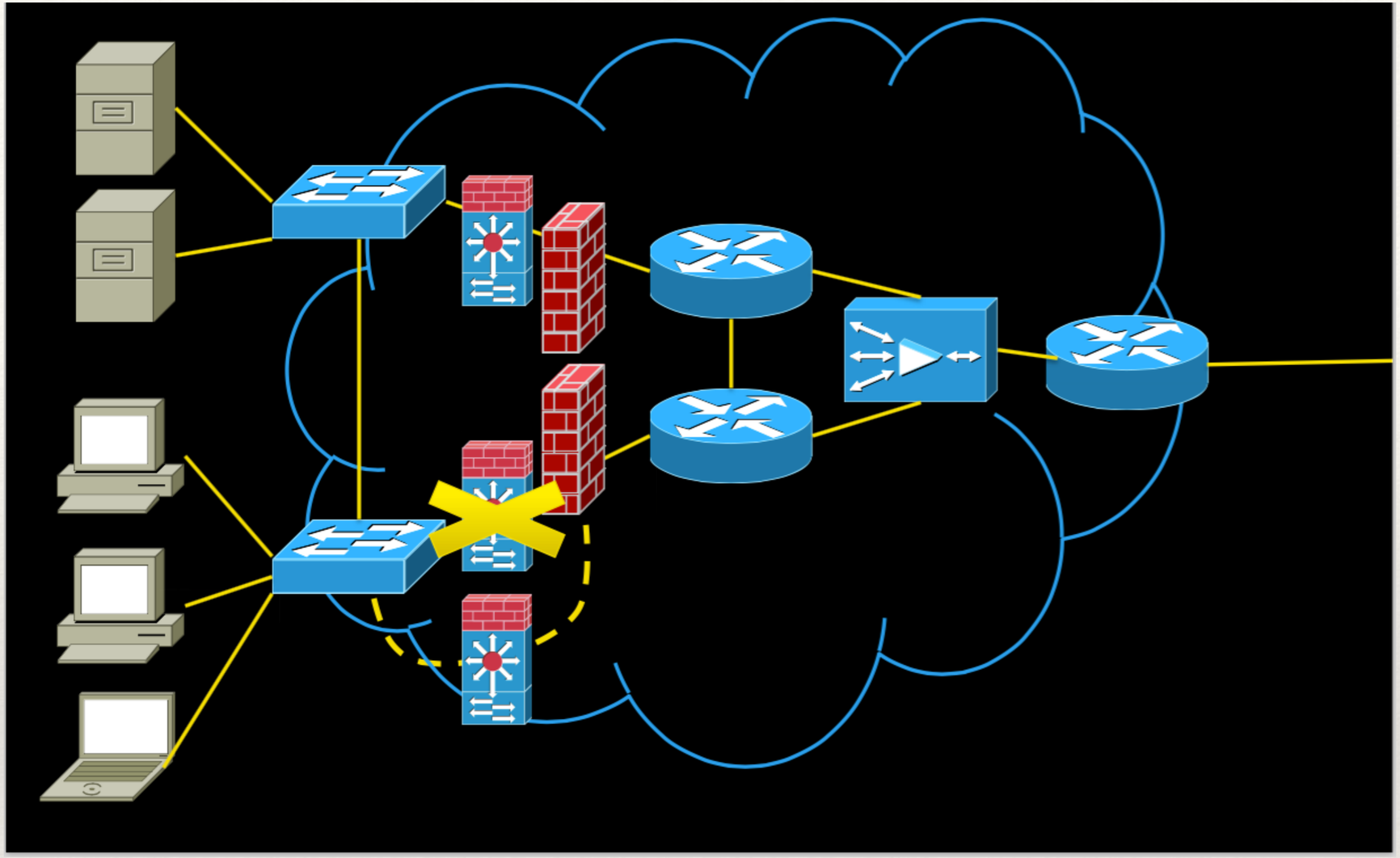
Rollback-Recovery for Middleboxes

Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda,
Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh,
João Martins, Sylvia Ratnasamy, Luigi Rizzo, Scott Shenker



Middlebox Recovery: fail over to a back-up device after a middlebox goes offline, without interrupting connectivity or causing errors.

Key Challenge:
Correctness vs Performance

Systems Today: Correctness xor Performance

- Cold restart:
  - Fast, no overhead
  - Leads to buggy behaviour for stateful MBs, like missed attack detection
- Using snapshot/checkpoint:
  - Correctness guaranteed, no modification to MB
  - But adds latencies of 8-50ms; increases page loads by 200ms-1s
- Active-active implementation:
  - Cannot guarantee correctness either because of non-determinism

1980's FT Research: "Output Commit"

Before releasing a packet: has all inform[illegible]n reflecting that packet been committed to storage?

Output Commit triggered frequently.

Necessary condition for correctness.

Typically implemented with check every time data is released.

Middleboxes produce output every microsecond; release operates in parallel.

FTMB: “Fault-Tolerant Middlebox”

Correct Recovery and Performance

Obeys output commit using **ordered logging and parallel release.**

30us latency overhead

5-30% throughput reduction

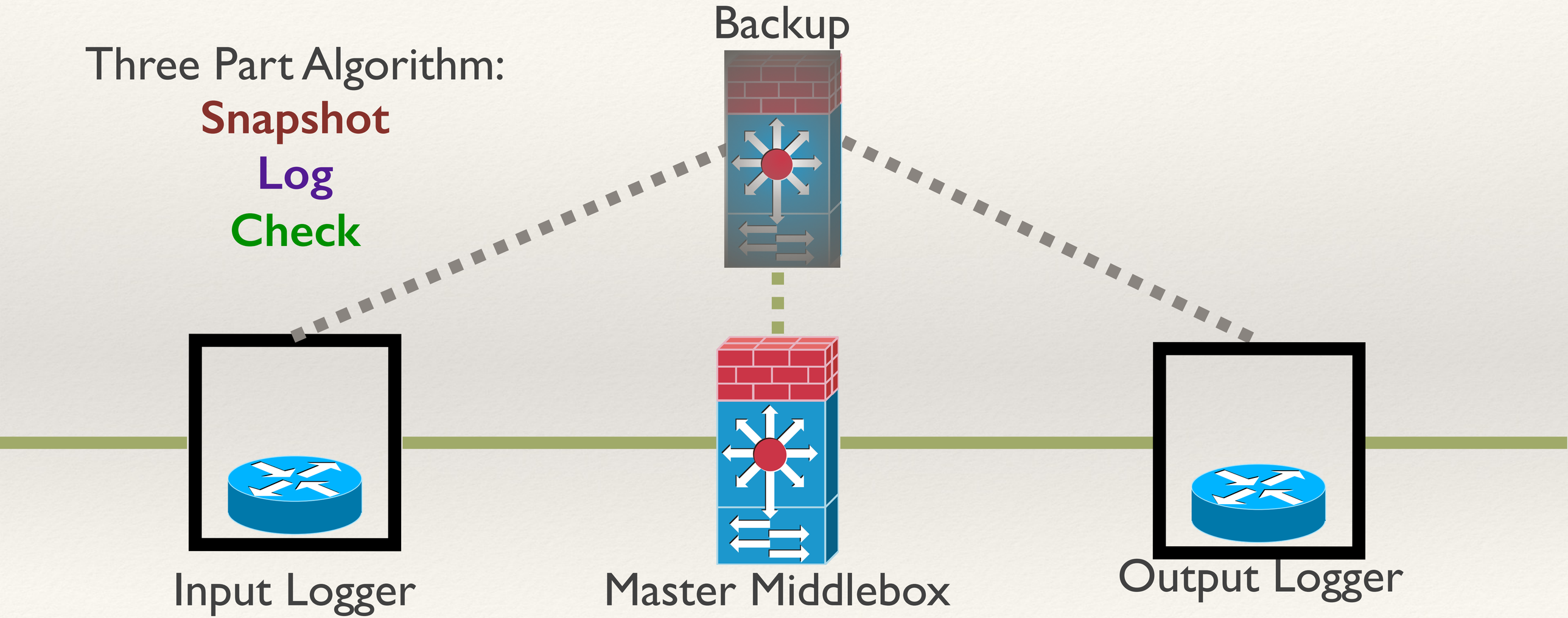
FTMB implements Rollback Recovery.

Three Part Algorithm:

Snapshot

Log

Check



Rollback Recovery

Three Part Algorithm:

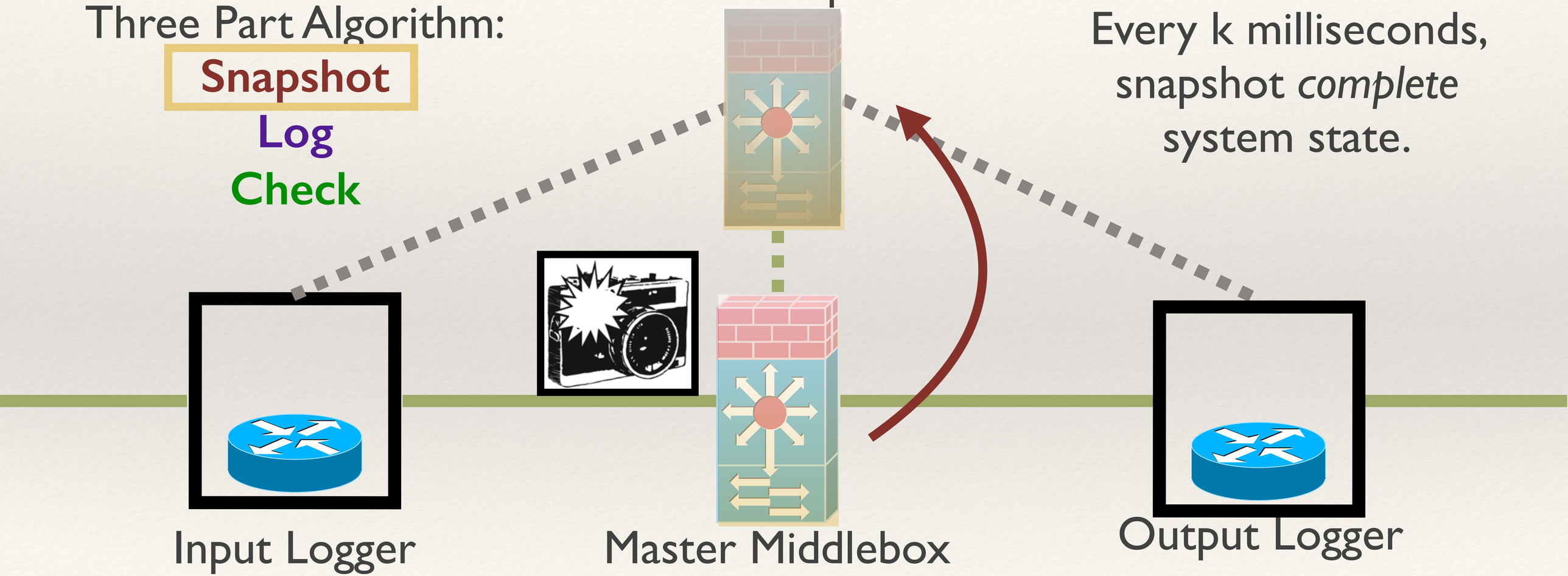
Snapshot

Log

Check

Backup

Every k milliseconds, snapshot *complete* system state.



Input Logger

Master Middlebox

Output Logger

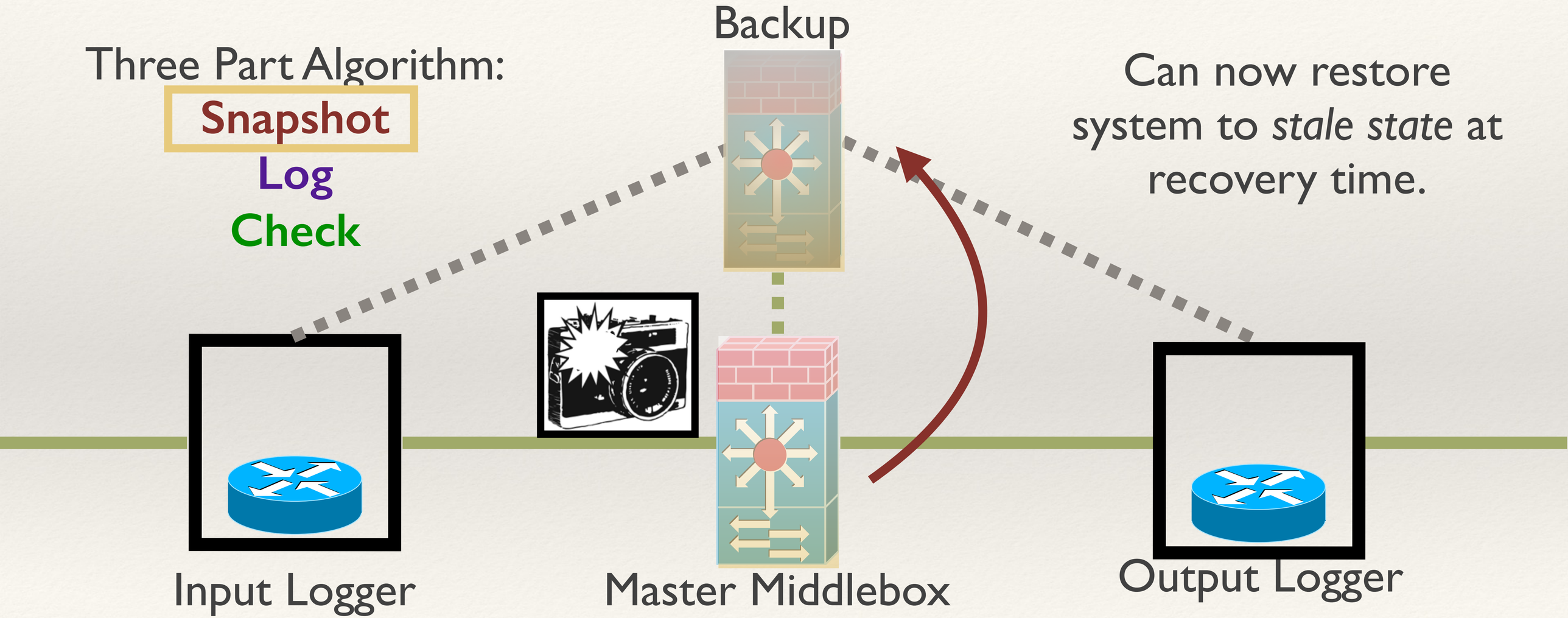
Rollback Recovery

Three Part Algorithm:

Snapshot

Log

Check



Can now restore system to *stale state* at recovery time.

