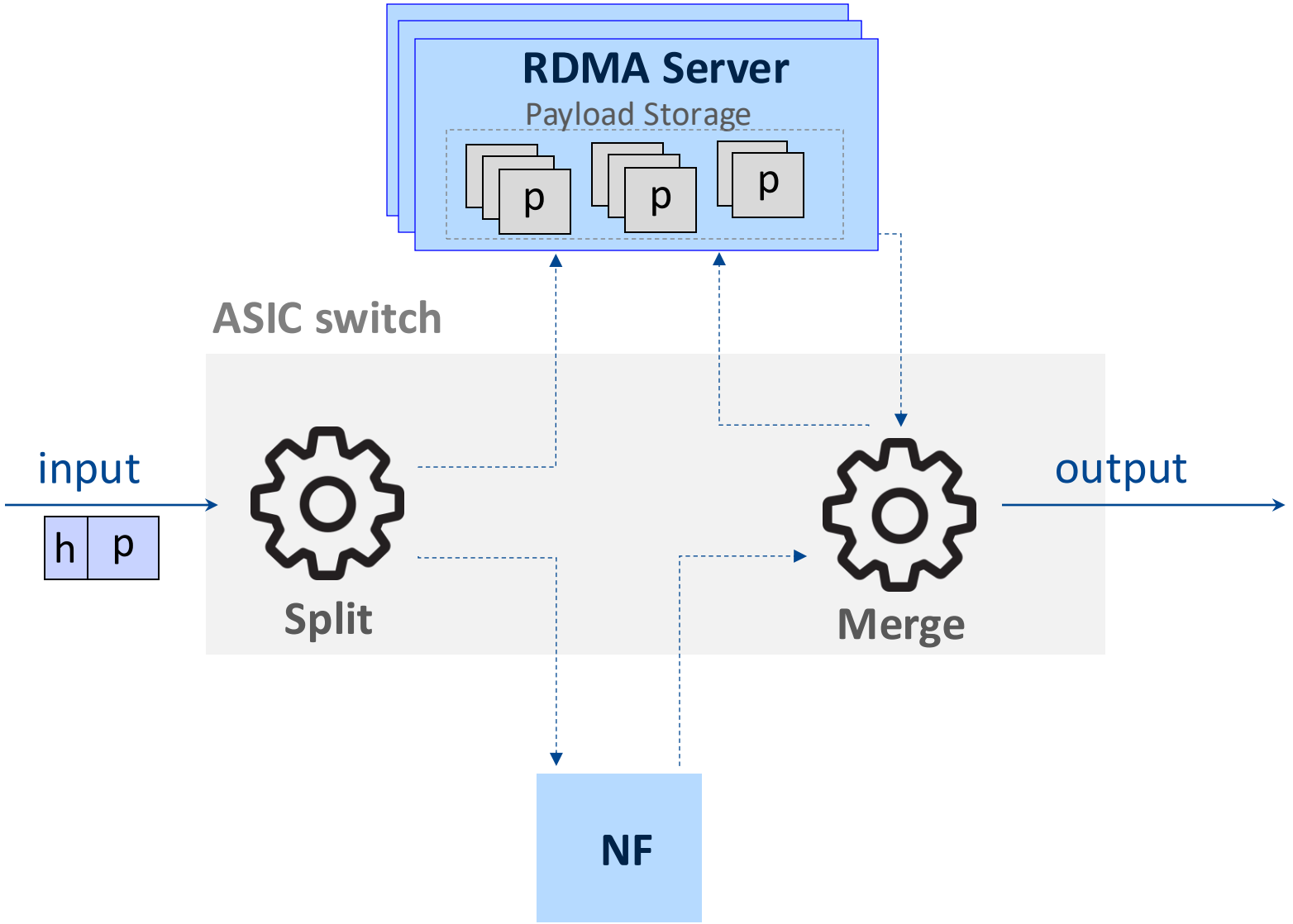
Packet-Splitting Approaches

State-of-the-art packet-splitting solutions increase NFs throughput

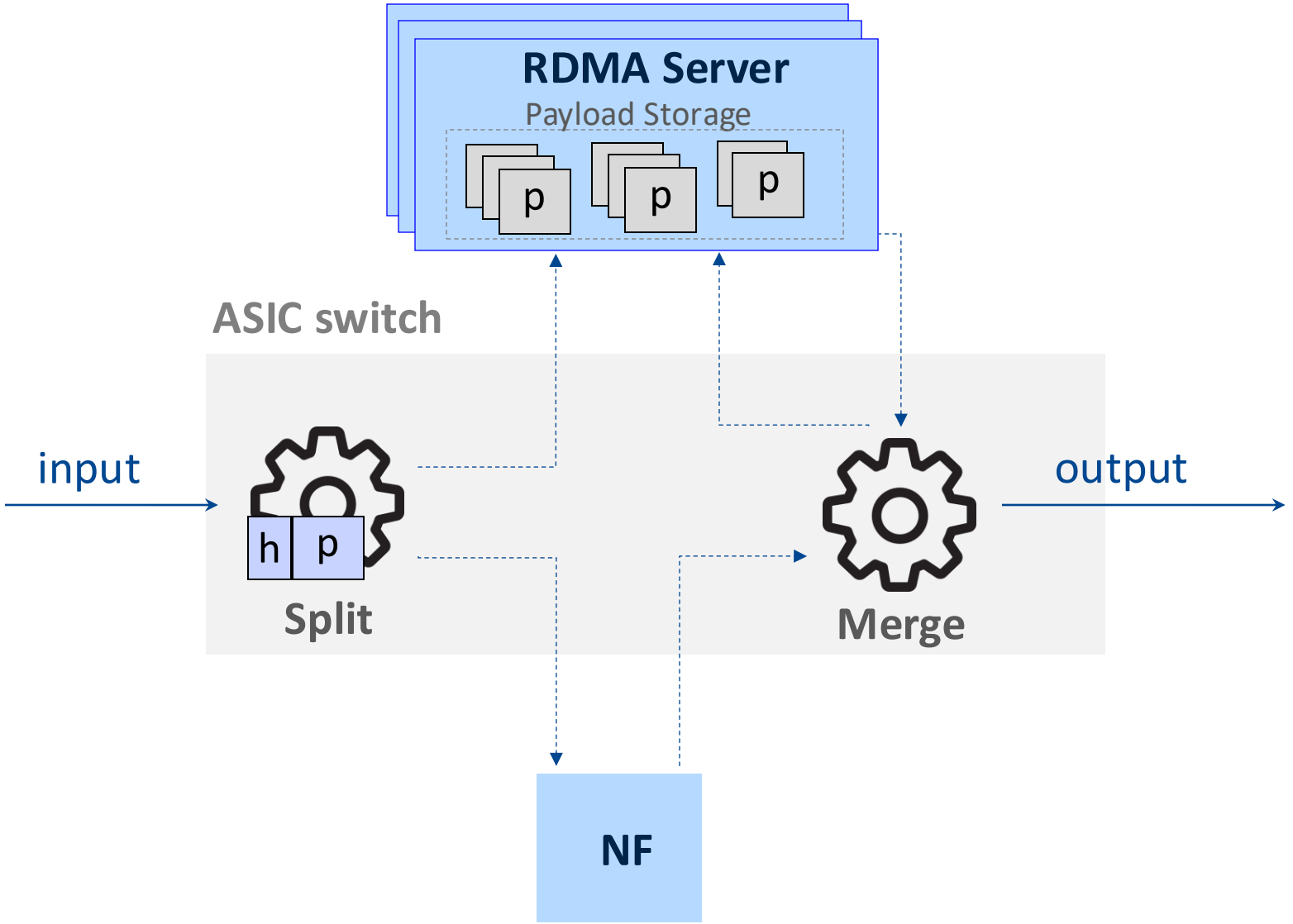
Storing Payloads in the On-NIC Memory (nicmem) 

Storing Payloads on Shared Resources (Ribosome) 