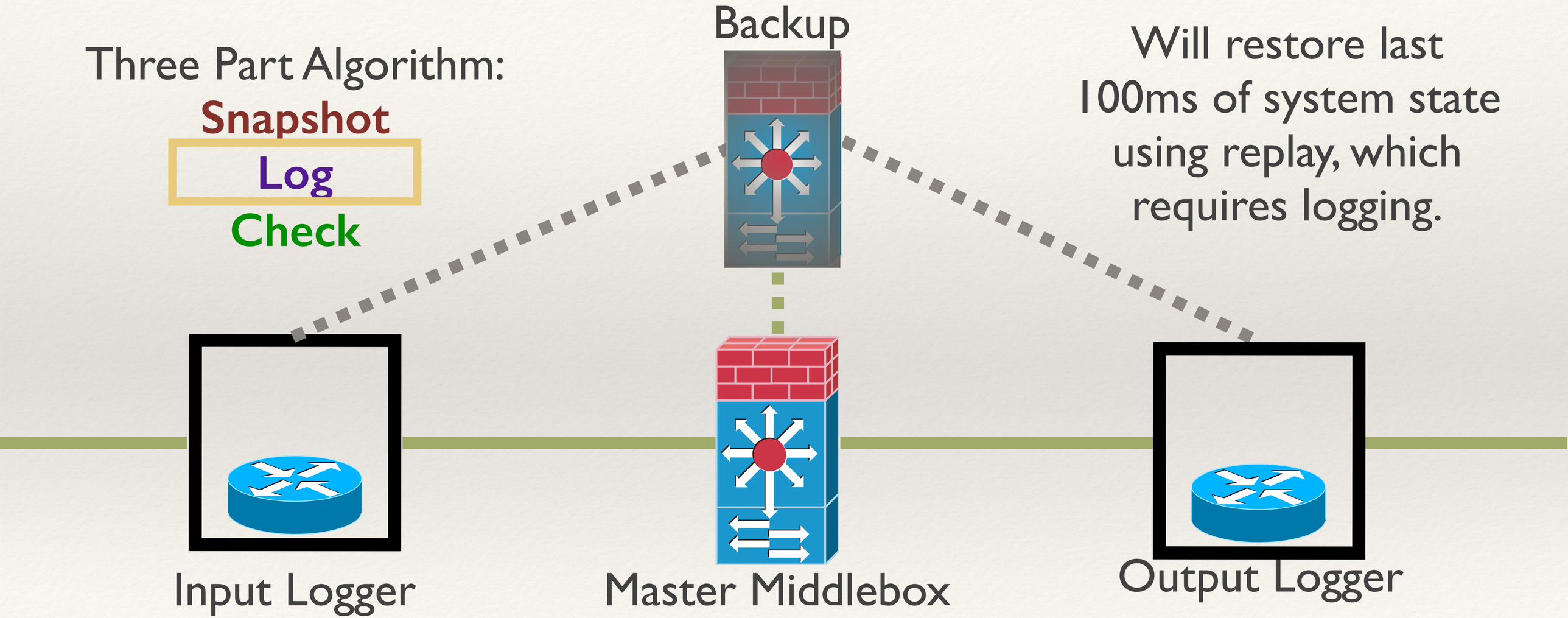
Rollback Recovery

Three Part Algorithm:

Snapshot

Log

Check



Will restore last 100ms of system state using replay, which requires logging.

Rollback Recovery

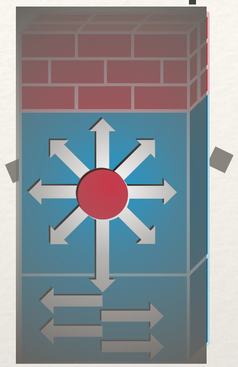
Three Part Algorithm:

Snapshot

Log

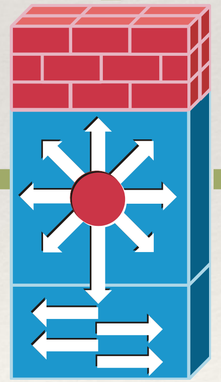
Check

Backup



Check to make sure we have all logged data required for replay at Output Logger.

Input Logger

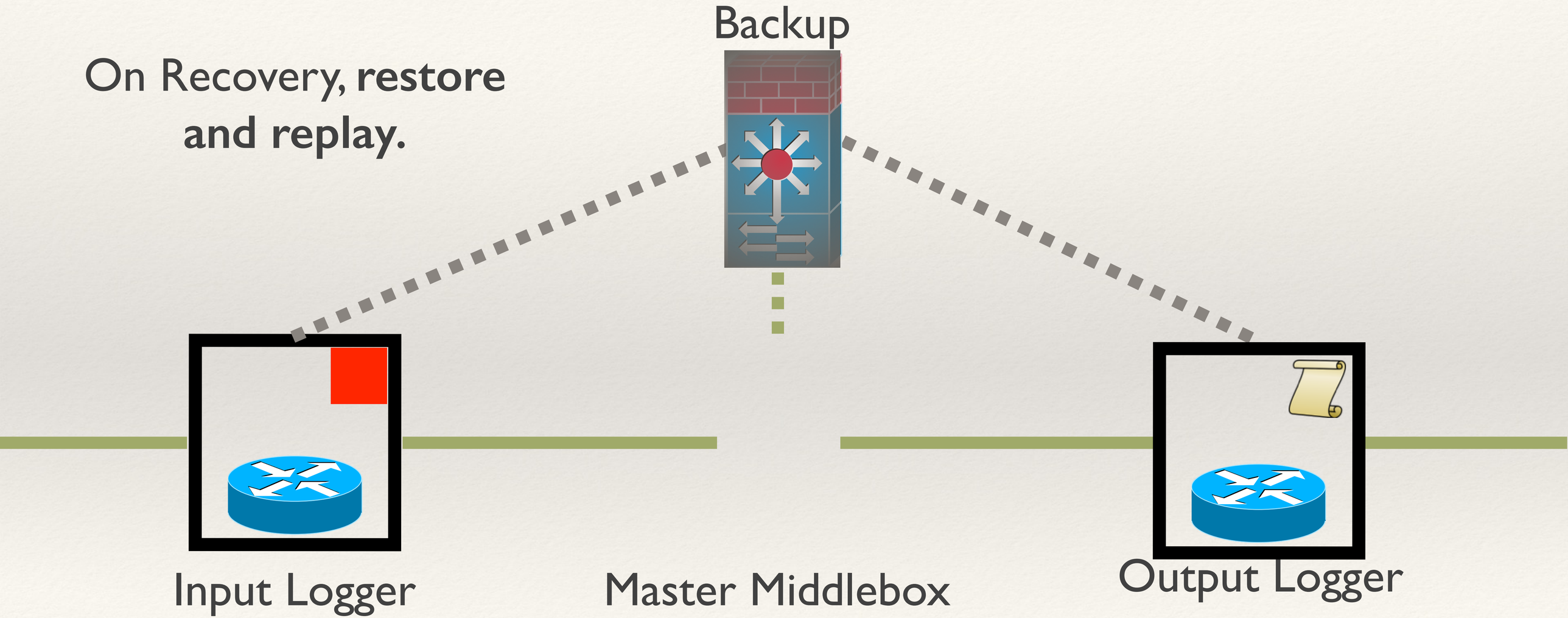


Master Middlebox

Output Logger

Rollback Recovery

On Recovery, **restore and replay.**



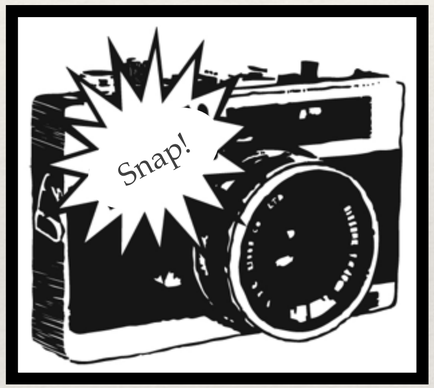
Rollback Recovery

Three Part Algorithm:

Snapshot

Log

Check



Snapshotting algorithms are well-known. We used VM checkpointing.

Rollback Recovery

Three Part Algorithm:

Open Questions:

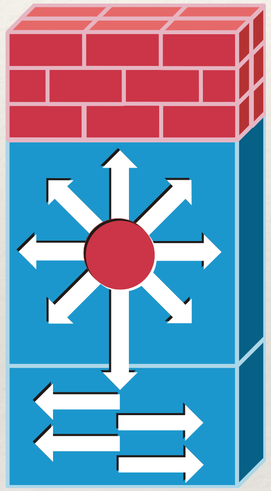
(1) What do we need to log for correct replay?

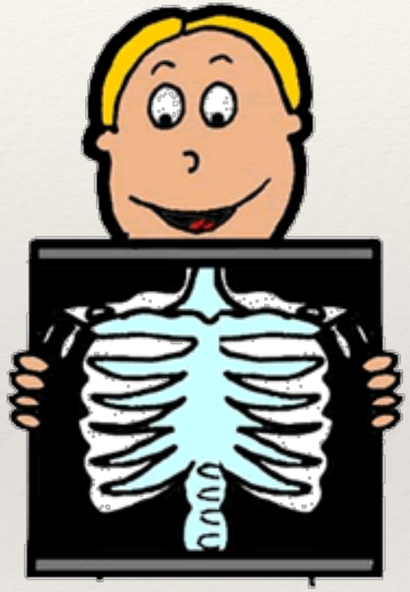
- A classically hard problem due to nondeterminism.

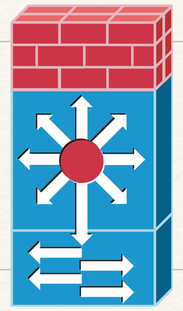
(2) How do we check that we have everything we need to replay a given packet?

- Need to monitor system state that is updated frequently and on multiple cores.

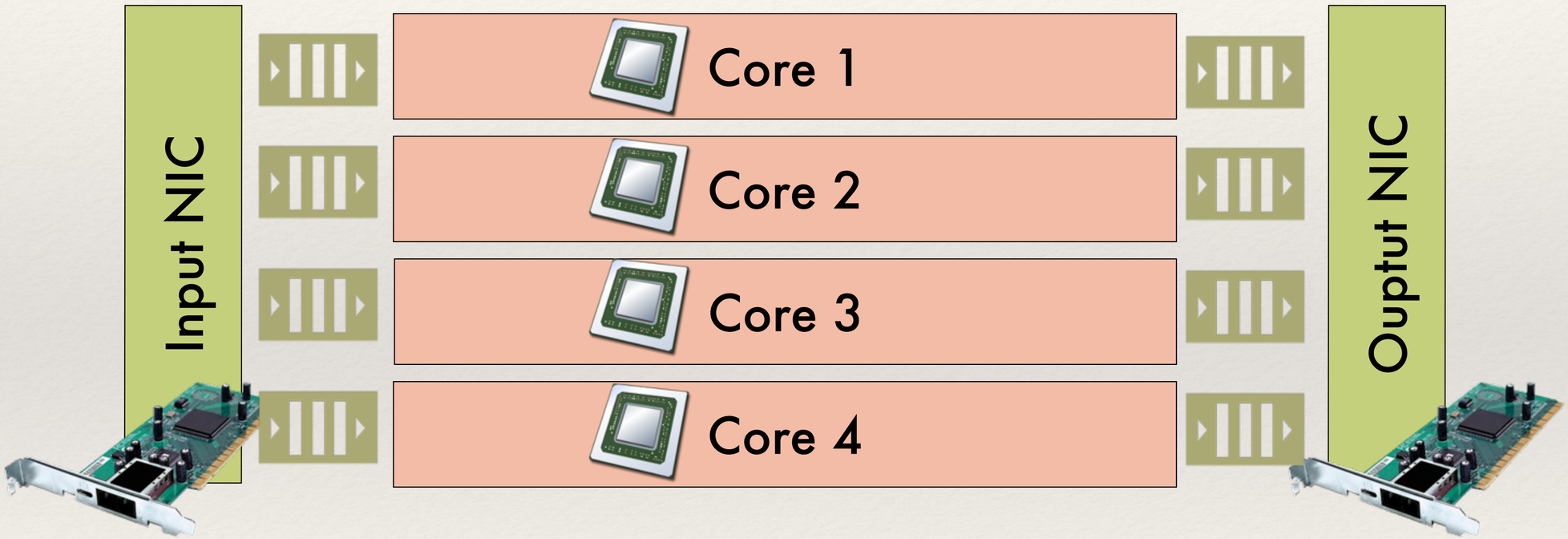
Quick Intro: Middlebox Architecture

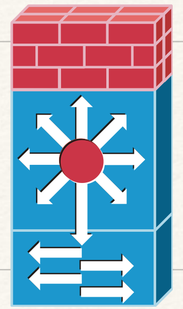






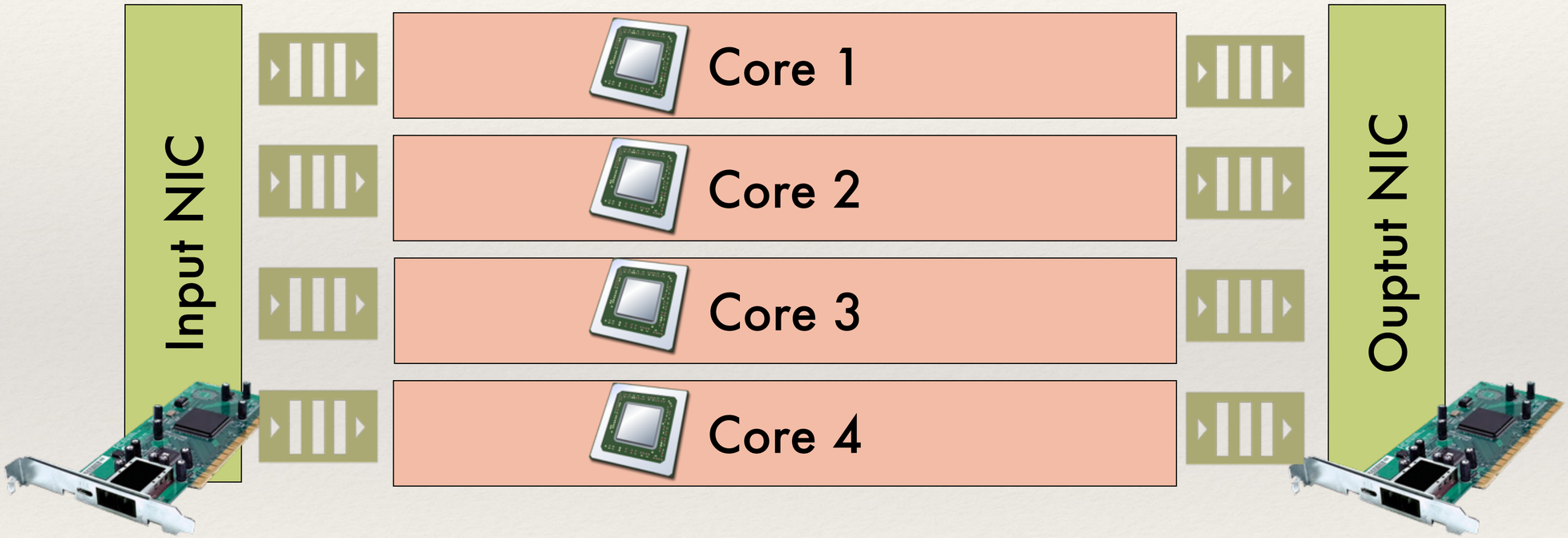
Middlebox Architecture



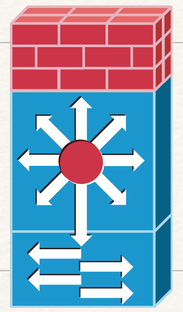


Middlebox Architecture

Input NIC "hashes" incoming packets to cores.