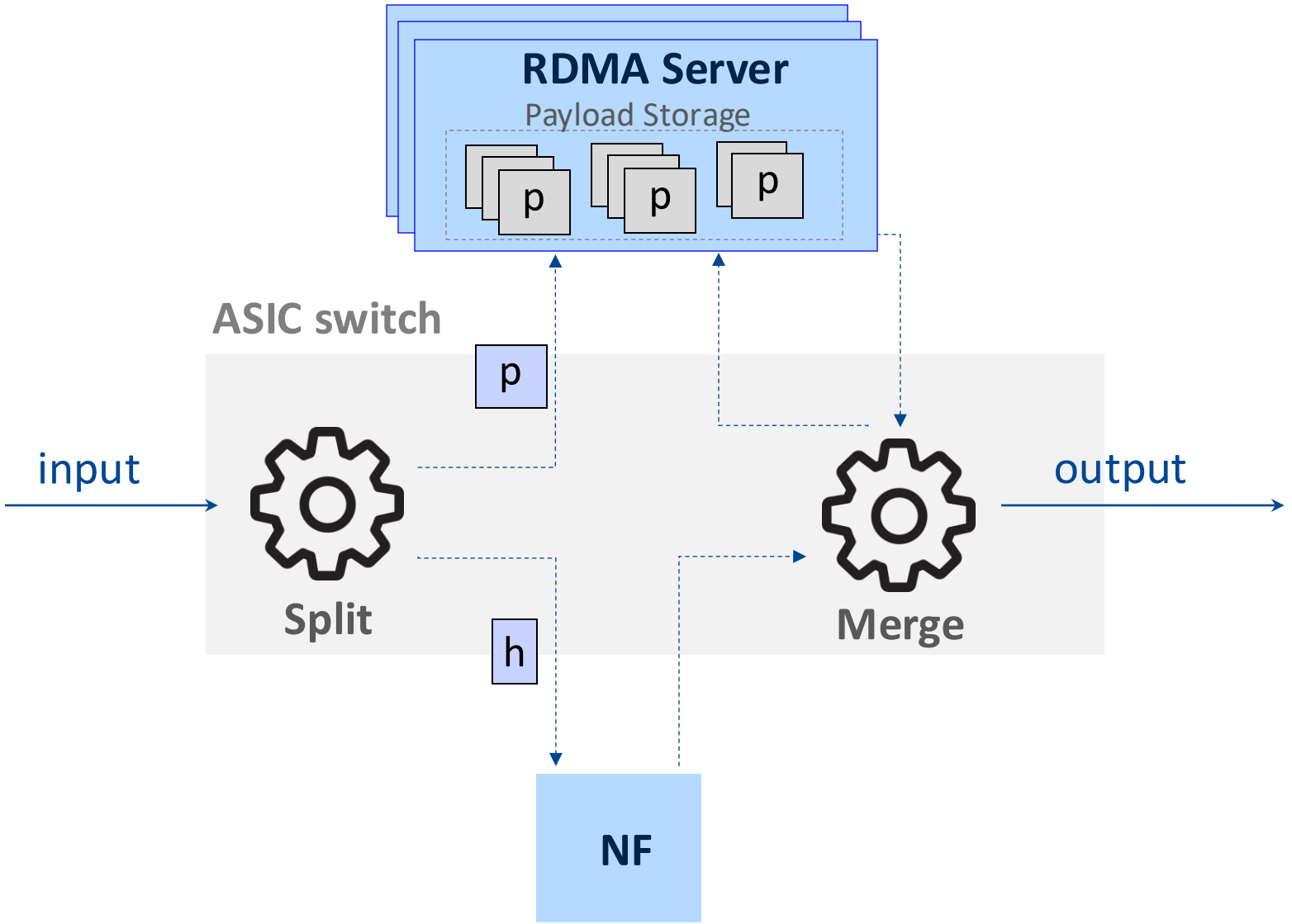
Storing Payloads on Shared Resources (Ribosome)



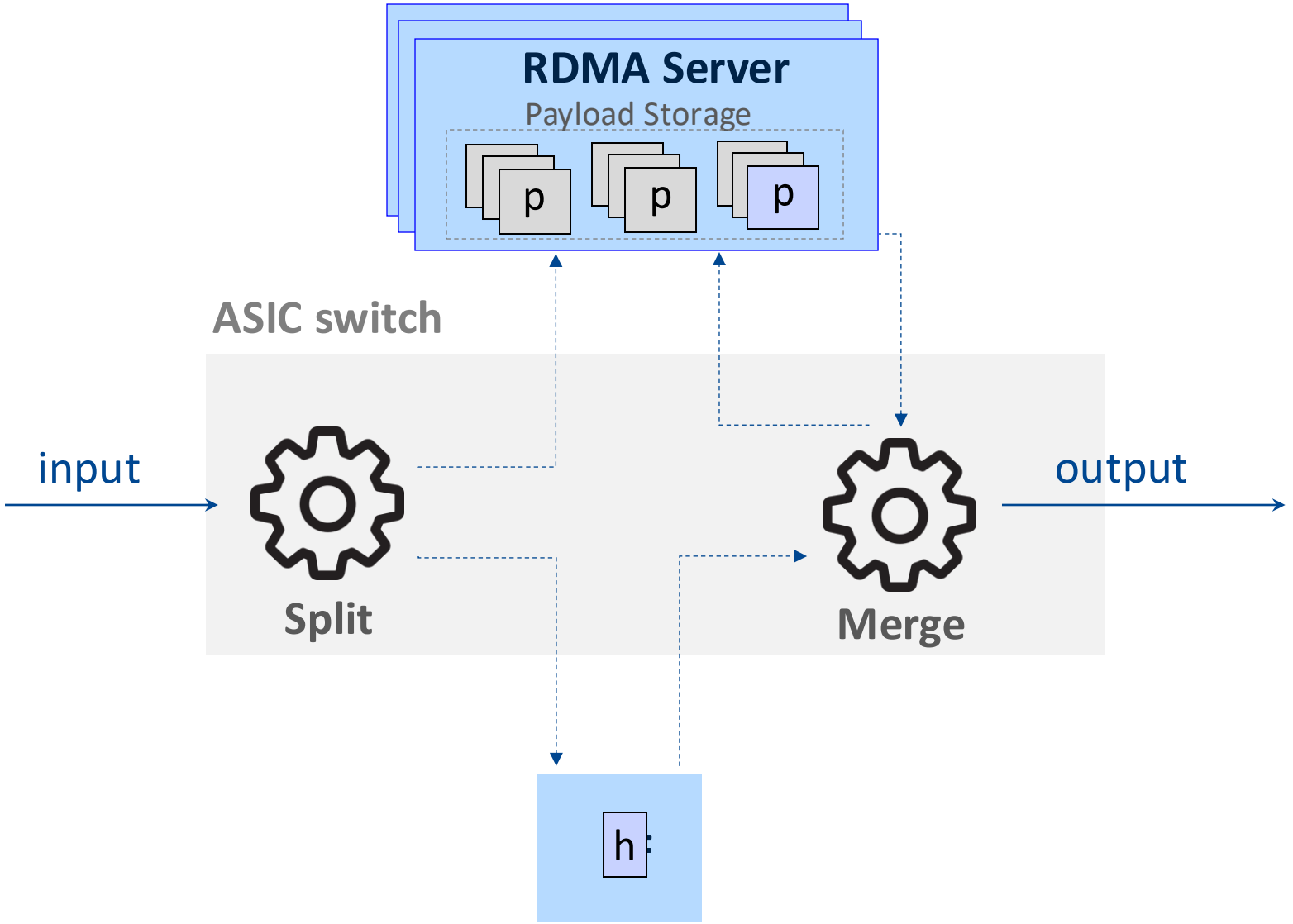
Storing Payloads on Shared Resources (Ribosome)



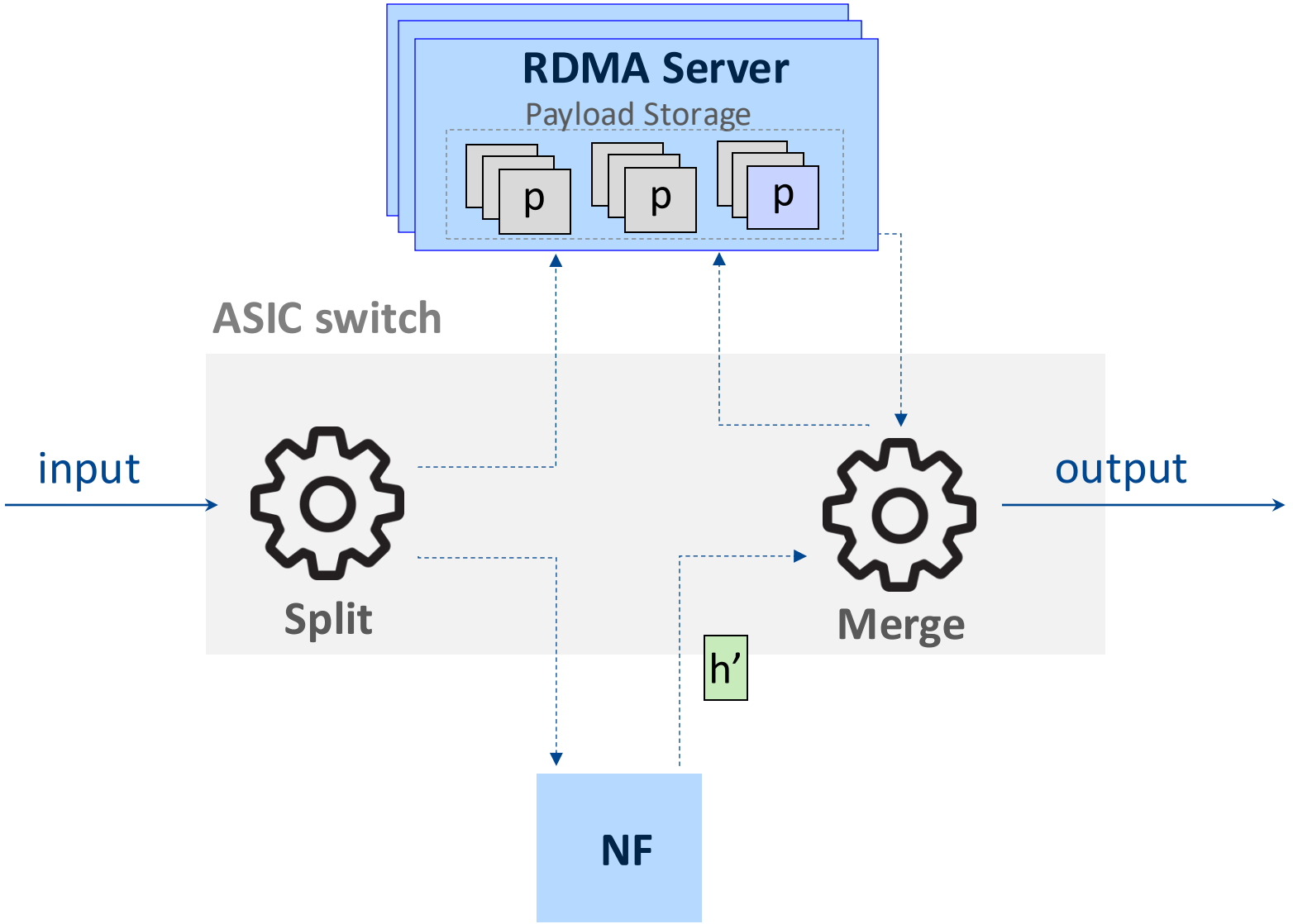
Storing Payloads on Shared Resources (Ribosome)



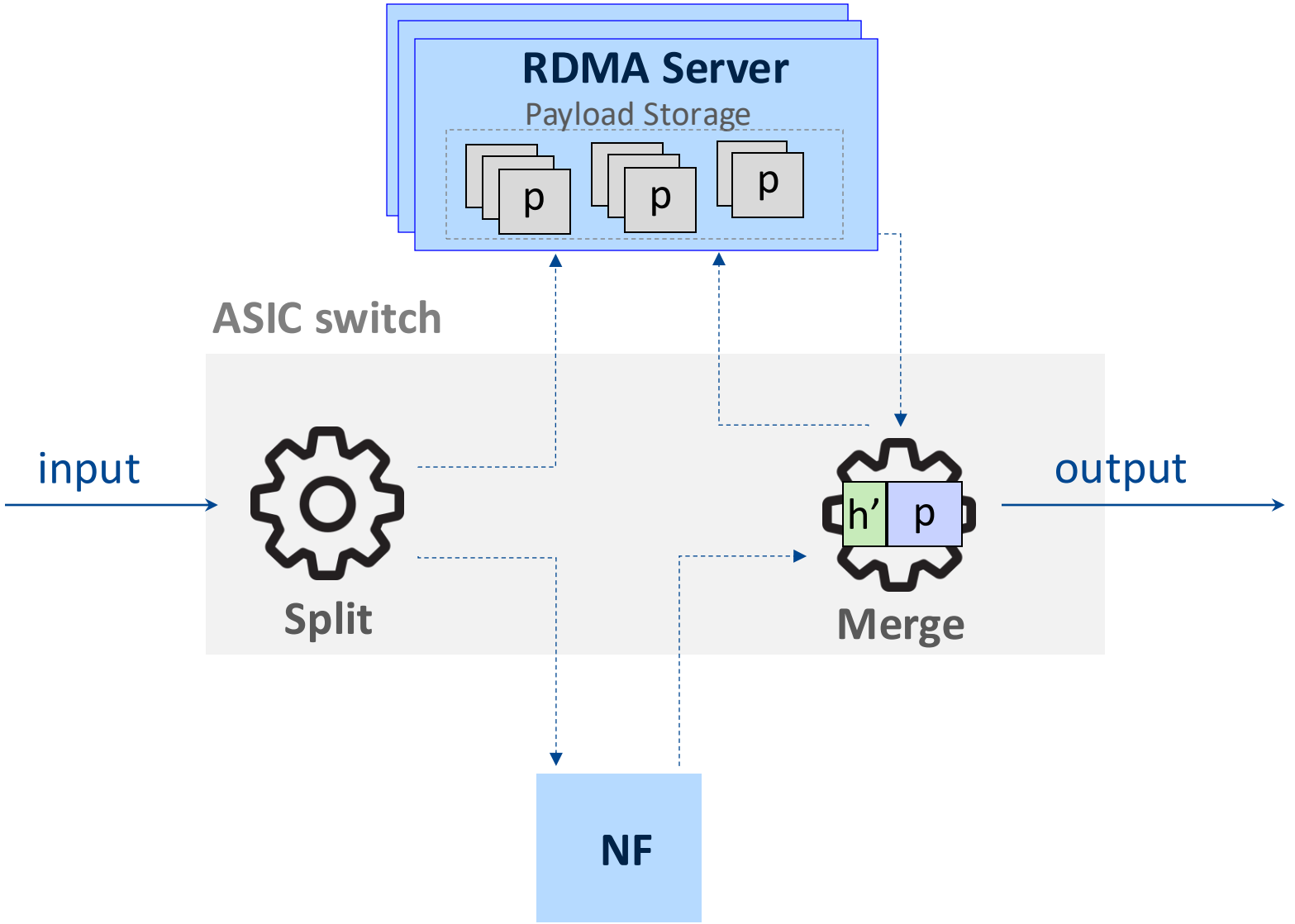
Storing Payloads on Shared Resources (Ribosome)



Storing Payloads on Shared Resources (Ribosome)

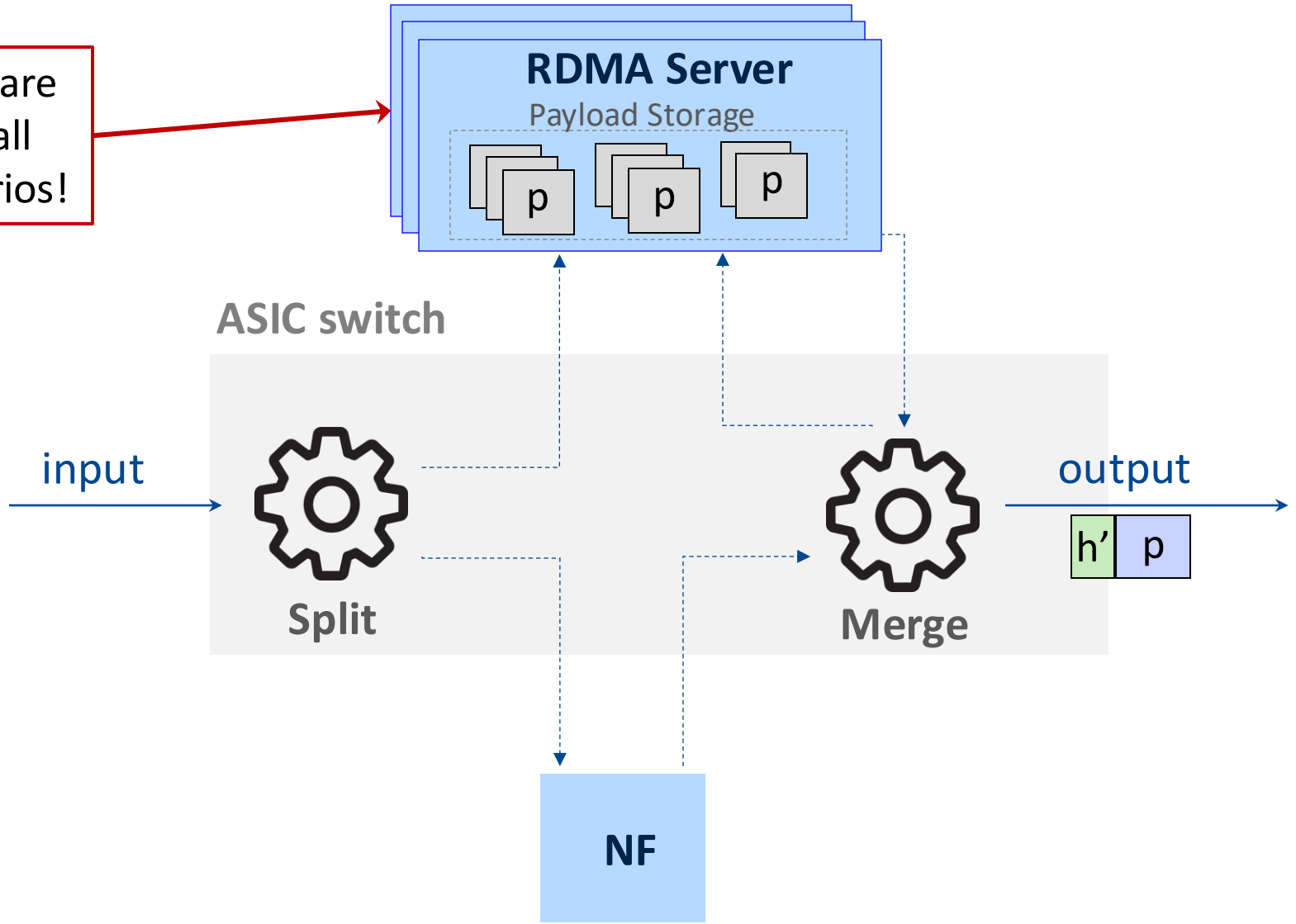


Storing Payloads on Shared Resources (Ribosome)



Storing Payloads on Shared Resources (Ribosome)

Shared resources are not available in all operational scenarios!



Packet-Splitting Approaches

State-of-the-art packet-splitting solutions increase NFs throughput

Storing Payloads in the On-NIC Memory (nicmem) 

Storing Payloads on Shared Resources (Ribosome) 

Store on the ASIC SRAM

- ideal, but how? no API to store payloads!