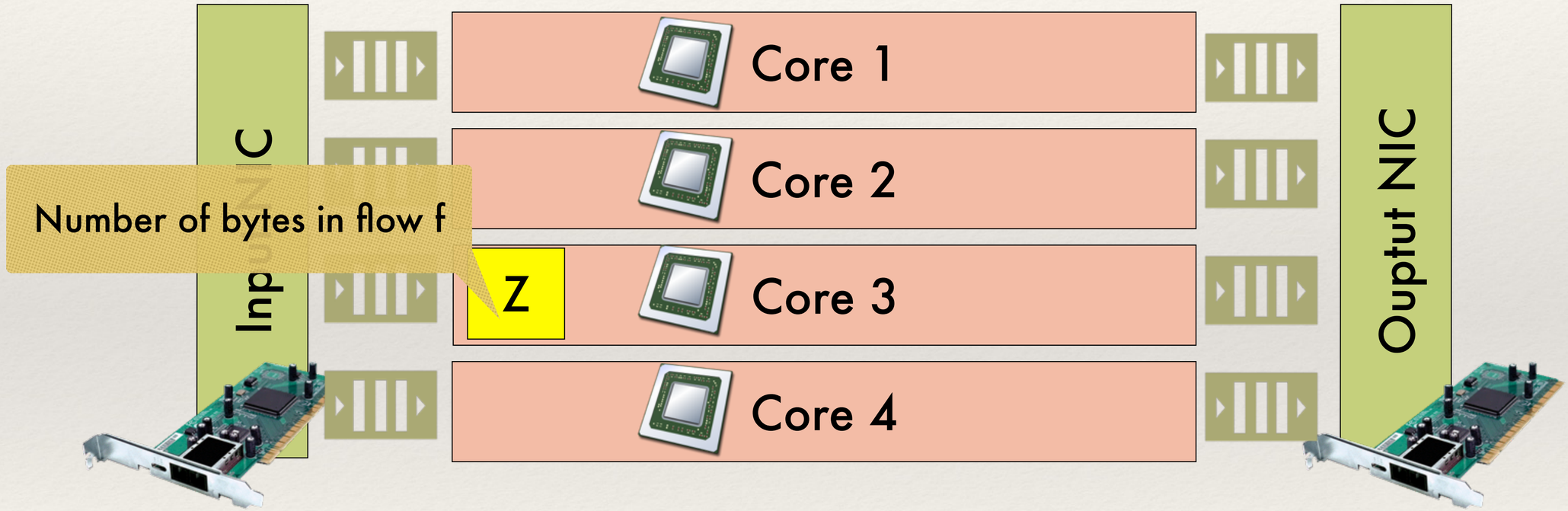


All packets from same flow are processed by same core.



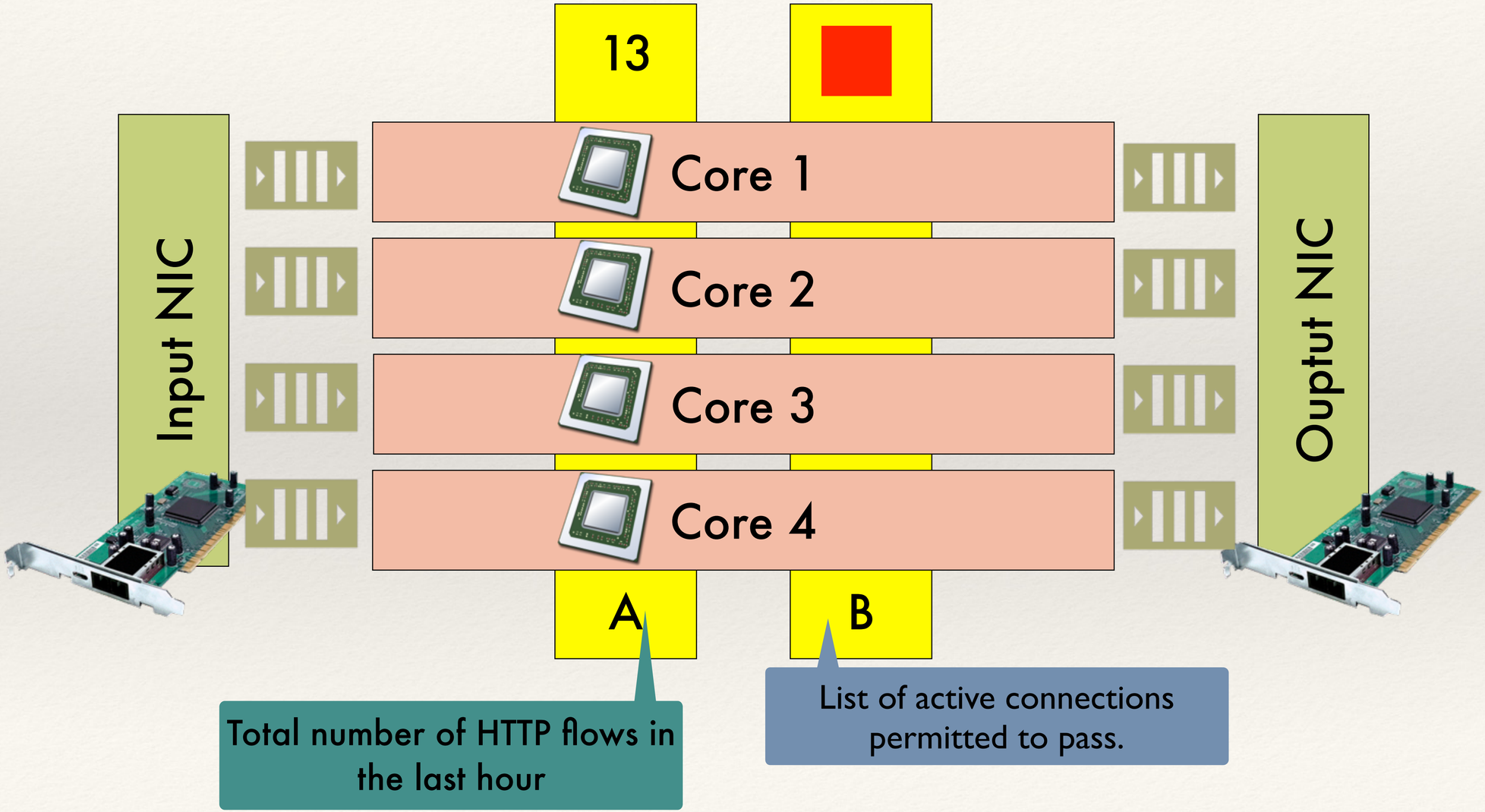
Middlebox Architecture: State

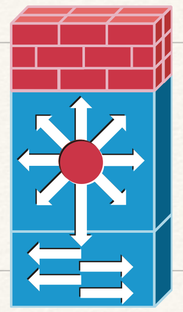
Local state: only relevant to one connection.



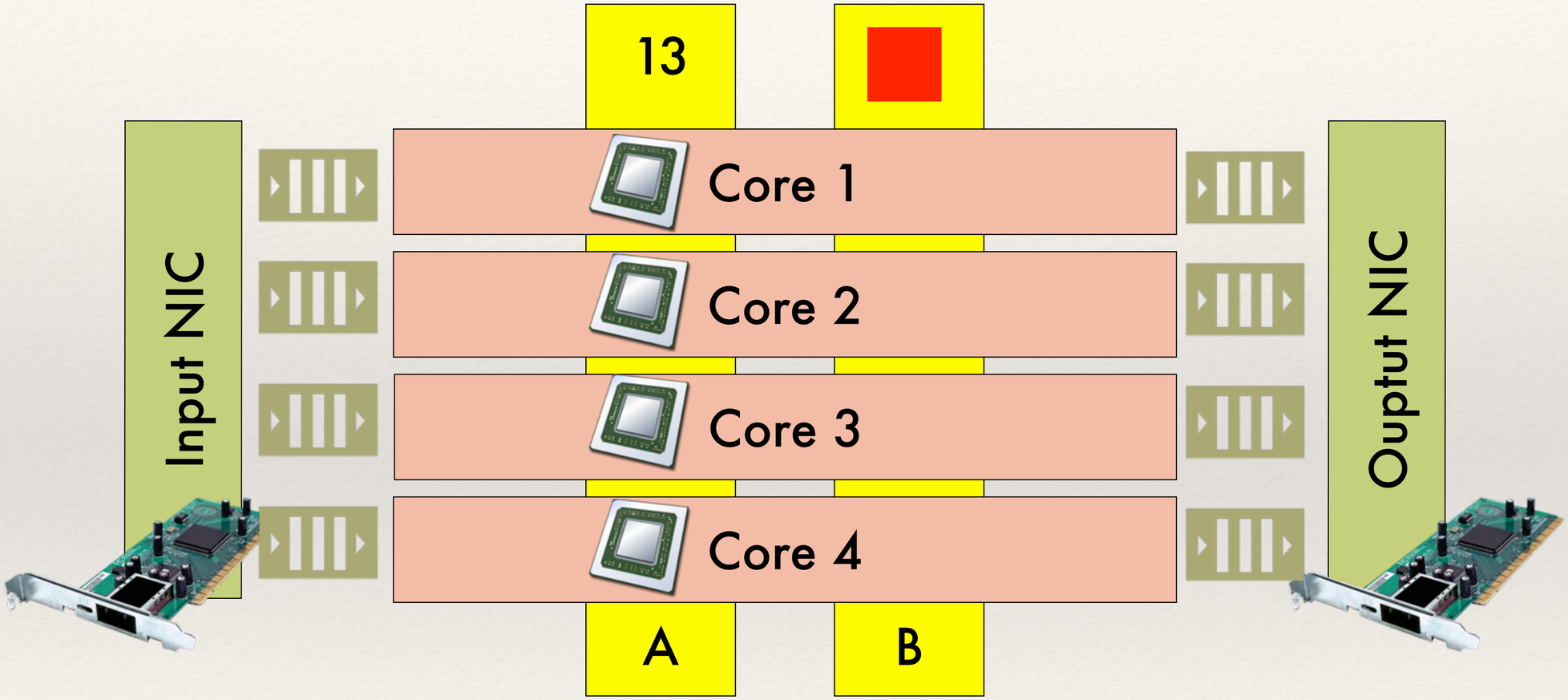
Accessing local state is *fast* because only one core "owns" the data.

Middlebox Architecture: State





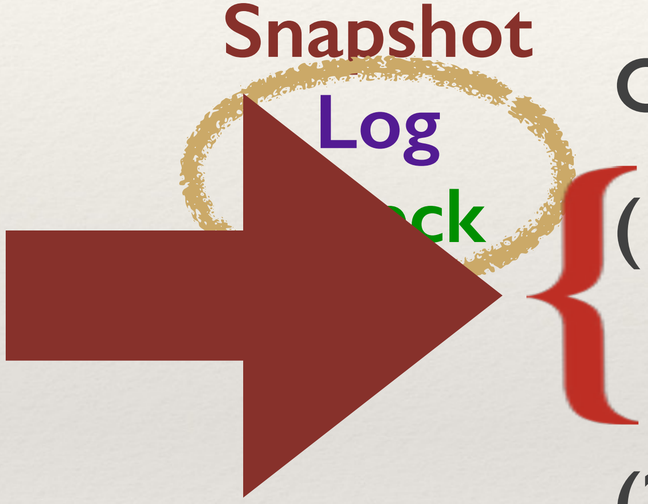
Middlebox Architecture: State



Reading shared state is slower.
Writing is most expensive because it can cause *contention!*

Rollback Recovery

Three Part Algorithm:



Open Questions:

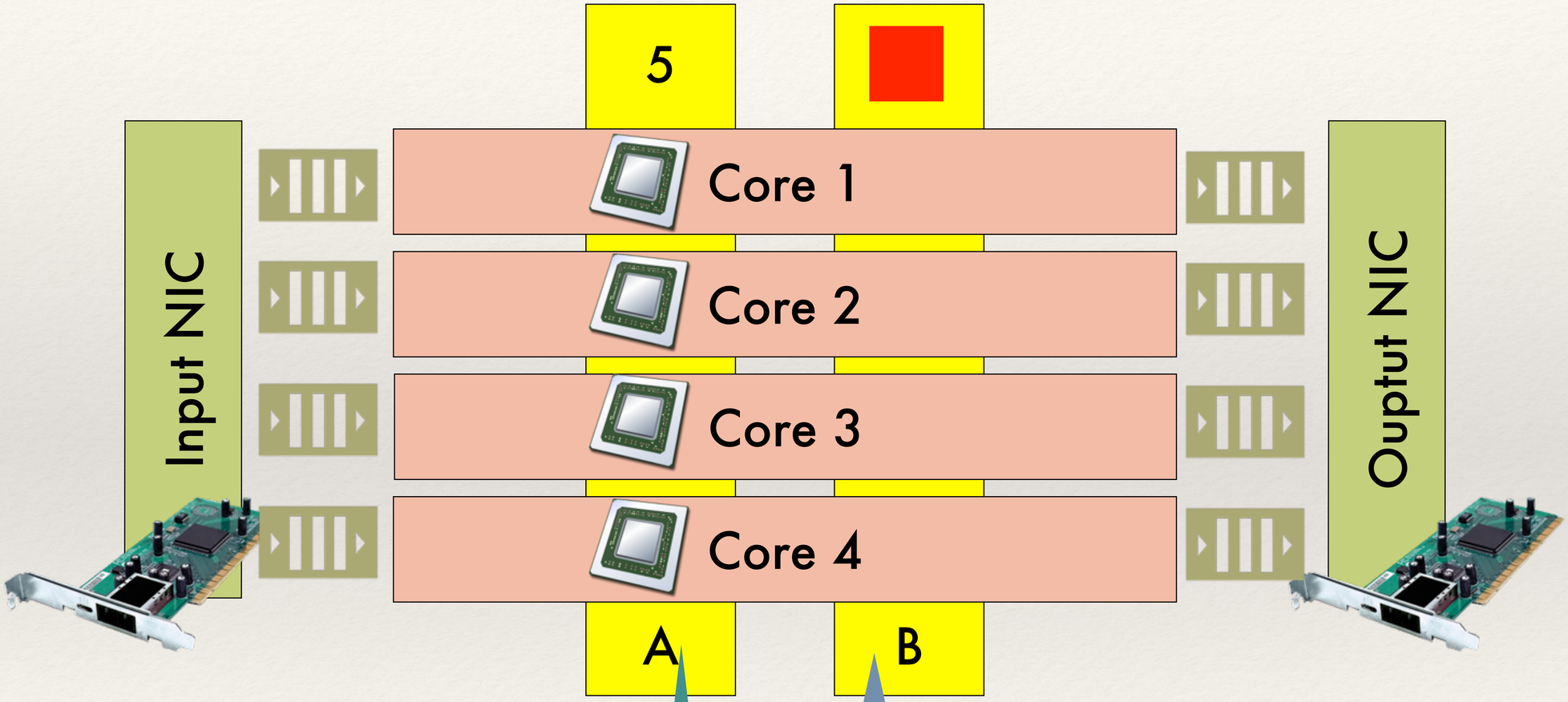
(1) What do we need to log for correct replay?

- A classically hard problem due to nondeterminism.