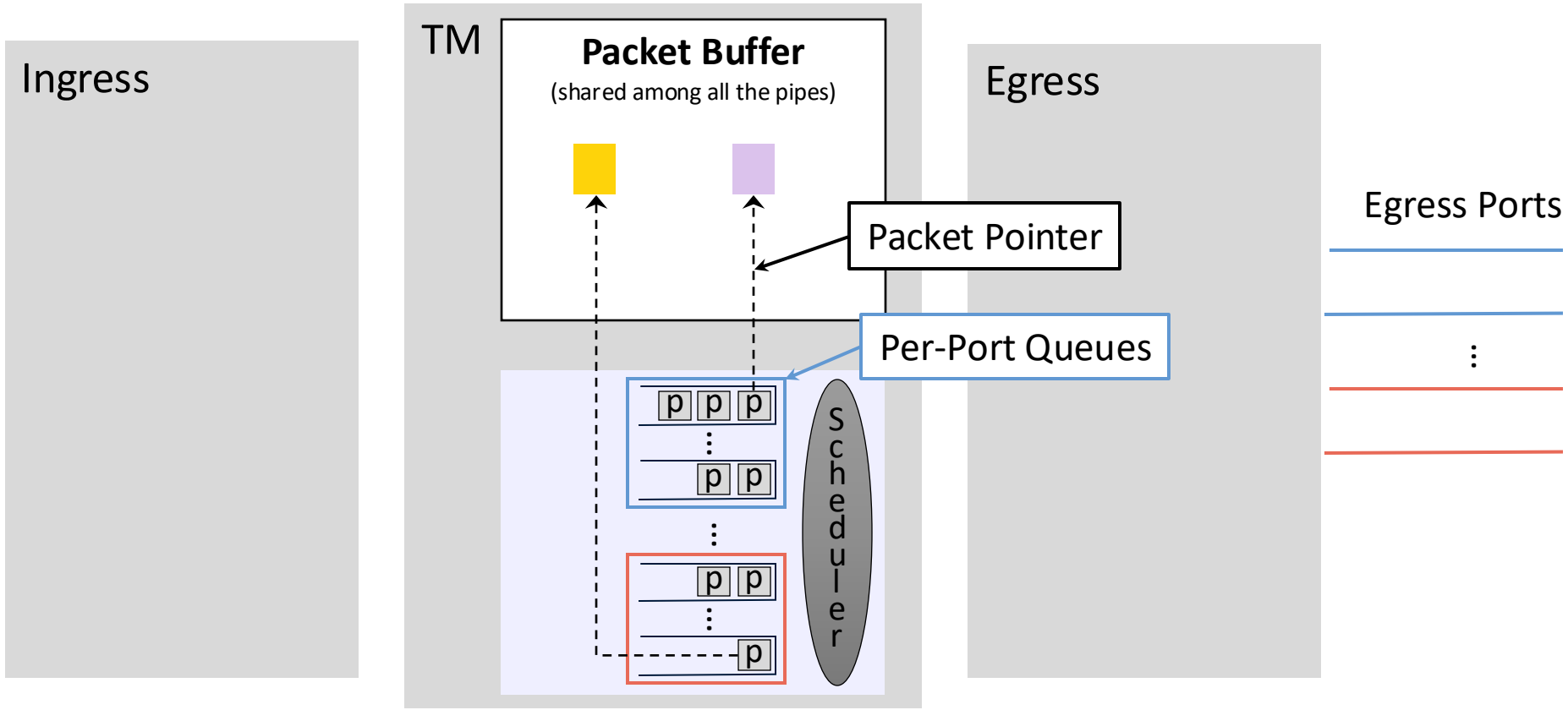
Can we store the entire payload within the switch without requiring external devices?

Core Idea: A Queue-Based Packet Storage

Exploit the switch shared buffer to store payloads

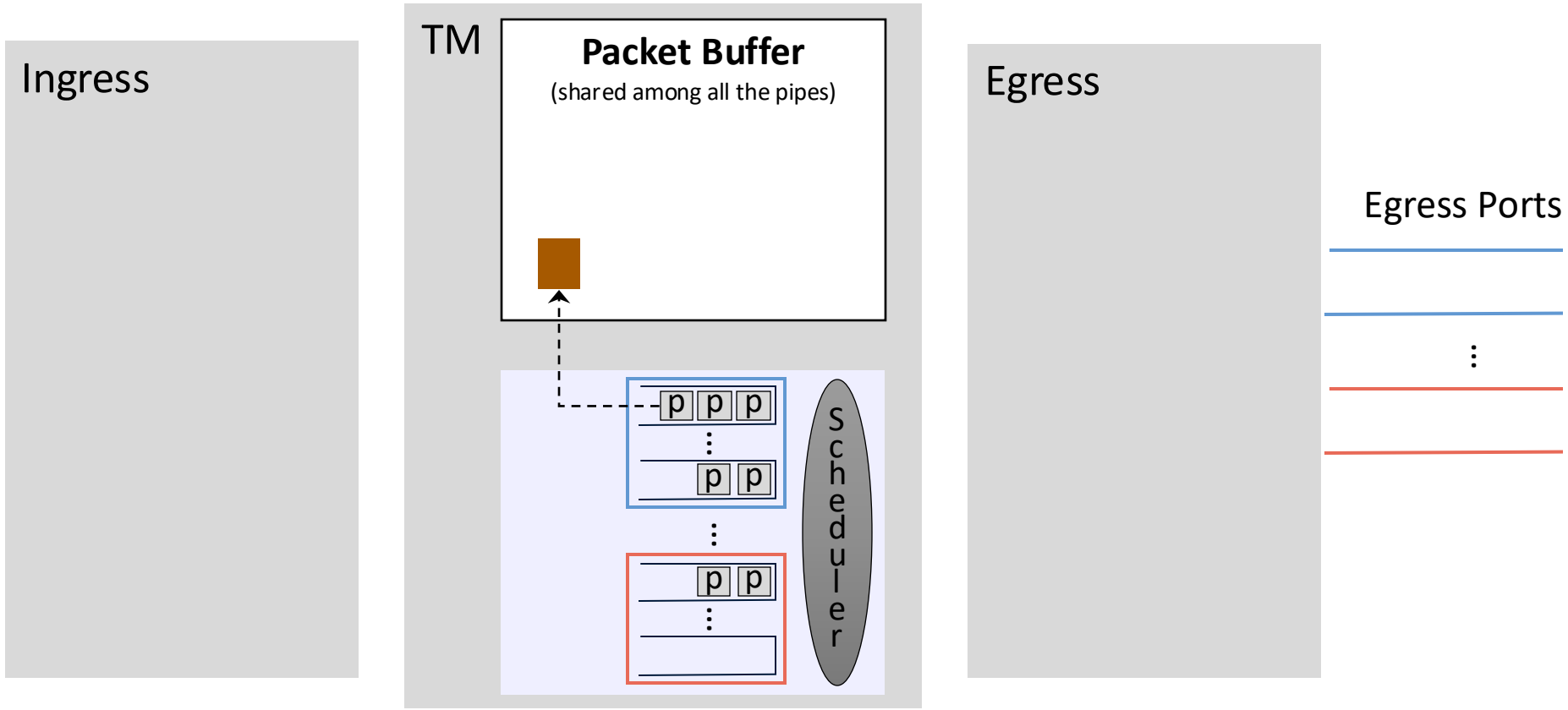
Core Idea: A Queue-Based Packet Storage

Exploit the switch shared buffer to store payloads



Core Idea: A Queue-Based Packet Storage

Exploit the switch shared buffer to store payloads



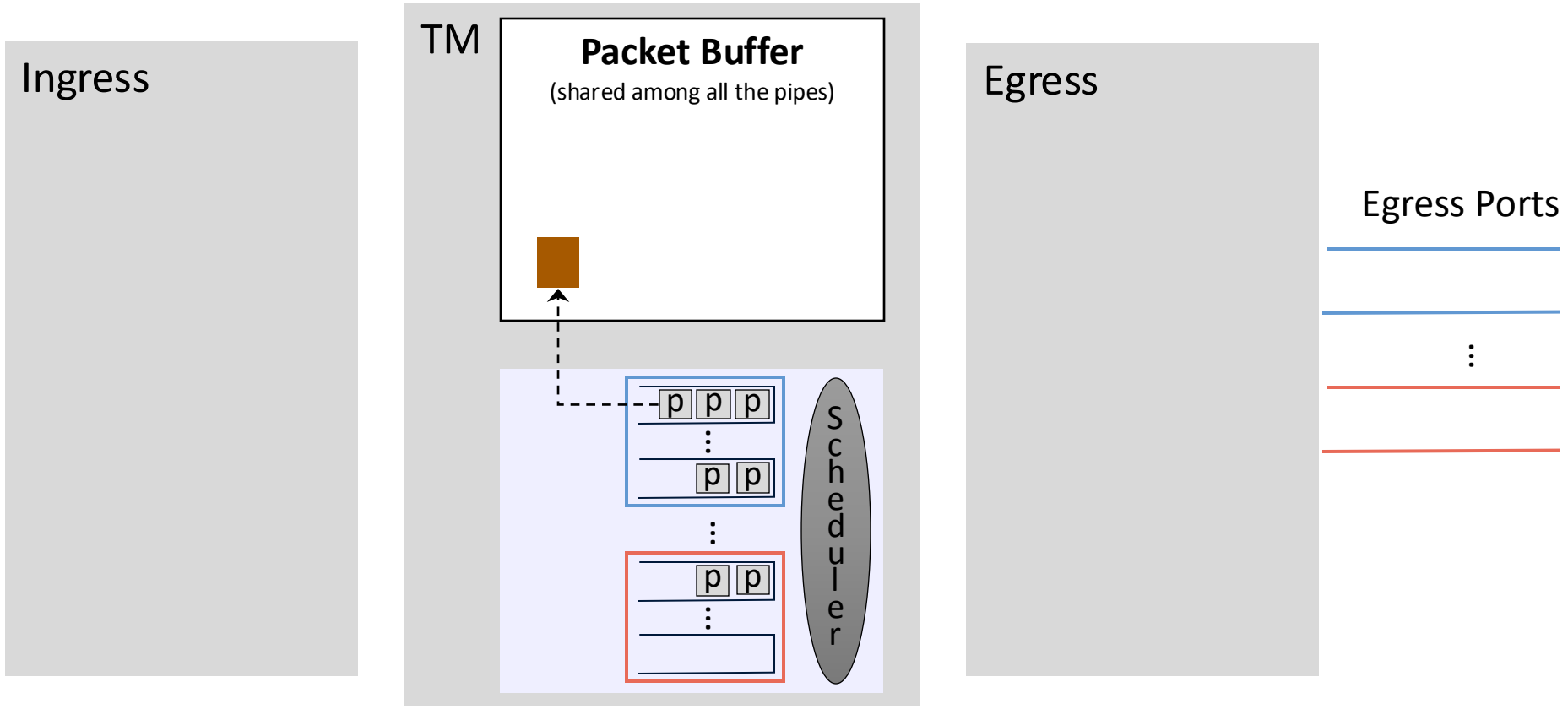
Core Idea: A Queue-Based Packet Storage

Exploit the switch shared buffer to store payloads

✓ Large shared buffer

✗ No programmatic access

? How can we access the packet buffer?