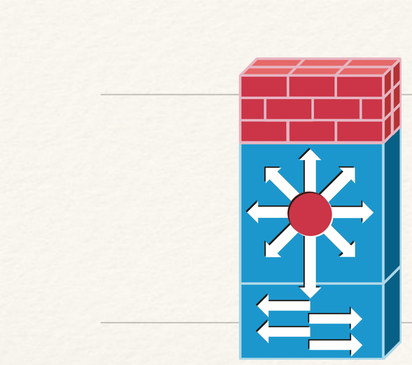
(2) How do we check that we have everything we need to replay a given packet?

- Need to monitor system state that is updated frequently and on multiple cores.

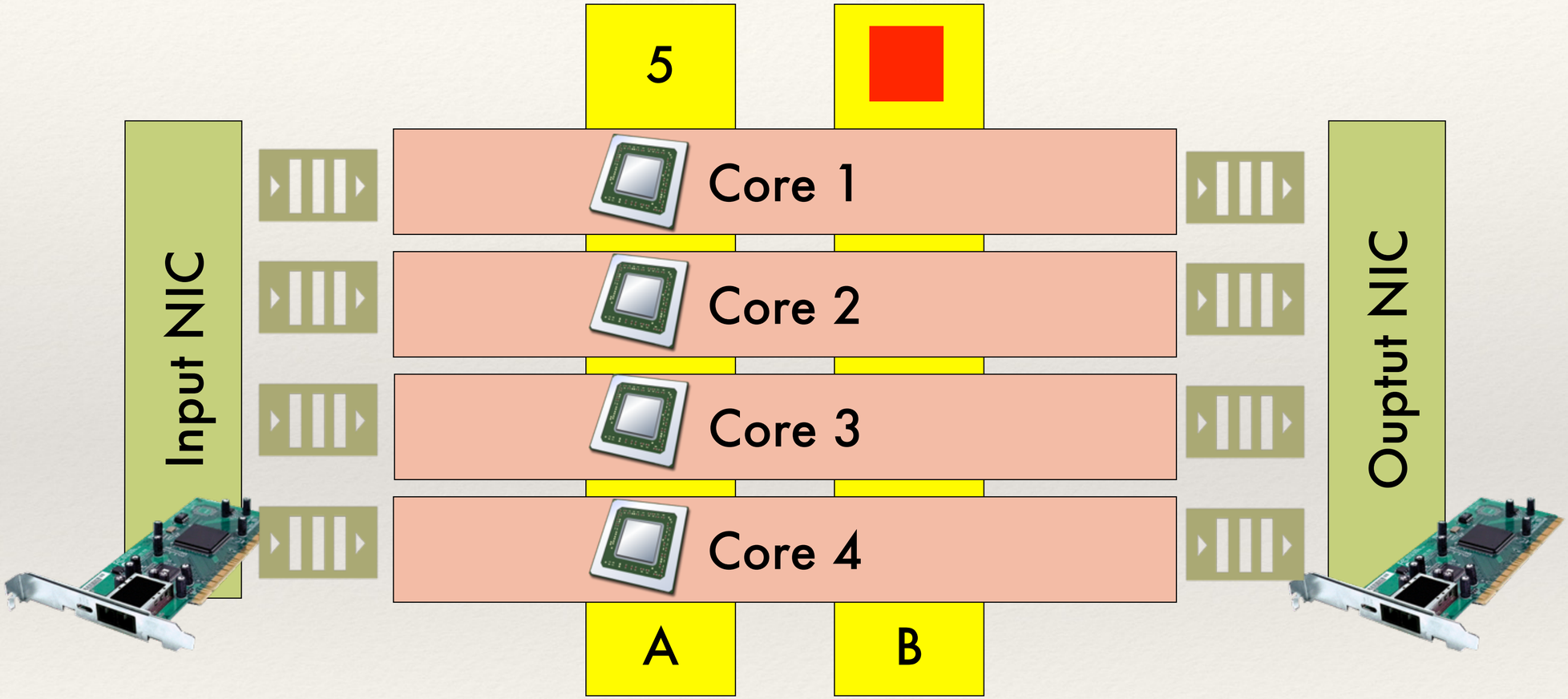
Parallelism + Shared State



connections, unless A >=5.

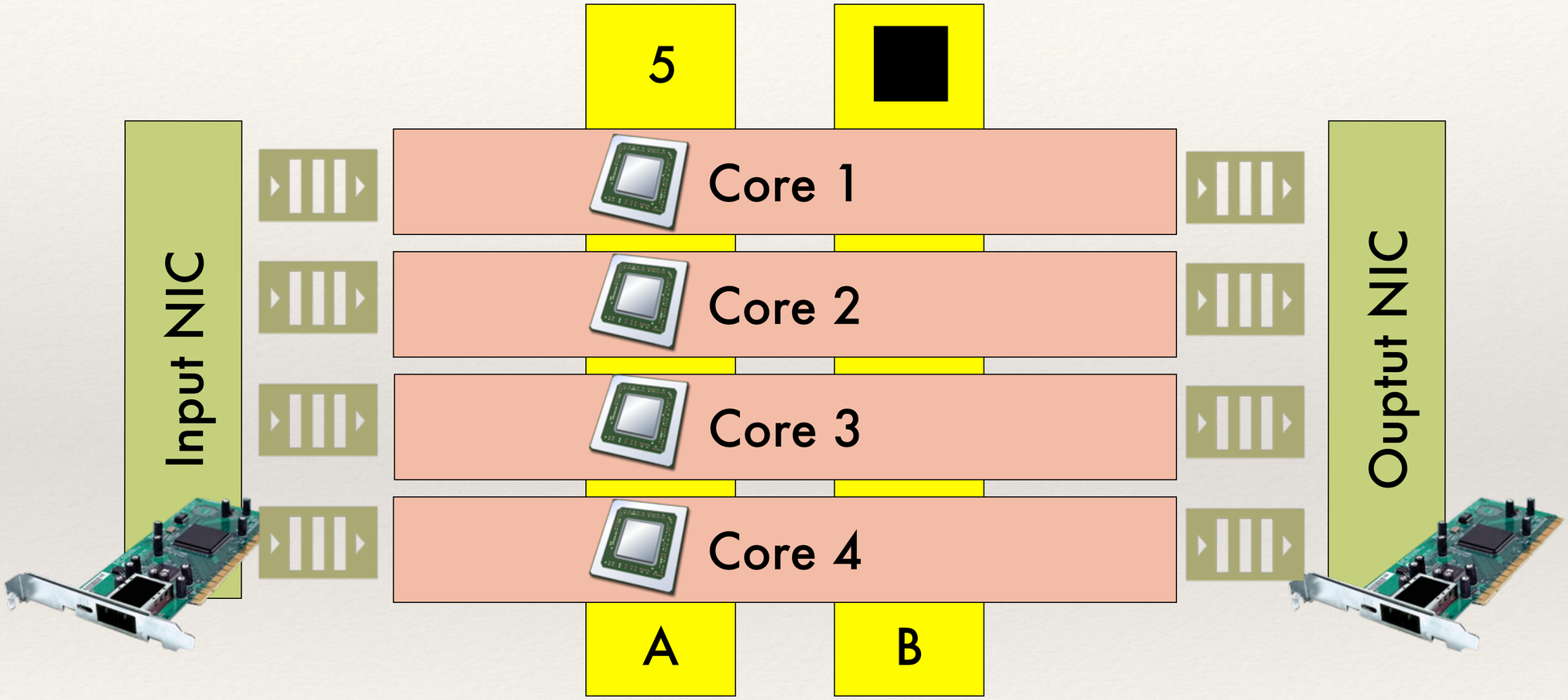


Parallelism + Shared State



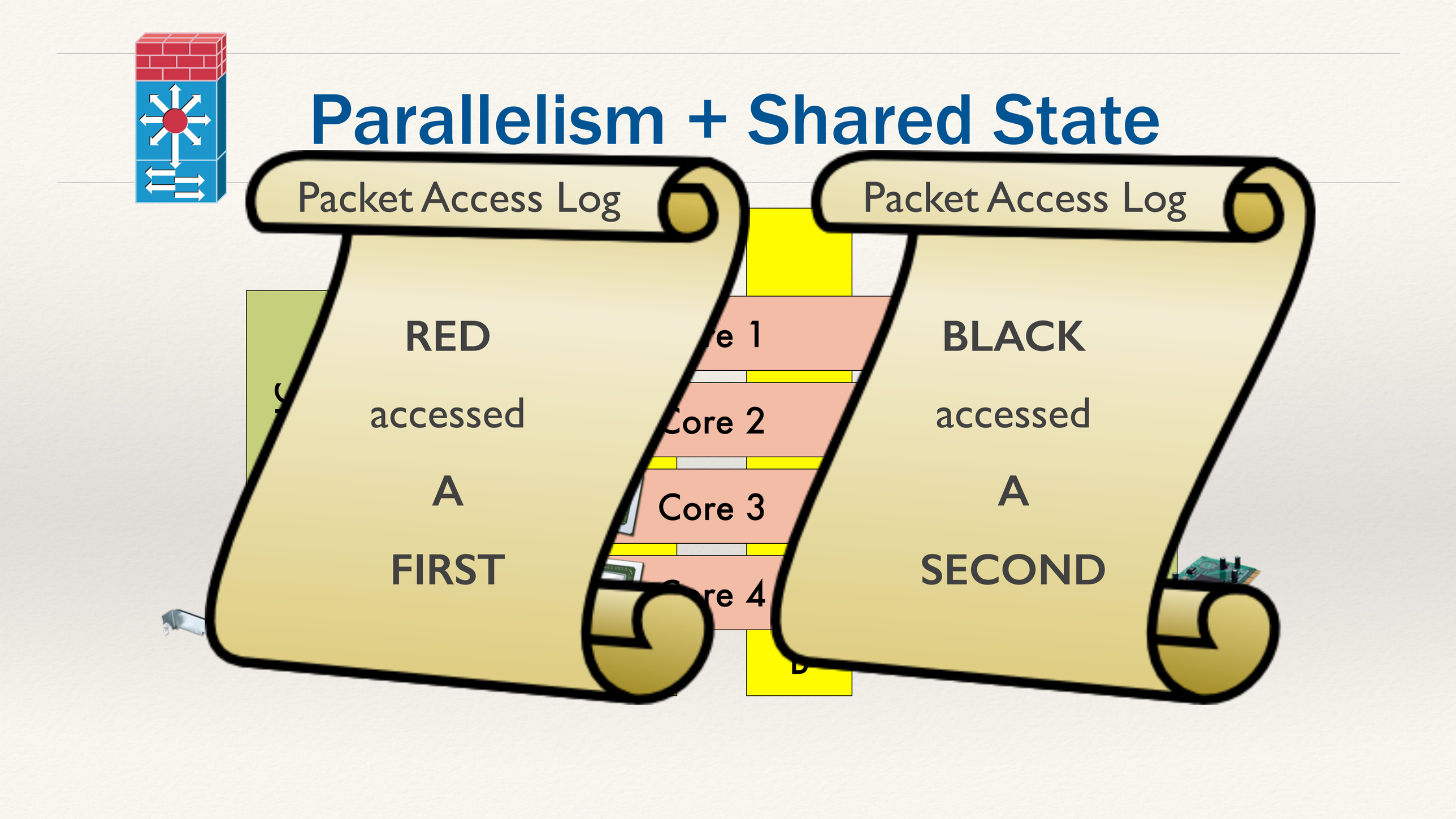
MB Rule: allow new connections, unless A>=5.

Parallelism + Shared State



MB Rule: allow new connections, unless A>=5.

FTMB logs all accesses to shared state using Packet Access Logs (PAL).


Parallelism + Shared State
Packet Access Log
RED
accessed
A
FIRST
Core 1
Core 2
Core 3
Core 4
Packet Access Log
BLACK
accessed
A
SECOND

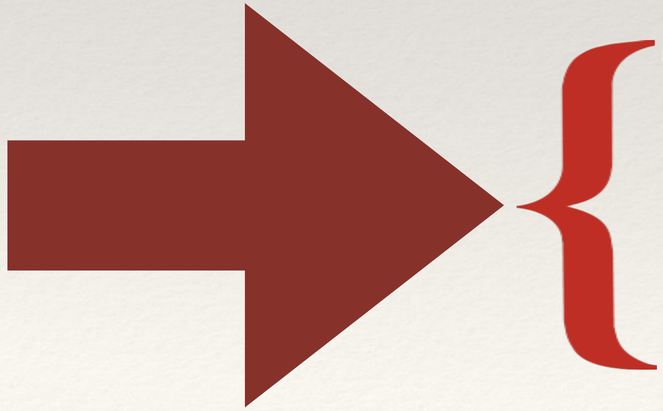
Rollback Recovery

Three Part Algorithm:

Open Questions:

(1) What do we need to log for correct replay?

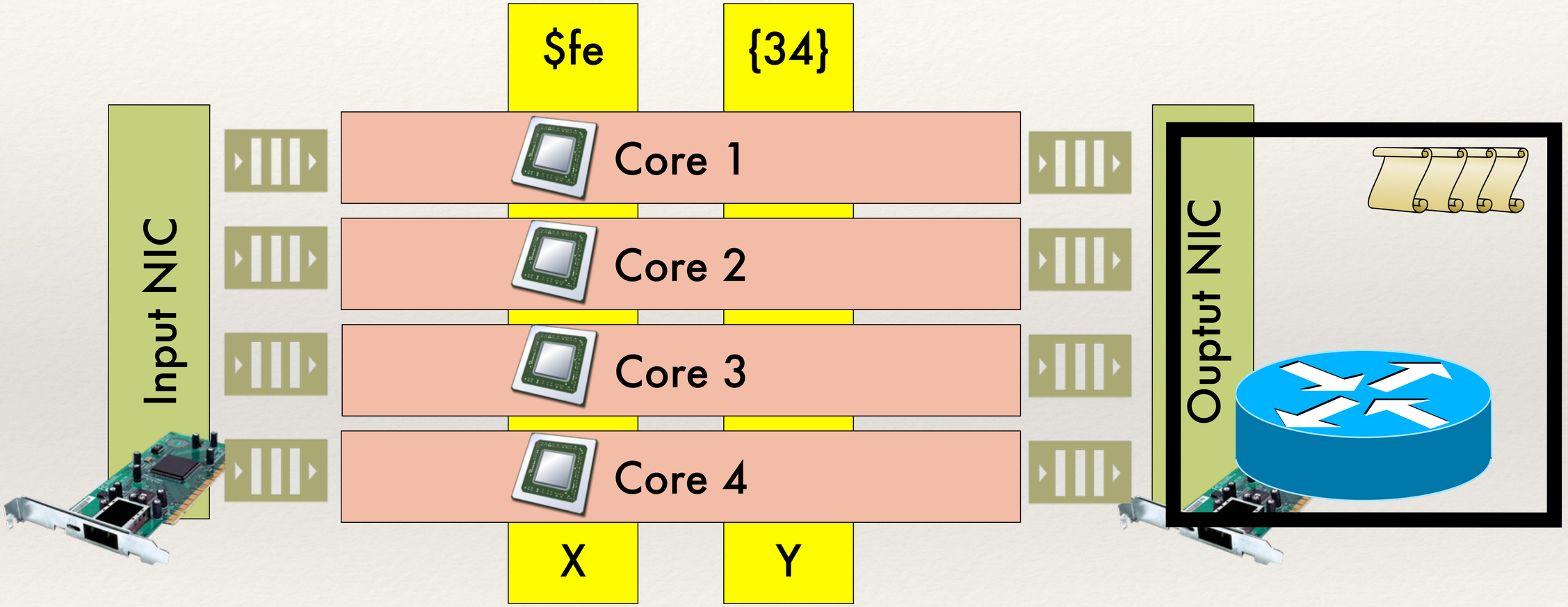
- Packet Access Logs record accesses to shared state.