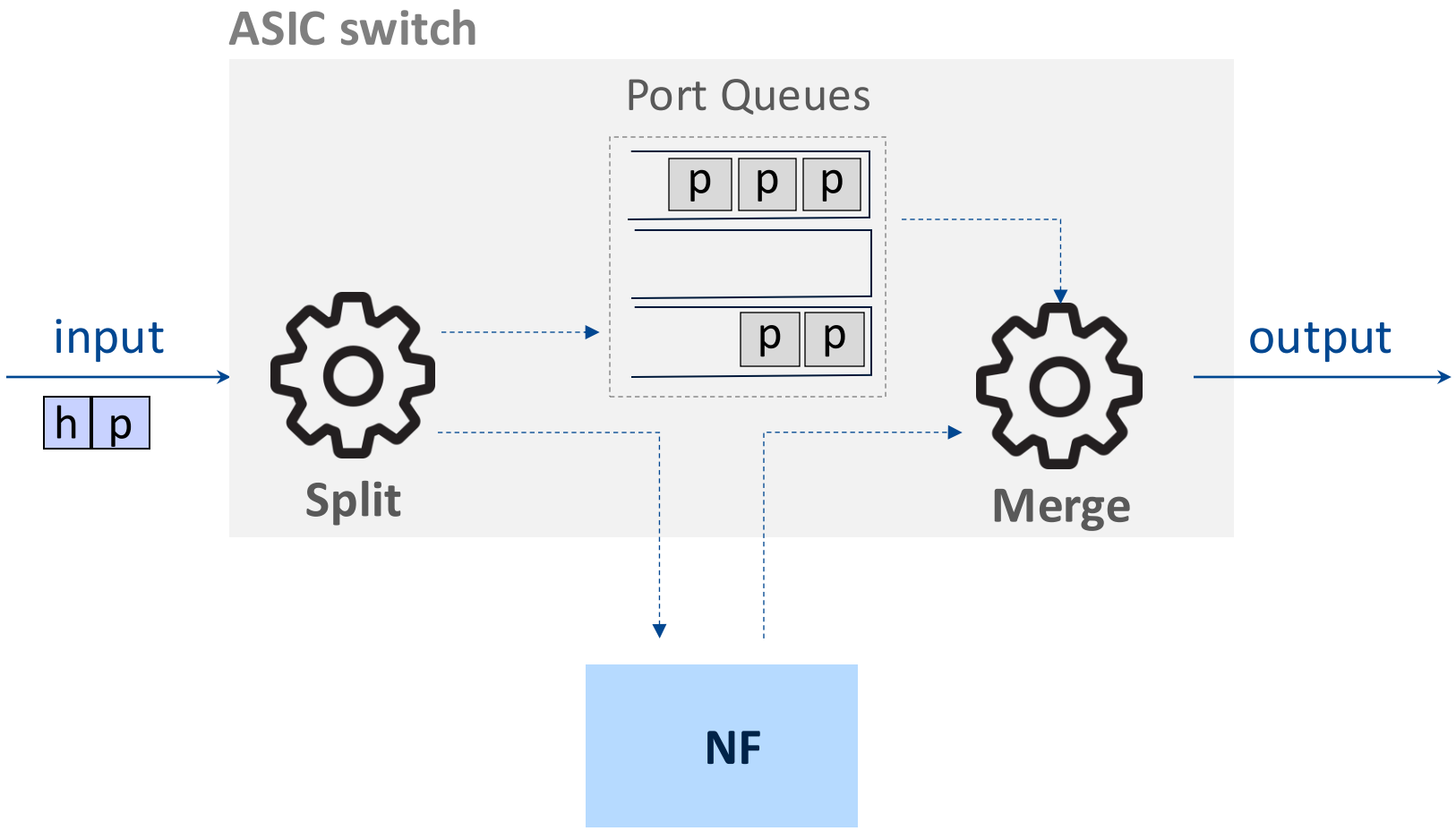
How can we access the packet buffer?

Enqueued packets **implicitly** control the buffer memory

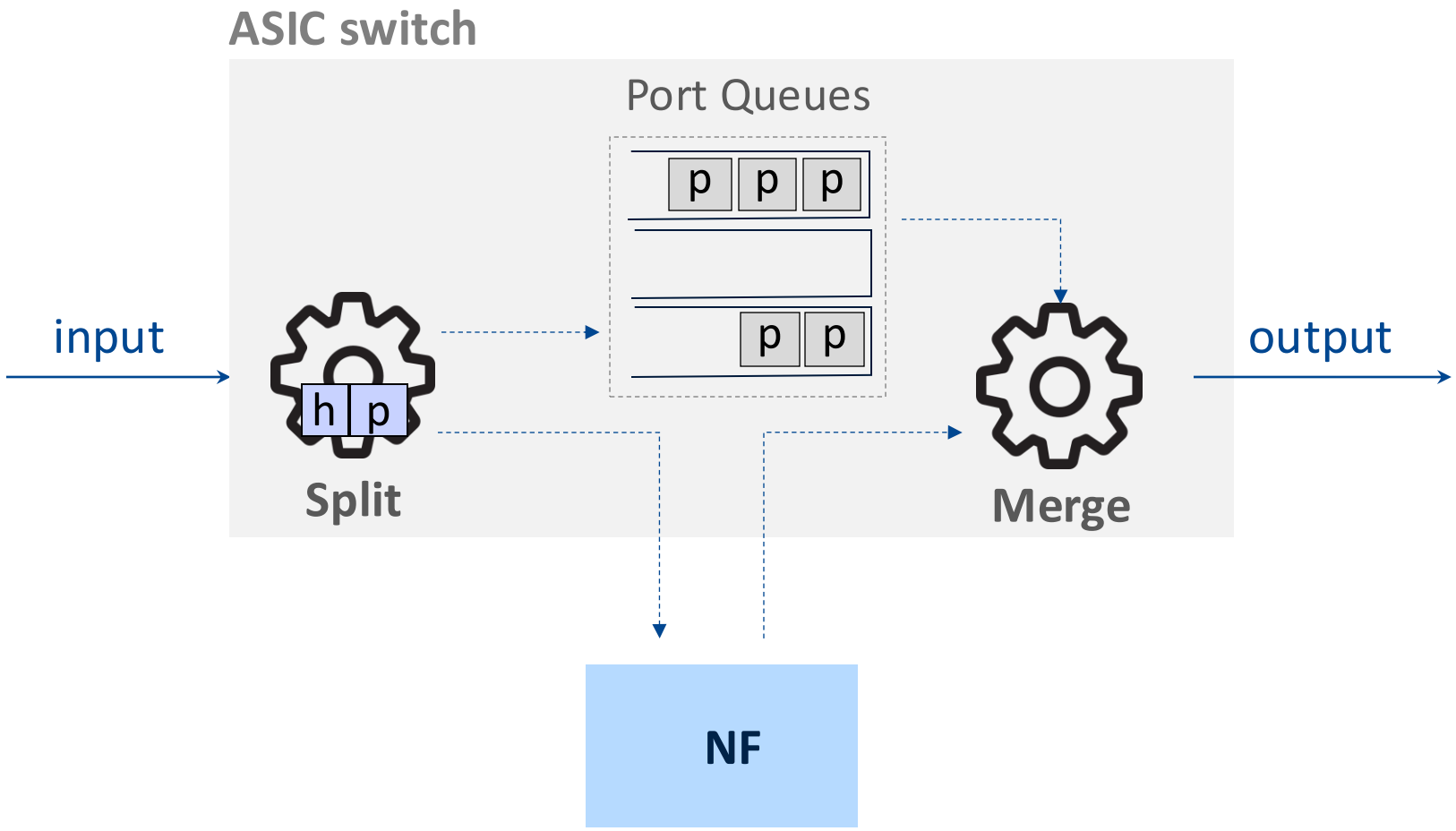
Hold payloads in the port queues until
the processed header from the NF is returned!

Would it work for real?

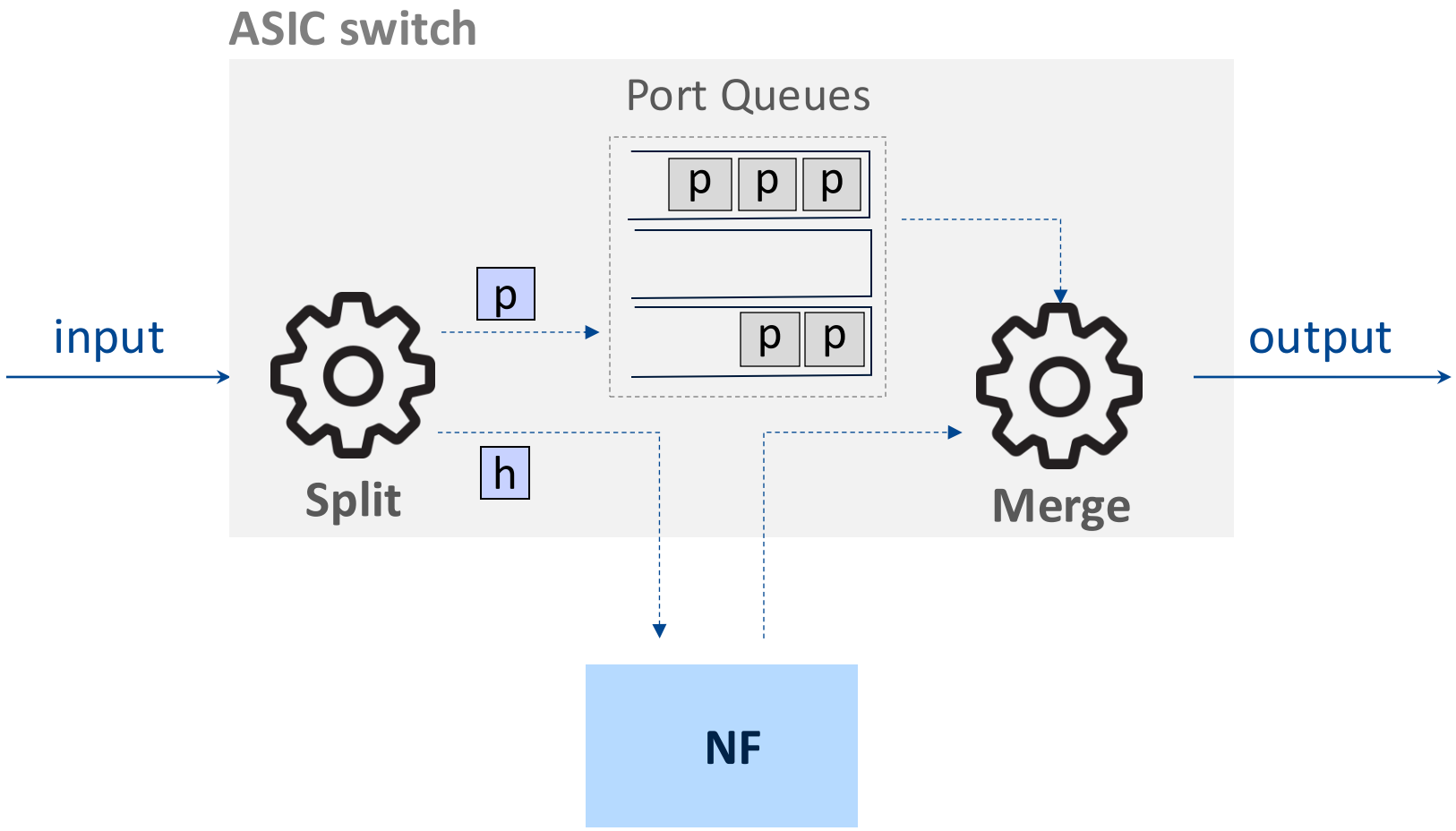
Challenges


ASIC switch
Port Queues
p
p
p
p
p
input
h
p
Split
Merge
output
NF

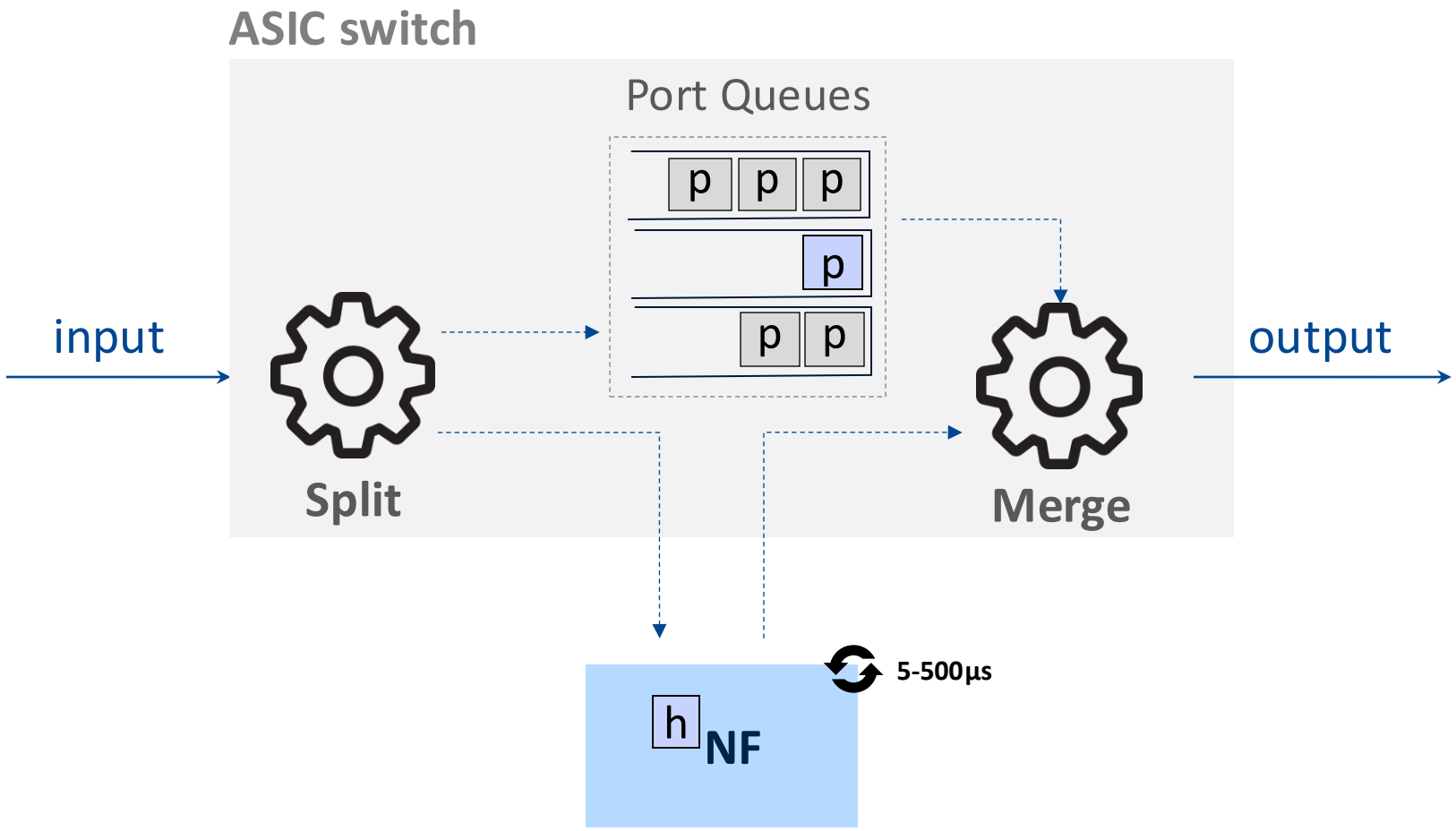
Challenges


ASIC switch
Port Queues
p
p
p
p
p
input
h
p
Split
Merge
output
NF

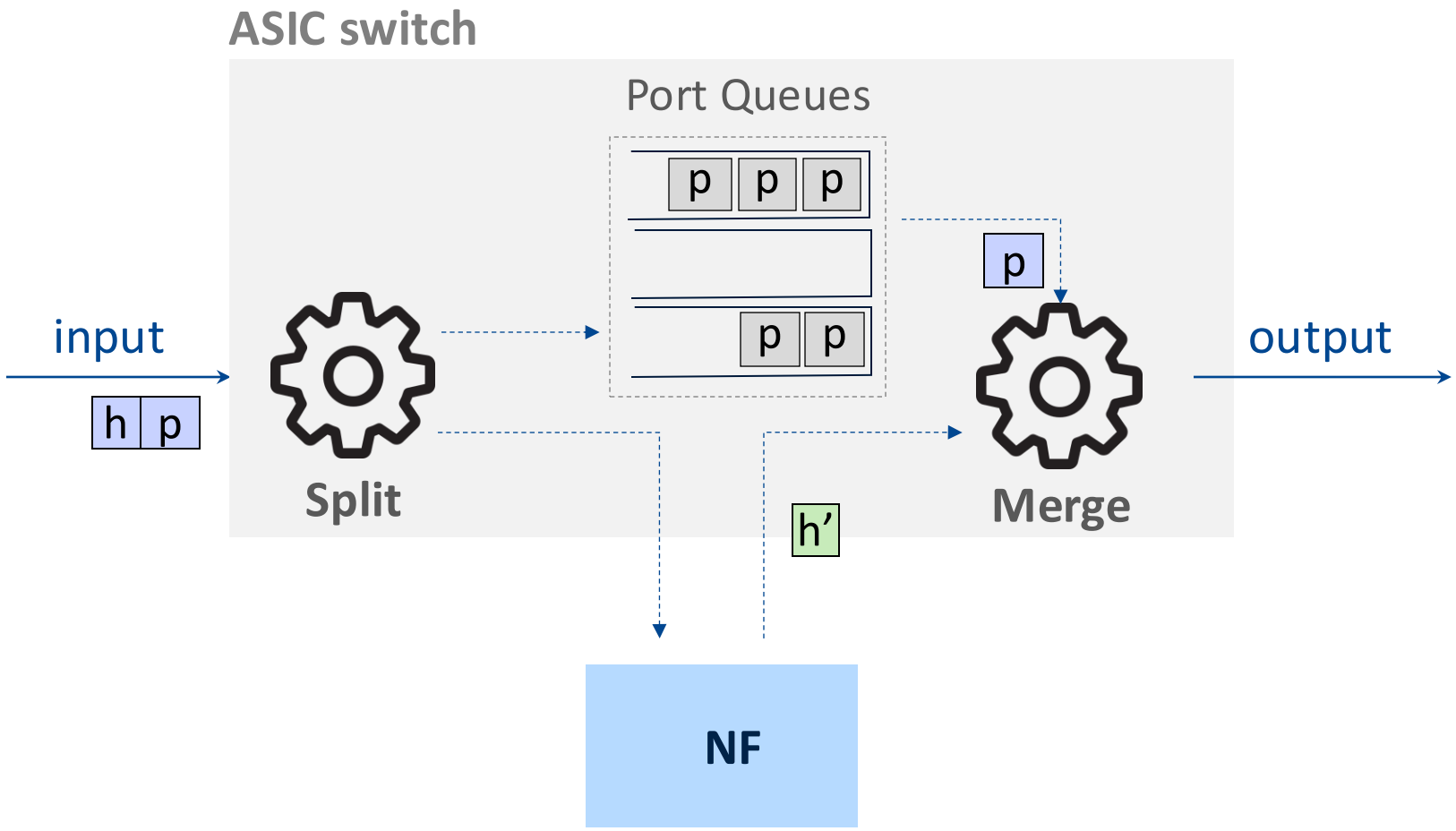
Challenges


ASIC switch
Port Queues
p
p
p
p
p
input
p
output
h
Split
Merge
NF

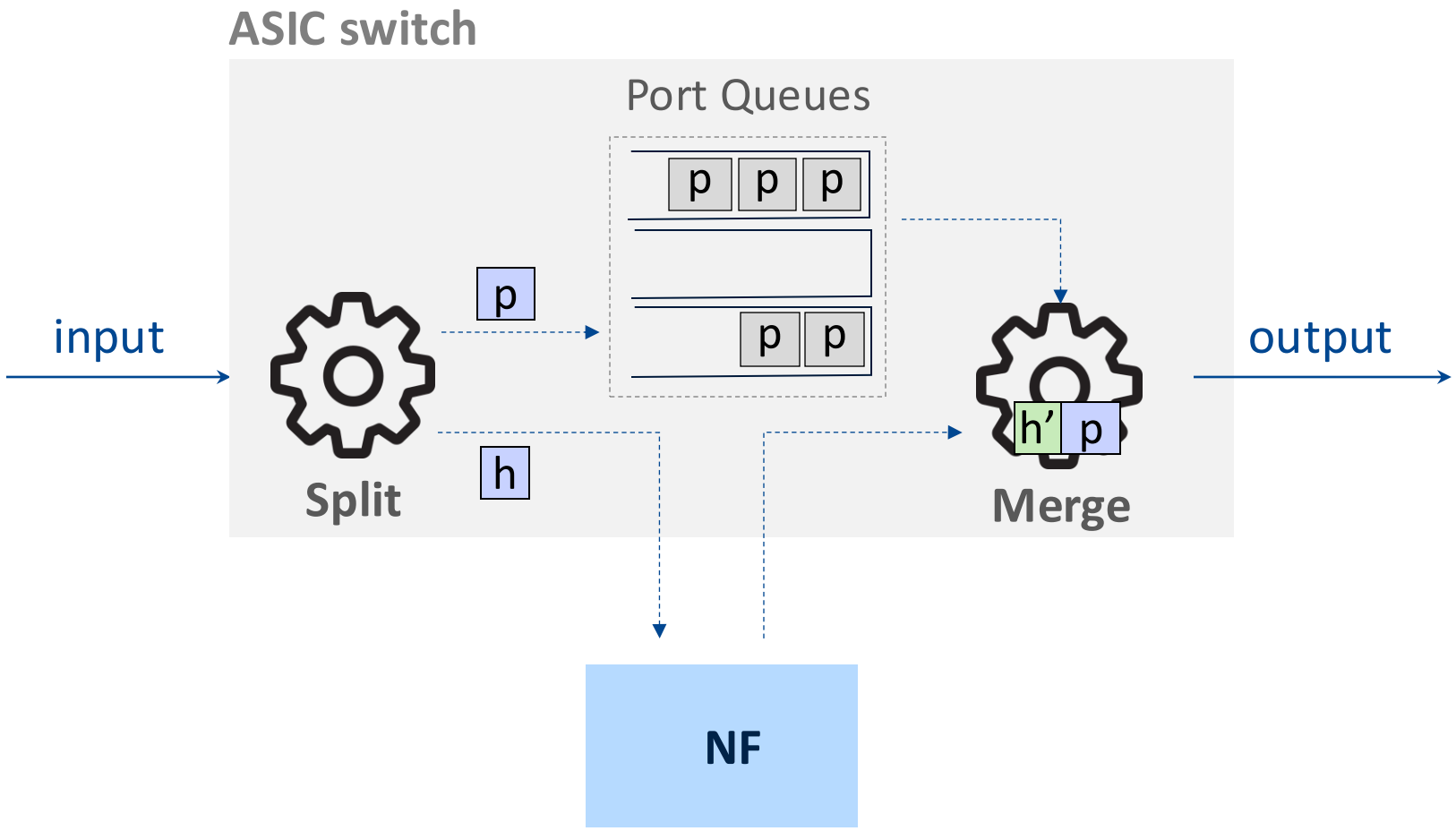
Challenges


ASIC switch
Port Queues
p
p
p
p
p
p
input
output
Split
Merge
h
NF
5-500µs

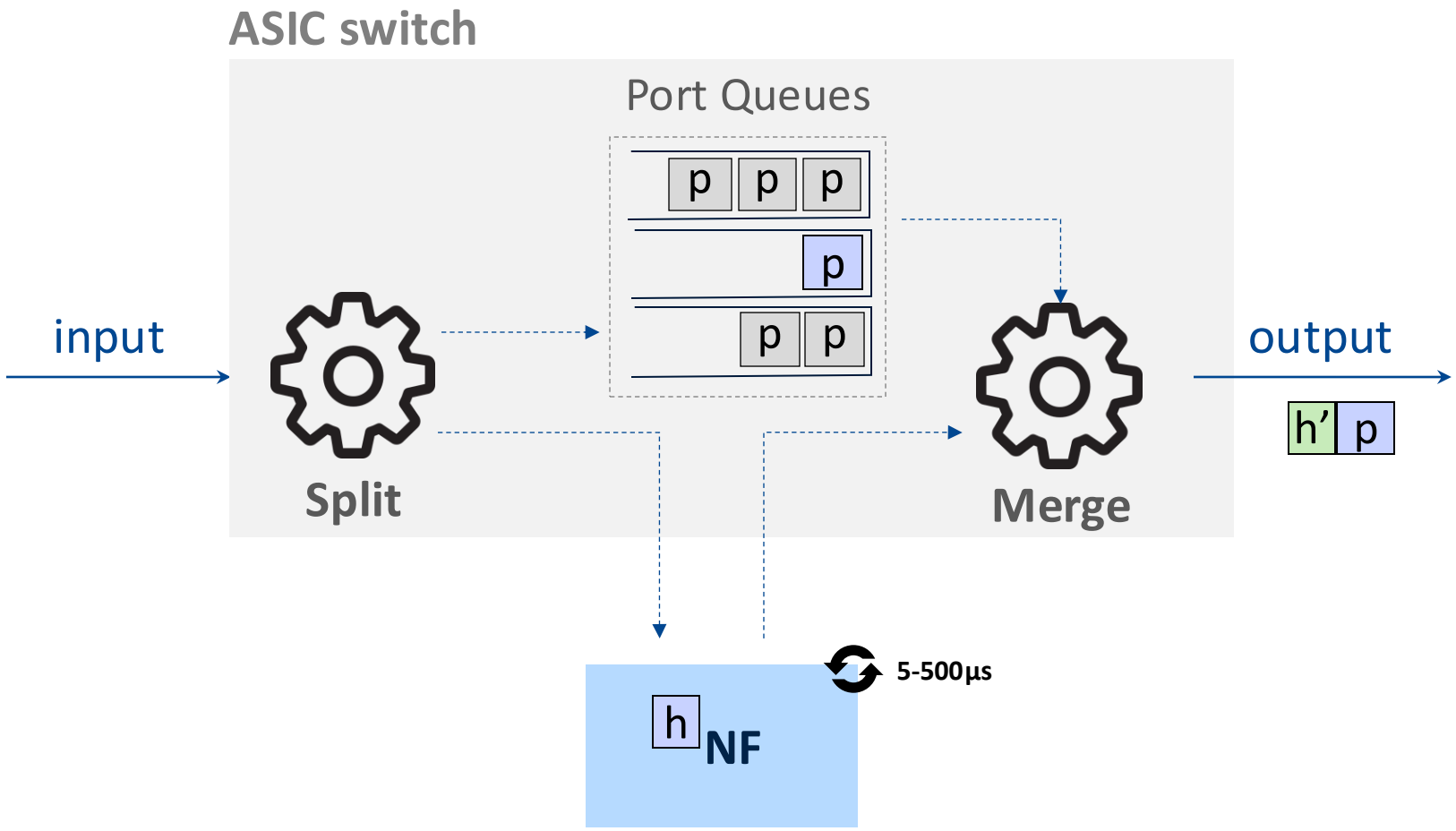
Challenges


ASIC switch
Port Queues
p
p
p
p
p
p
input
h
p
Split
h'
Merge
output
NF

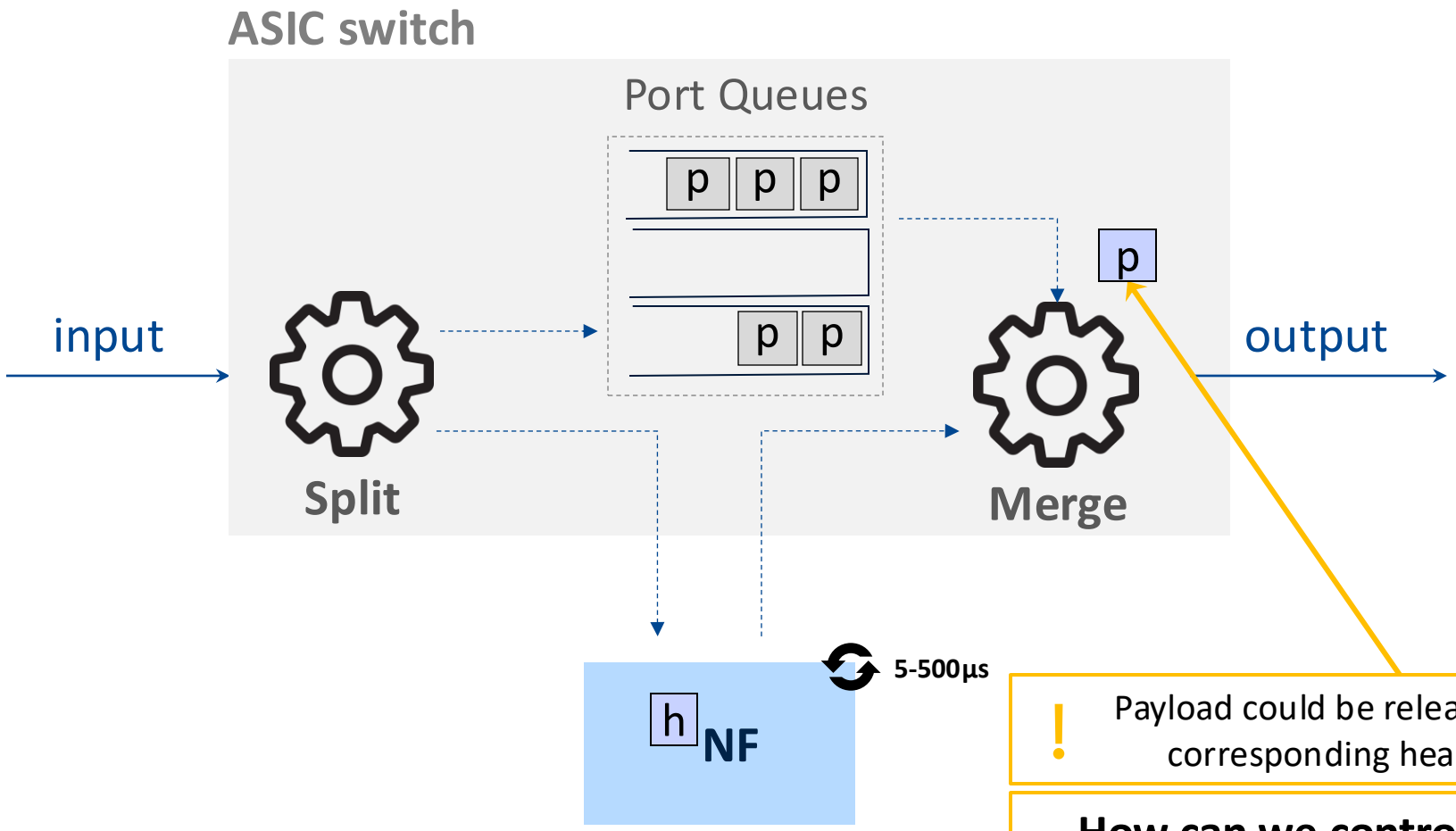
Challenges


ASIC switch
Port Queues
p
p
p
p
p
input
p
output
h'
p
h
Split
Merge
NF

Challenges


ASIC switch
Port Queues
p
p
p
p
p
p
input
output
h'
p
Split
Merge
5-500µs
h
NF

Challenges


ASIC switch
Port Queues
p
p
p
p
p
p
input
output
Split
Merge
h
NF
5-500µs
!
Payload could be released before the corresponding header is back!
How can we control the queues?

How can we control the queues?

Congest the egress queues to delay payloads release by the time required for NF processing

How can we control the queues?

Congest the egress queues to delay payloads release by the time required for NF processing