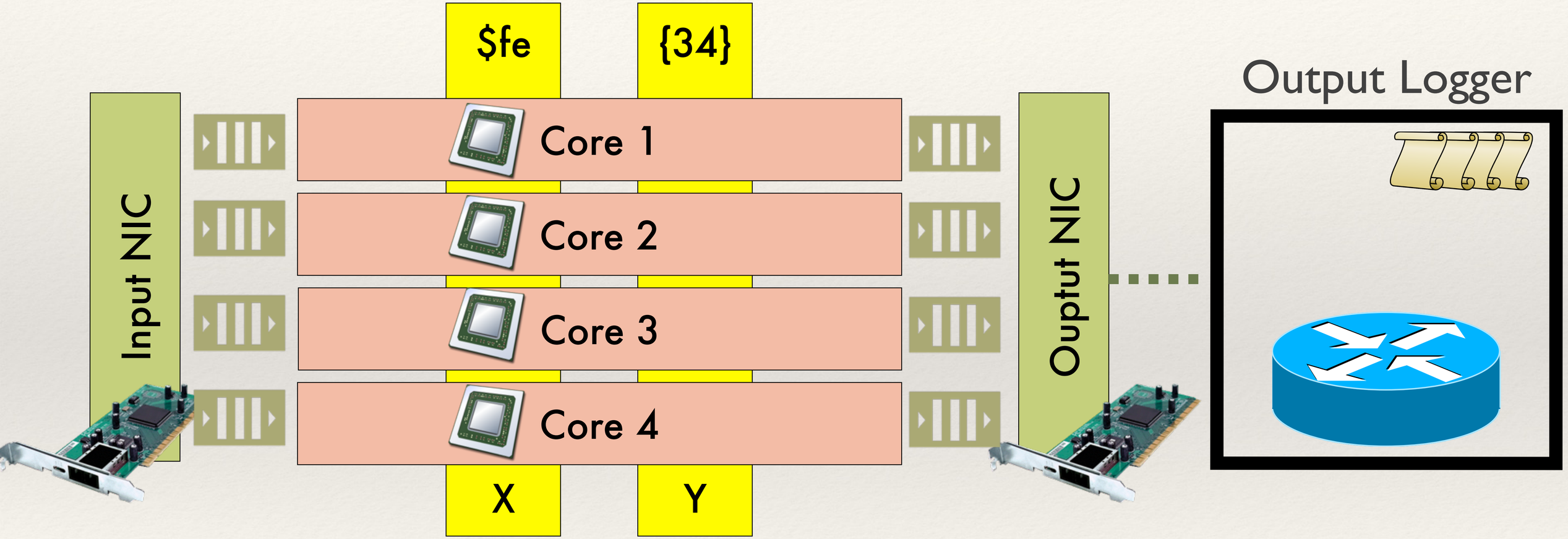
(2) How do we check that we have everything we need to replay a given packet?

- Need to monitor system state that is updated frequently and on multiple cores.

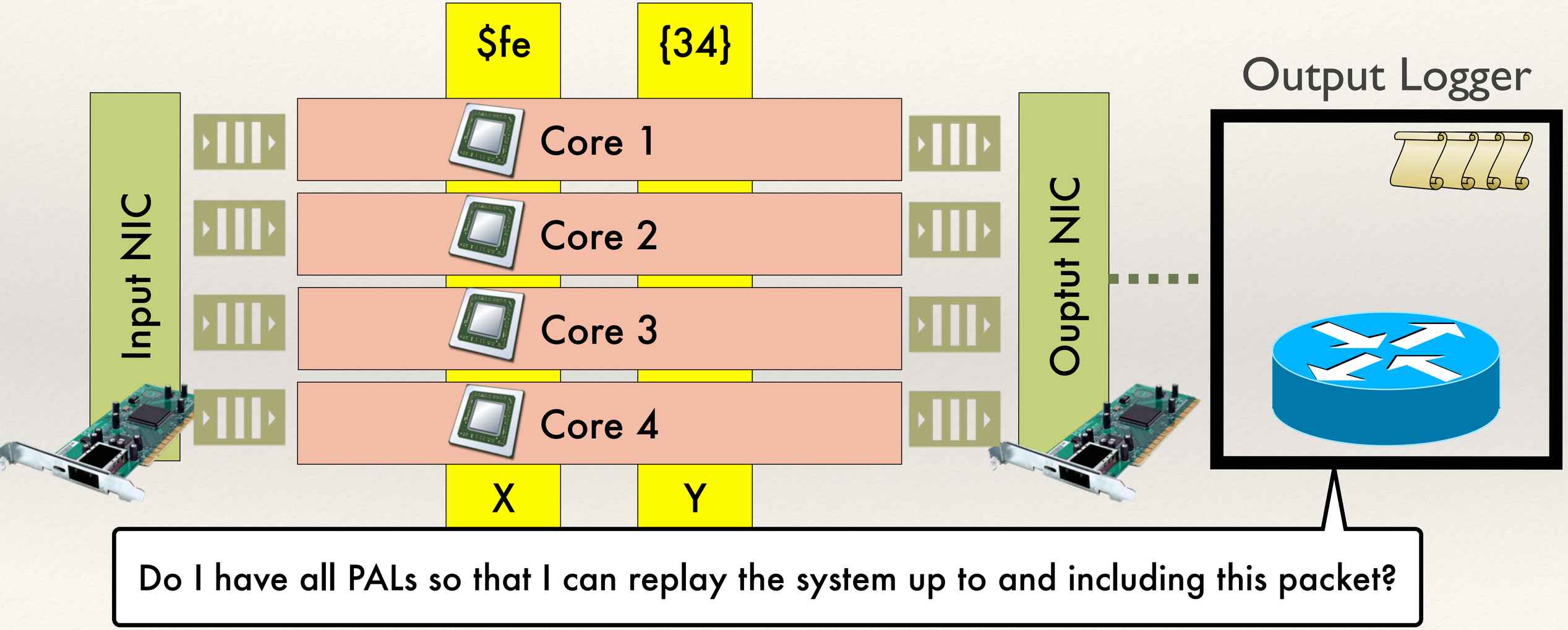
Checking for Safe Release



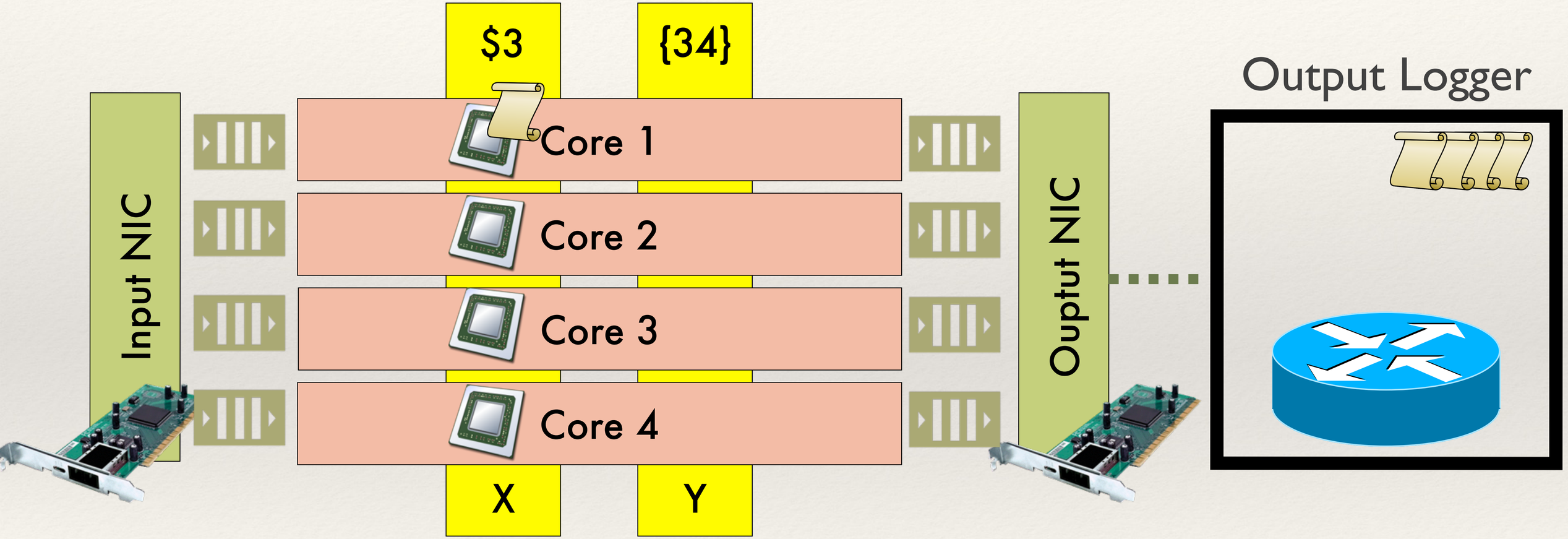
Checking for Safe Release



Checking for Safe Release

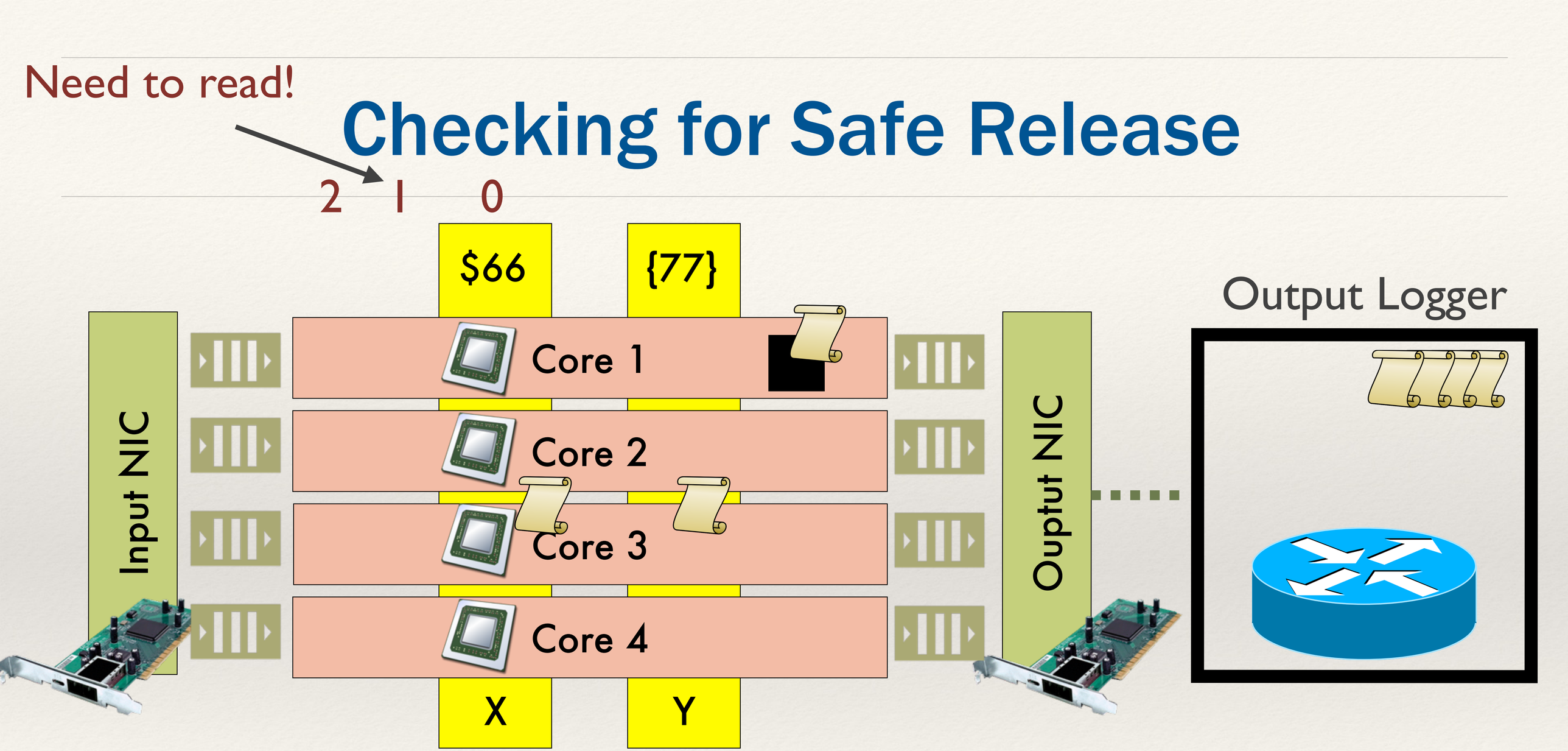


Checking for Safe Release



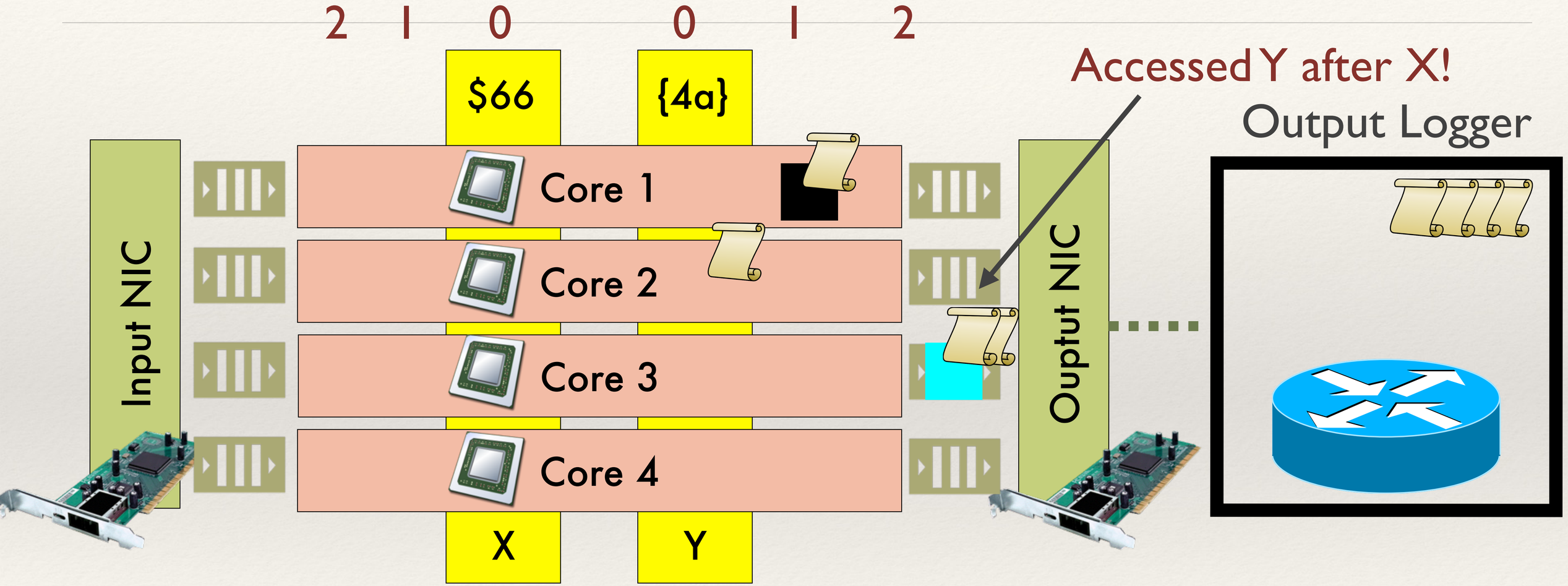
If black packet were released now, would only need PAL {X, Black, First}

Checking for Safe Release



If blue packet were released now, would need its own PALs, and {X, Black, First}

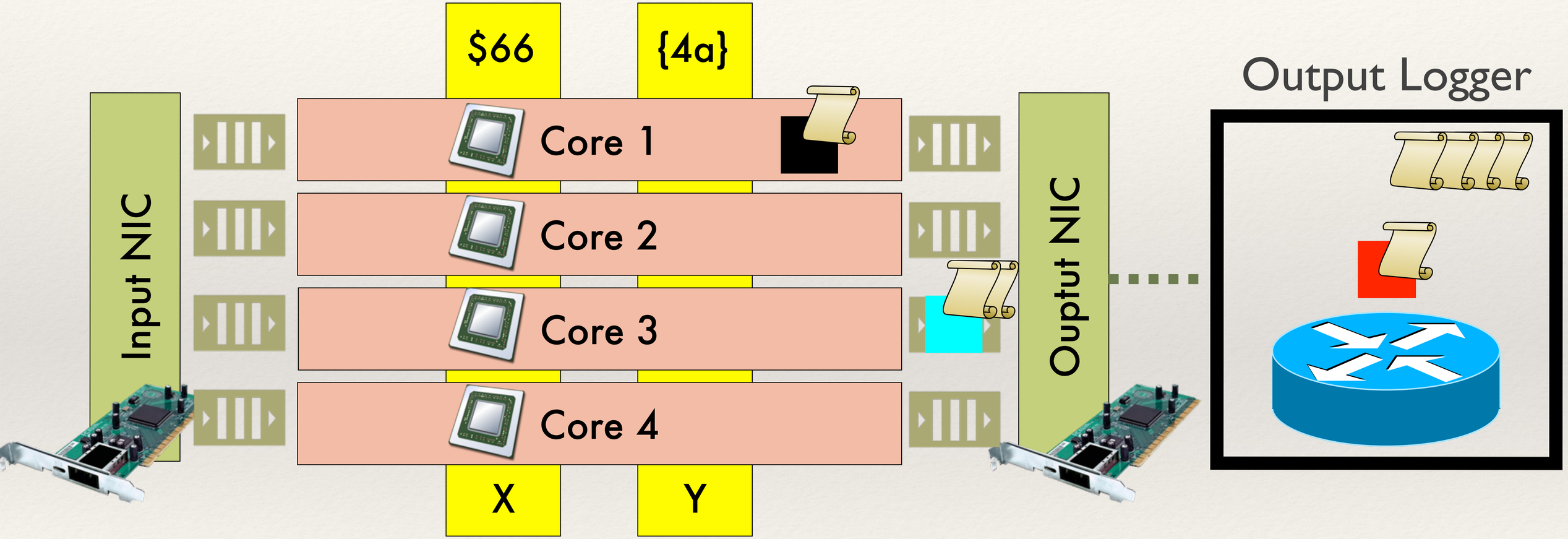
Checking for Safe Release



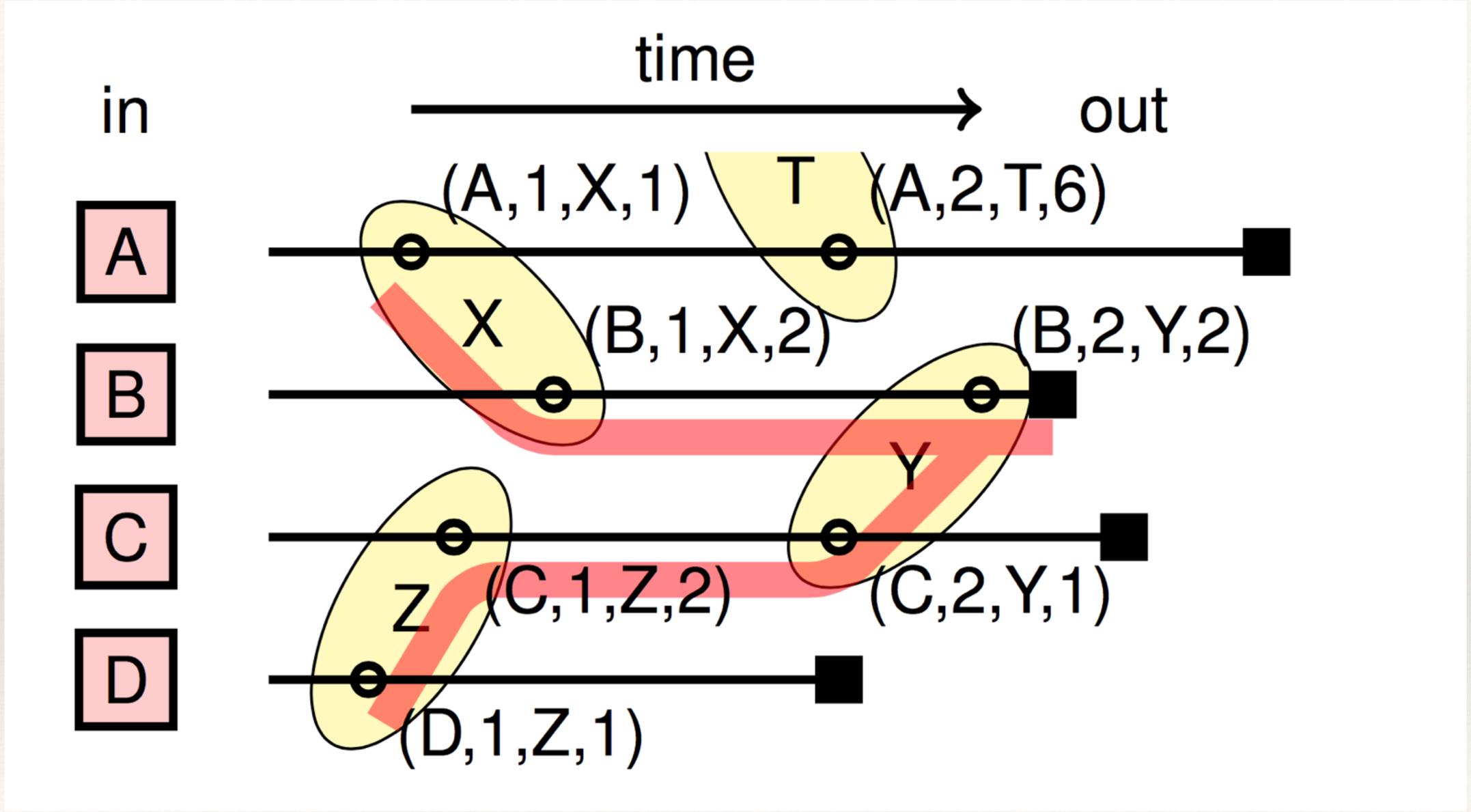
Red packet needs its own PAL, and {Blue, Y, First}

…and {Blue, X, 2nd} and {Black, X, First}

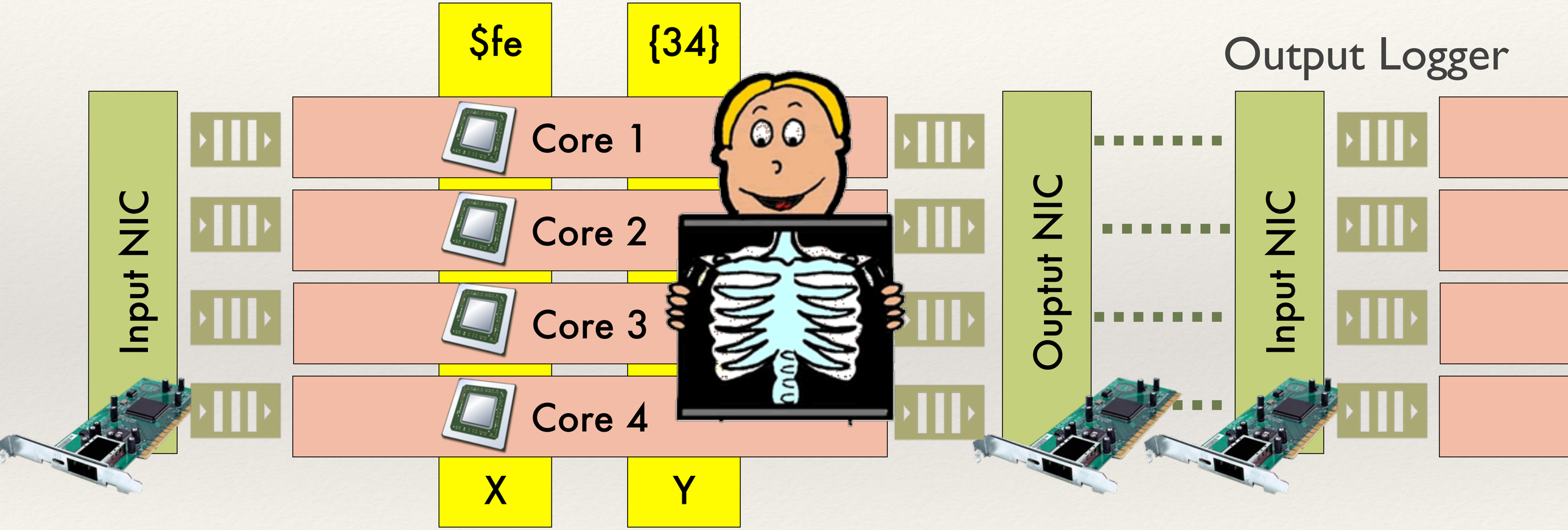
Checking for Safe Release



Can depend on PALs from different cores, for variables packet never accessed!

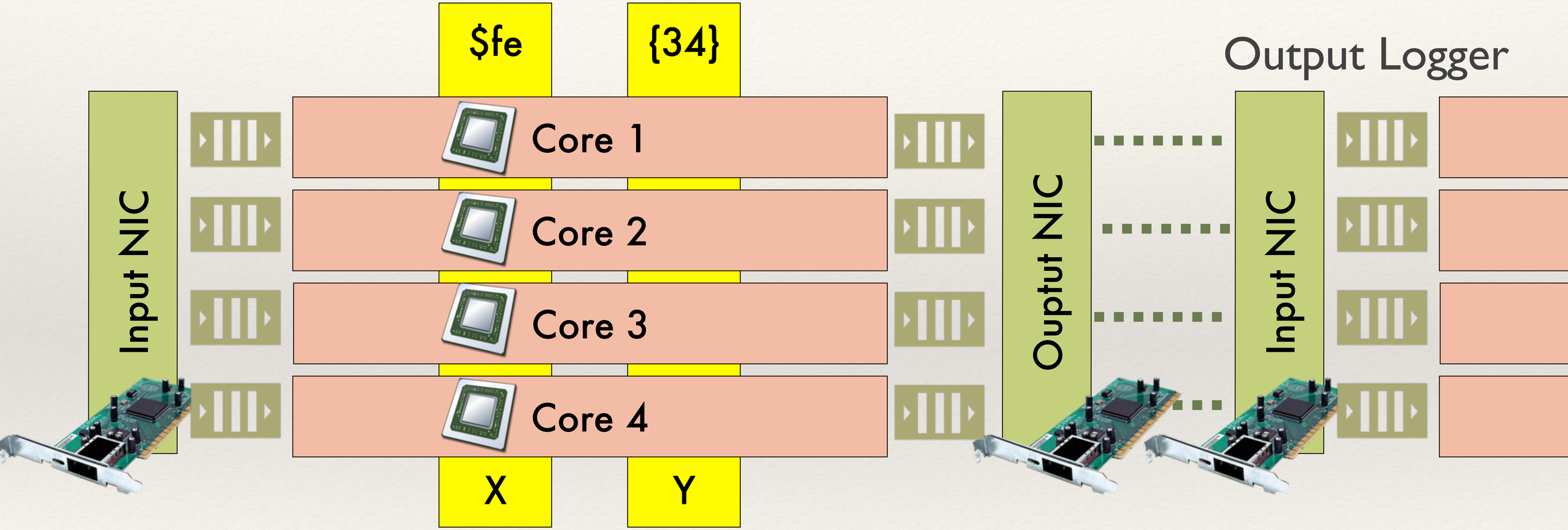

time
in
out
A
B
C
D
(A,1,X,1)
T
(A,2,T,6)
X
(B,1,X,2)
(B,2,Y,2)
Y
Z
(C,1,Z,2)
(C,2,Y,1)
(D,1,Z,1)

Checking for Safe Release



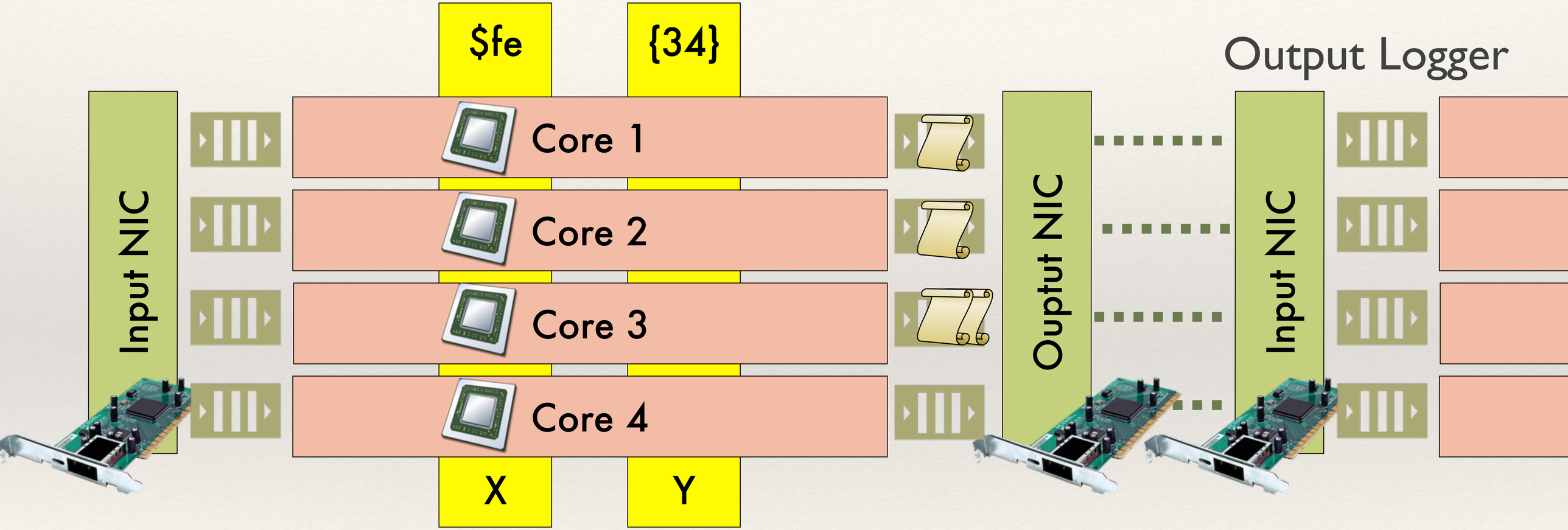
FTMB is O(#cores) and read-only, making it fast.