Recirculate payloads until headers are ready

How can we control the queues?

Congest the egress queues to delay payloads release by the time required for NF processing

Recirculate payloads until headers are ready

Create copies of payloads to fill up the queues

How can we control the queues?

Congest the egress queues to delay payloads release by the time required for NF processing

Recirculate payloads until headers are ready

Create copies of payloads to fill up the queues

Stop the queue with Priority-Flow-Control (PFC)

How can we control the queues?

Congest the egress queues to delay payloads release by the time required for NF processing

Recirculate payloads until headers are ready

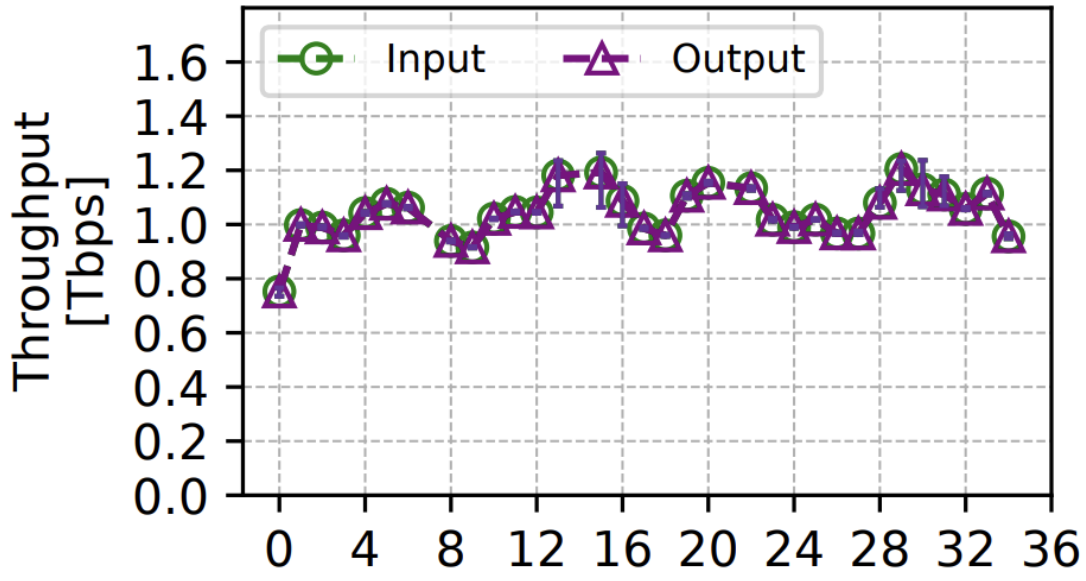
Create copies of payloads to fill up the queues

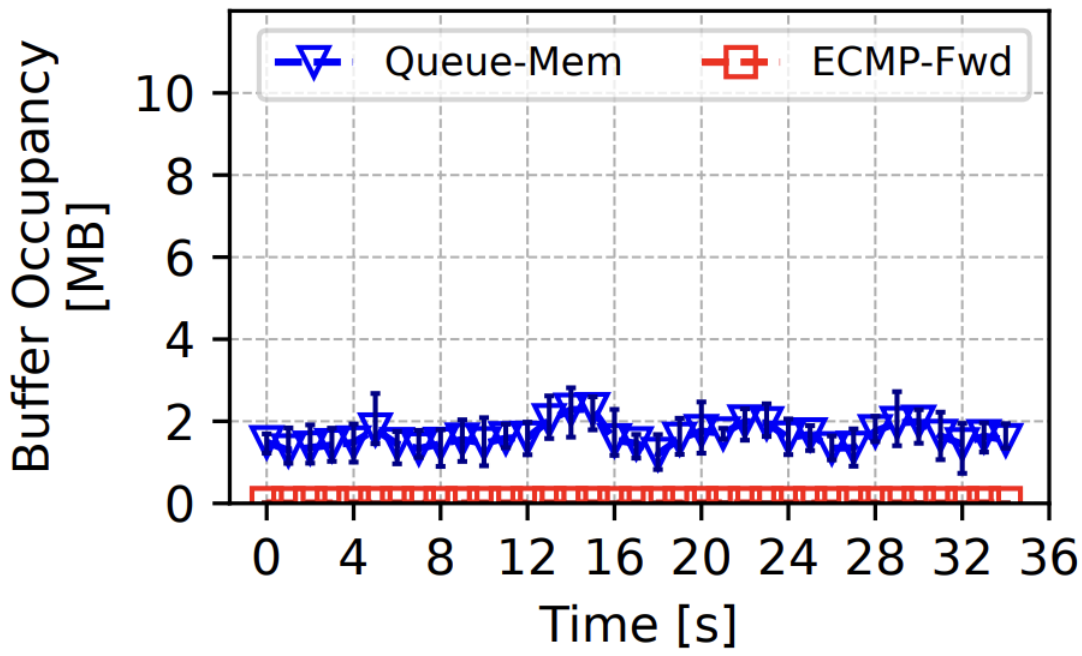
QueueMem

Stop the queue with Priority-Flow-Control (PFC)

Evalution of QueueMem (combined with FAJITA)

Network function: Per-flow counter + Load balancer + Per-flow rate limiter





FAJITA + QueueMem conclusions

The **FAJITA** system:

- **incorporates** the three essential principles for stateful packet processing **together**
- **minimizes** memory access overheads by **exploiting batching** & **software prefetching**
- **alleviates** overheads of accessing shared data structures **by introducing auxiliary HTs**
- achieves **performance 2x higher** than existing stateful packet processors
- demonstrates that dynamic inter-core load balancing is **detrimental** at high rates
- only processes packet headers

The **QueueMem** system (under submission):

- supports packet splitting entirely on the switch
- reduces power consumption
- relies on PFC to limit buffer usage

FAJITA + QueueMem conclusions

The **FAJITA** system:

- **incorporates** the three essential principles for stateful packet processing **together**
- **minimizes** memory access overheads by **exploiting batching** & **software prefetching**
- **alleviates** overheads of accessing shared data structures **by introducing auxiliary HTs**
- achieves **performance 2x higher** than existing stateful packet processors
- demonstrates that dynamic inter-core load balancing is **detrimental** at high rates
- only processes packet headers

The **QueueMem** (under submission):

- supports packet splitting entirely on the switch
- reduces power consumption
- relies on PFC to limit buffer usage