Ordered Logging and Parallel Release



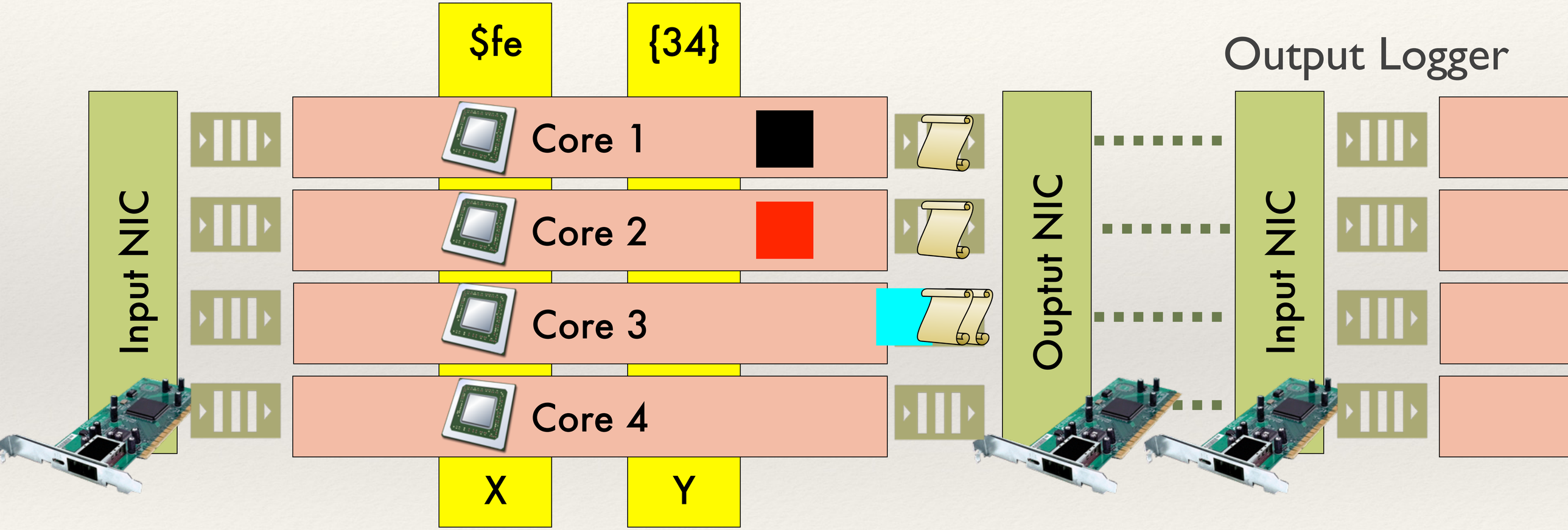
Key Insight: Packet cannot depend on a PAL that does not exist yet.

Ordered Logging and Parallel Release



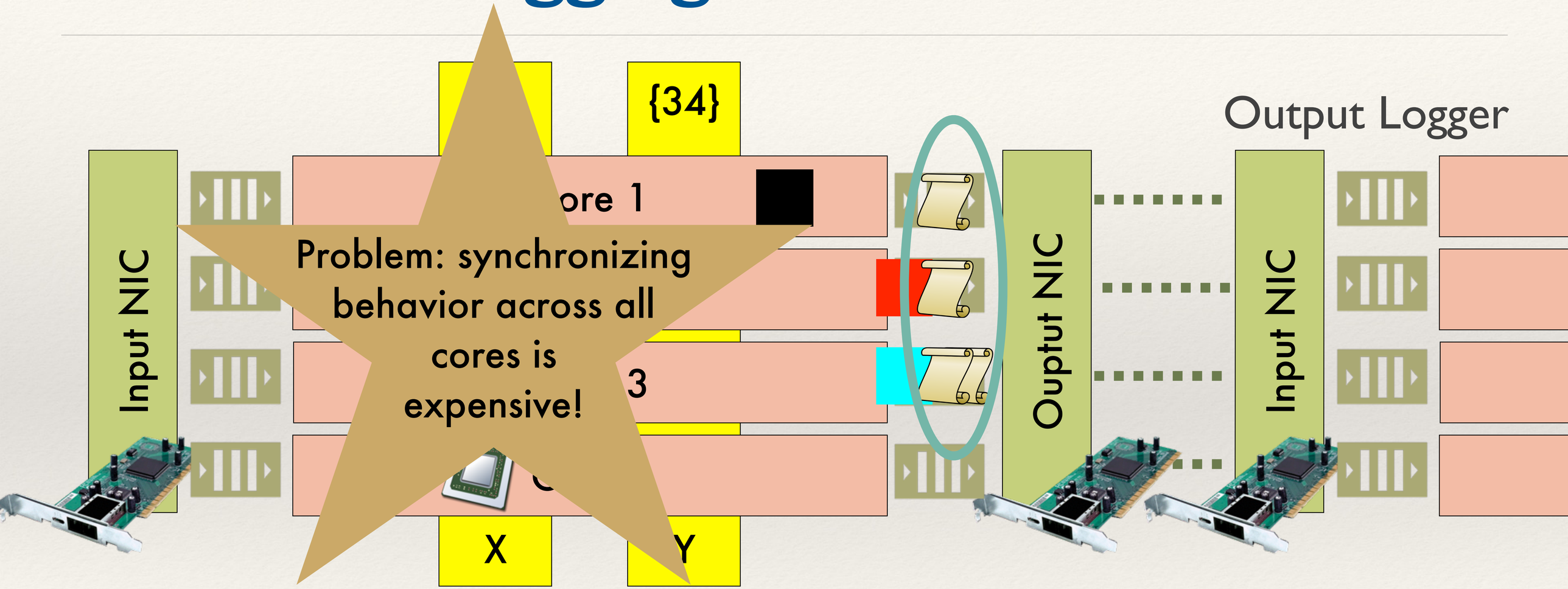
Key Insight: Packet cannot depend on a PAL that does not exist yet.
PALs are written to output queues immediately when created.

Ordered Logging and Parallel Release



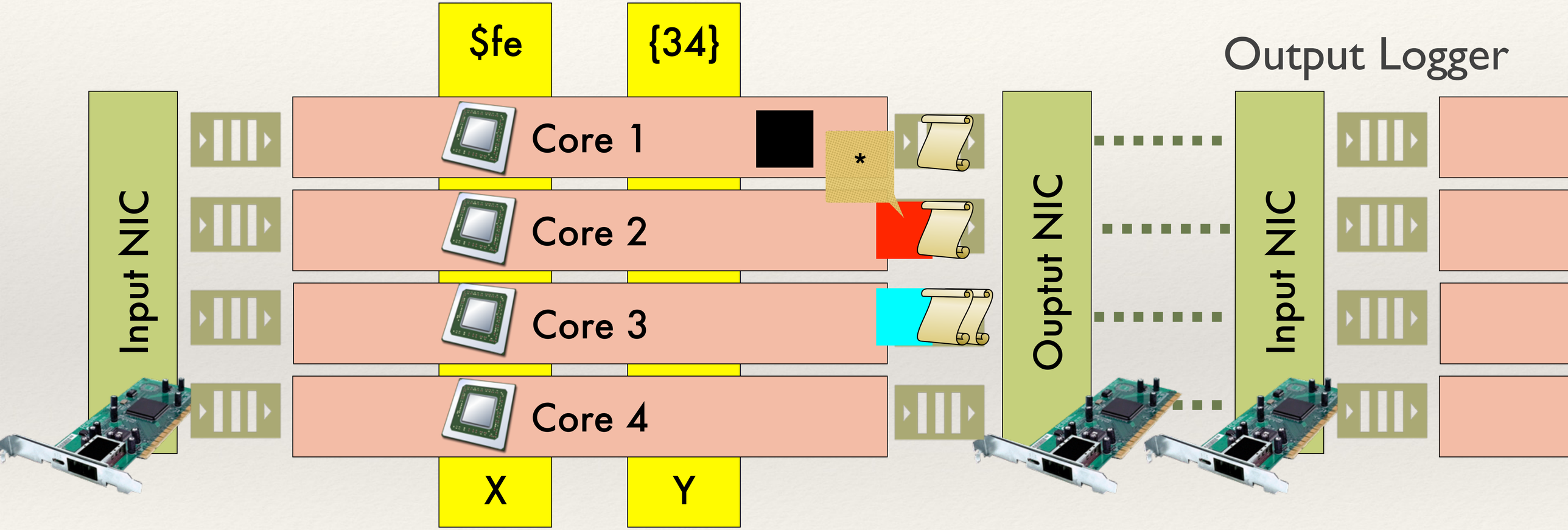
When packet arrives at output queue, all PALs it depends on are already enqueued; or are already at output logger.

Ordered Logging and Parallel Release



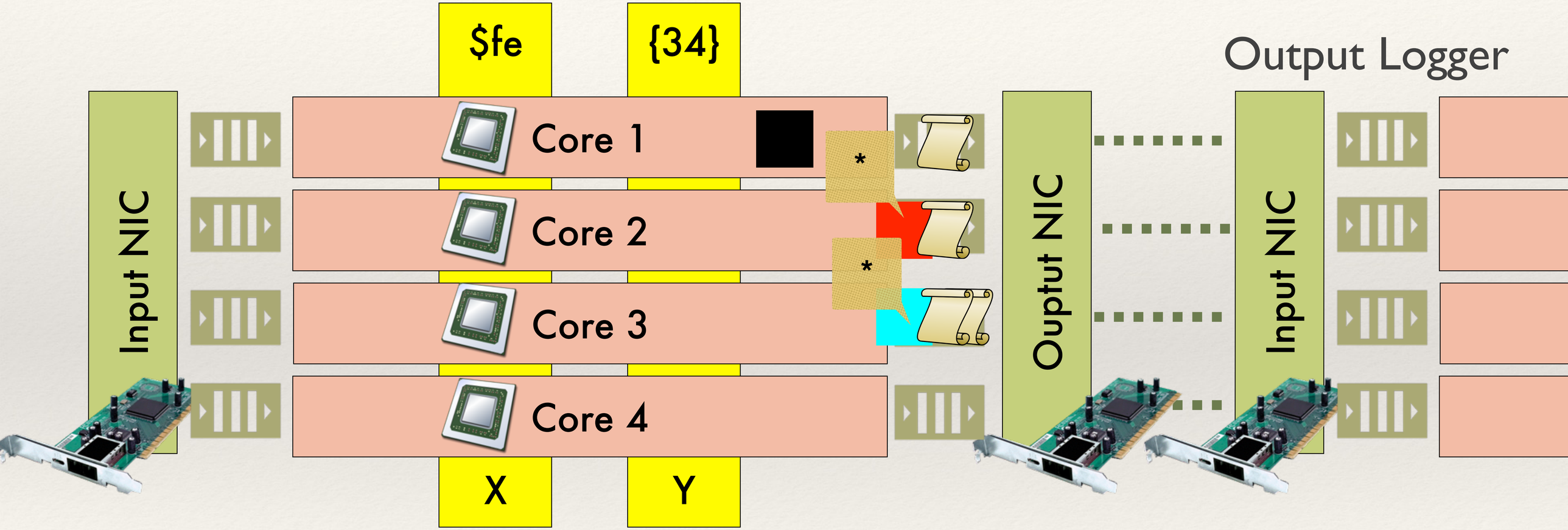
What we want: "flush" all PALs to Output Logger. Then we're done!

Ordered Logging and Parallel Release



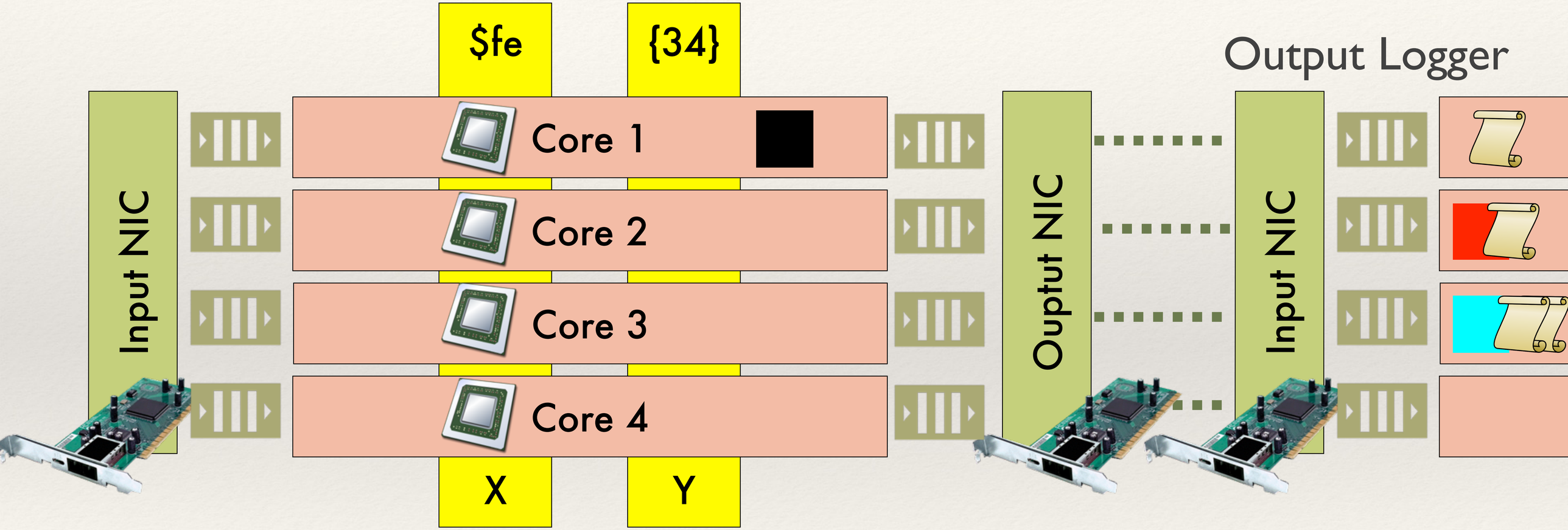
Each core keeps a counter tracking the "youngest" PAL it has created.
On release, packet *reads* counters across all cores. (O(#cores) reads)

Ordered Logging and Parallel Release

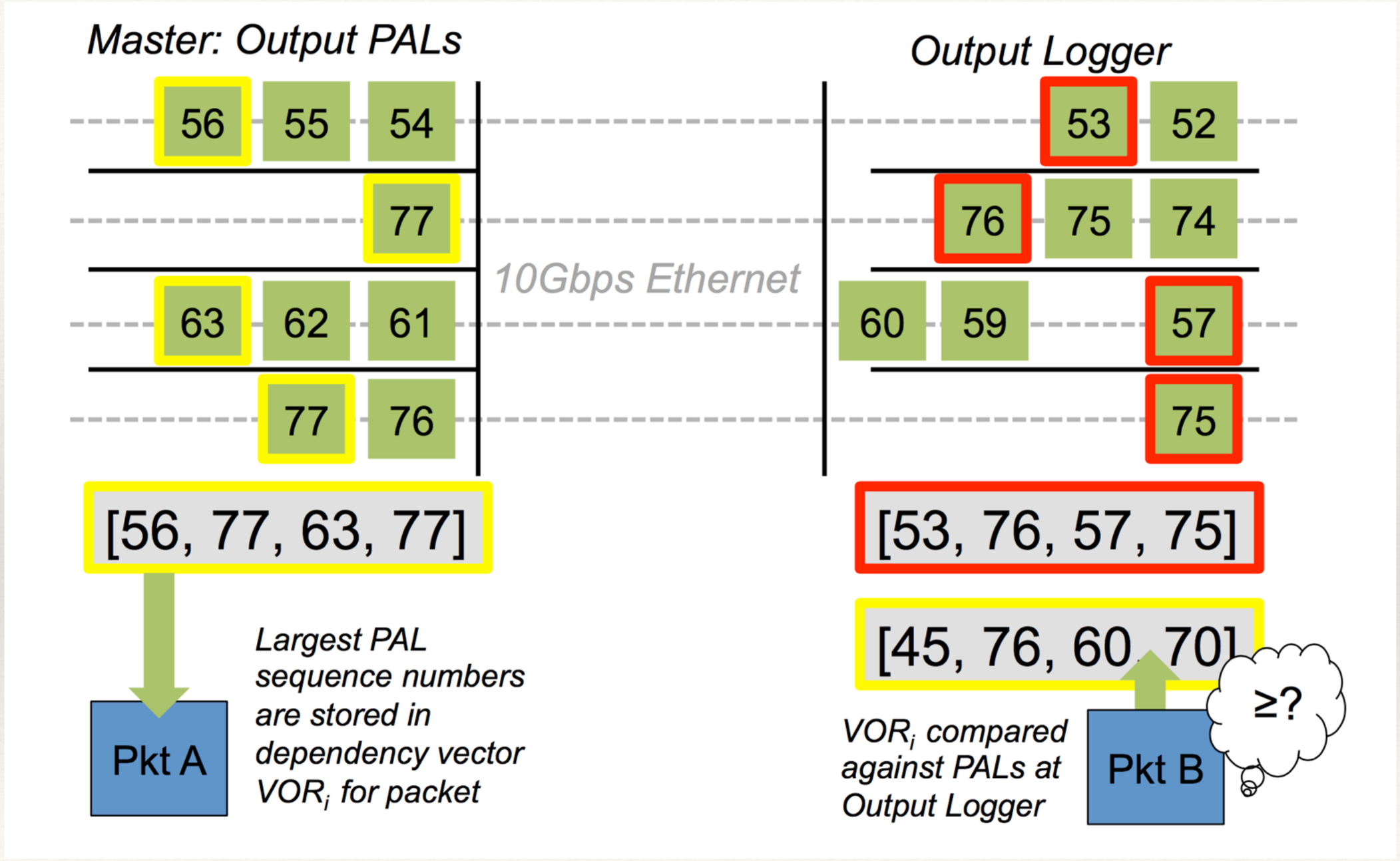


Output logger keeps counter representing max PALs received.
Receive packet: reads marker to compare against other cores' counters.

Ordered Logging and Parallel Release



If marker <= all counters, can release packet!


Master: Output PALs
56
55
54
77
63
62
61
77
76
[56, 77, 63, 77]
Pkt A
Largest PAL sequence numbers are stored in dependency vector VOR$_i$ for packet
10Gbps Ethernet
Output Logger
53
52
76
75
74
60
59
57
75
[53, 76, 57, 75]
[45, 76, 60, 70]
VOR$_i$ compared against PALs at Output Logger
Pkt B
≥?

Ordered Logging and Parallel Release

- Parallel! Threads are never blocked on each other to make progress.
- Cross-core accesses are *read only*.
  - Further amortized by batching.
- Linear: order # threads reads to perform.
- Fine-grained. Can make this decision with every packet release.

Rollback Recovery

Three Part Algorithm:

Open Questions:

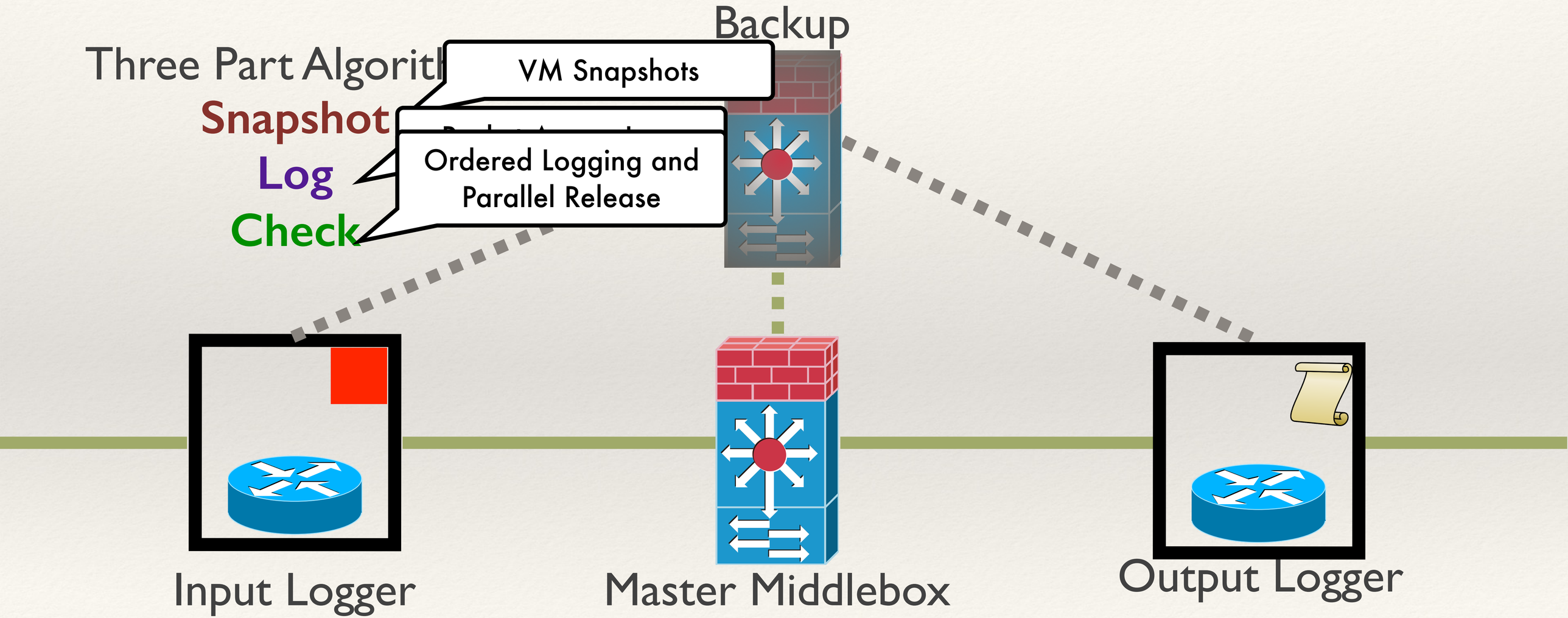
(1) What do we need to log for correct replay?

- Packet Access Logs record accesses to shared state.

(2) How do we check that we have everything we need to replay a given packet?

- Ordered logging and parallel release are read-only, O(#cores) cross core reads per release.

Recap

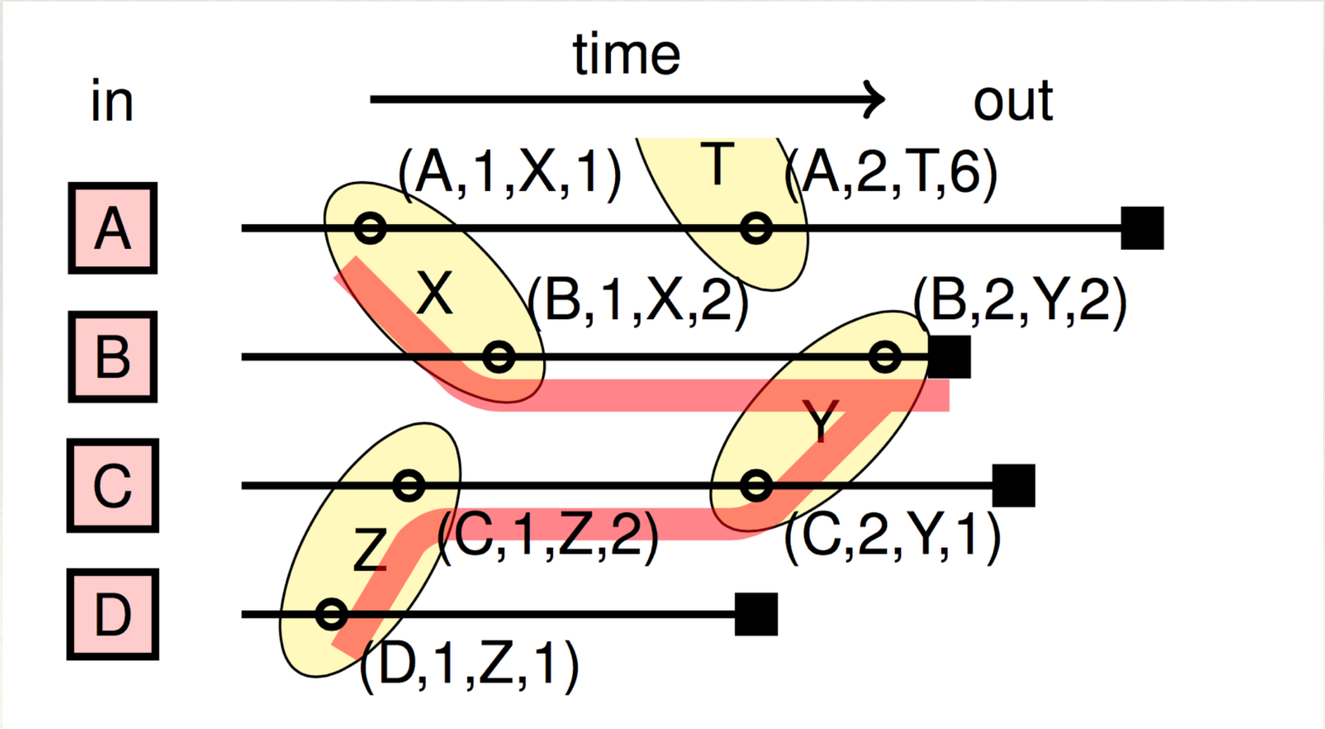

Backup
Three Part Algorith
Snapshot
Log
Check
VM Snapshots
Ordered Logging and
Parallel Release
Input Logger
Master Middlebox
Output Logger

Replay

- Replica starts from the last available snapshot.
- The recorded packets are fed by the Input logger.
- The threads of the replica use the PALs to drive nondeterministic choices.
  - When acquiring the lock that protects a shred variable, the PALs comes into play.
  - It checks whether it can access the lock, or it has to block waiting for some other thread that came earlier in the original execution.
- On output, packets are passed to the OL
  - The OL discards them if a previous instance and been already released.
- A thread exists replay mode when it finds that there are no more PALs for its shared variables

Replay

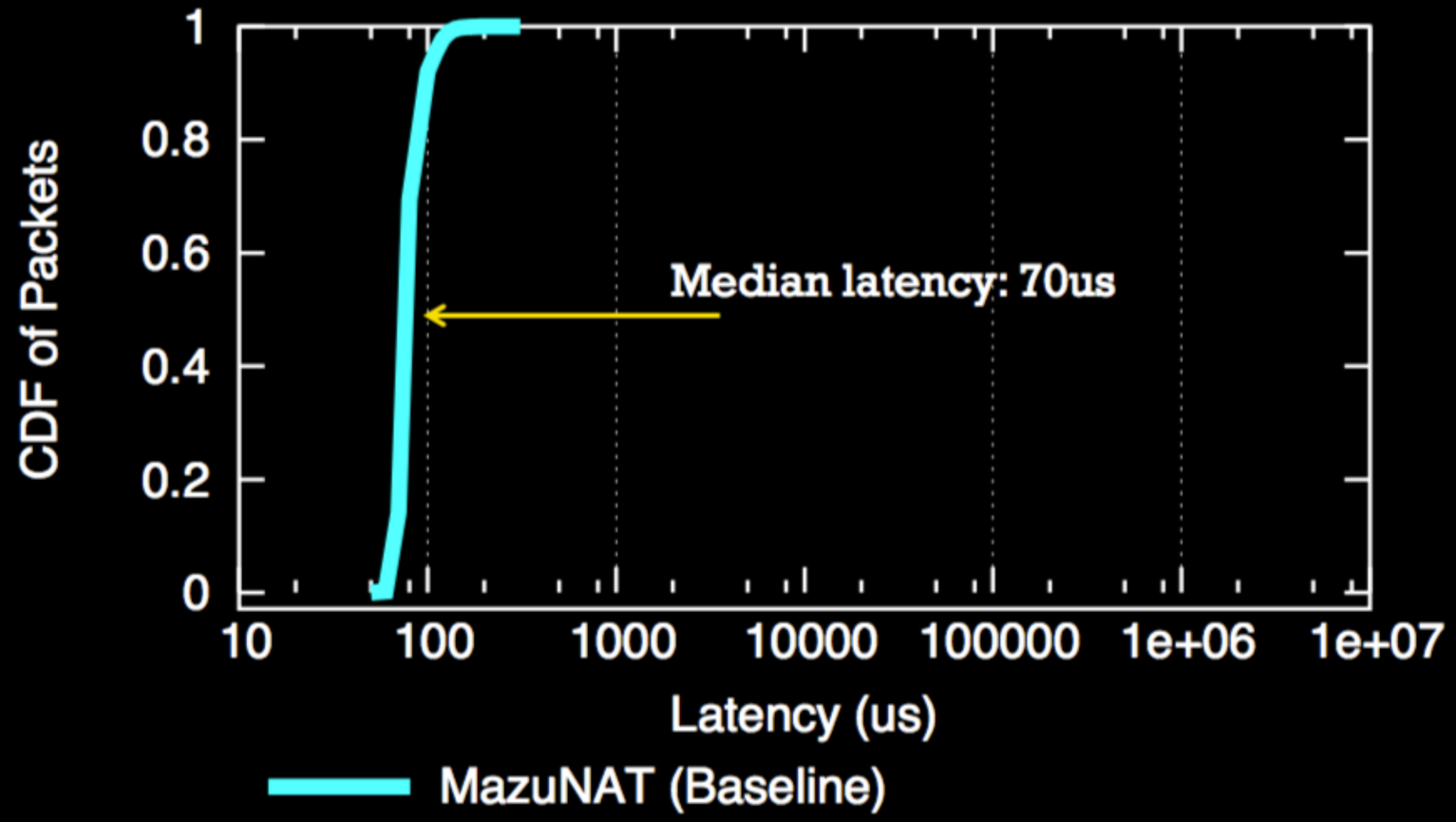
- The threads of the replica use the PALs to drive nondeterministic choices.
  - When acquiring the lock that protects a shred variable, the PALs comes into play.
  - It checks whether it can access the lock, or it has to block waiting for some other threads that came earlier in the original execution.



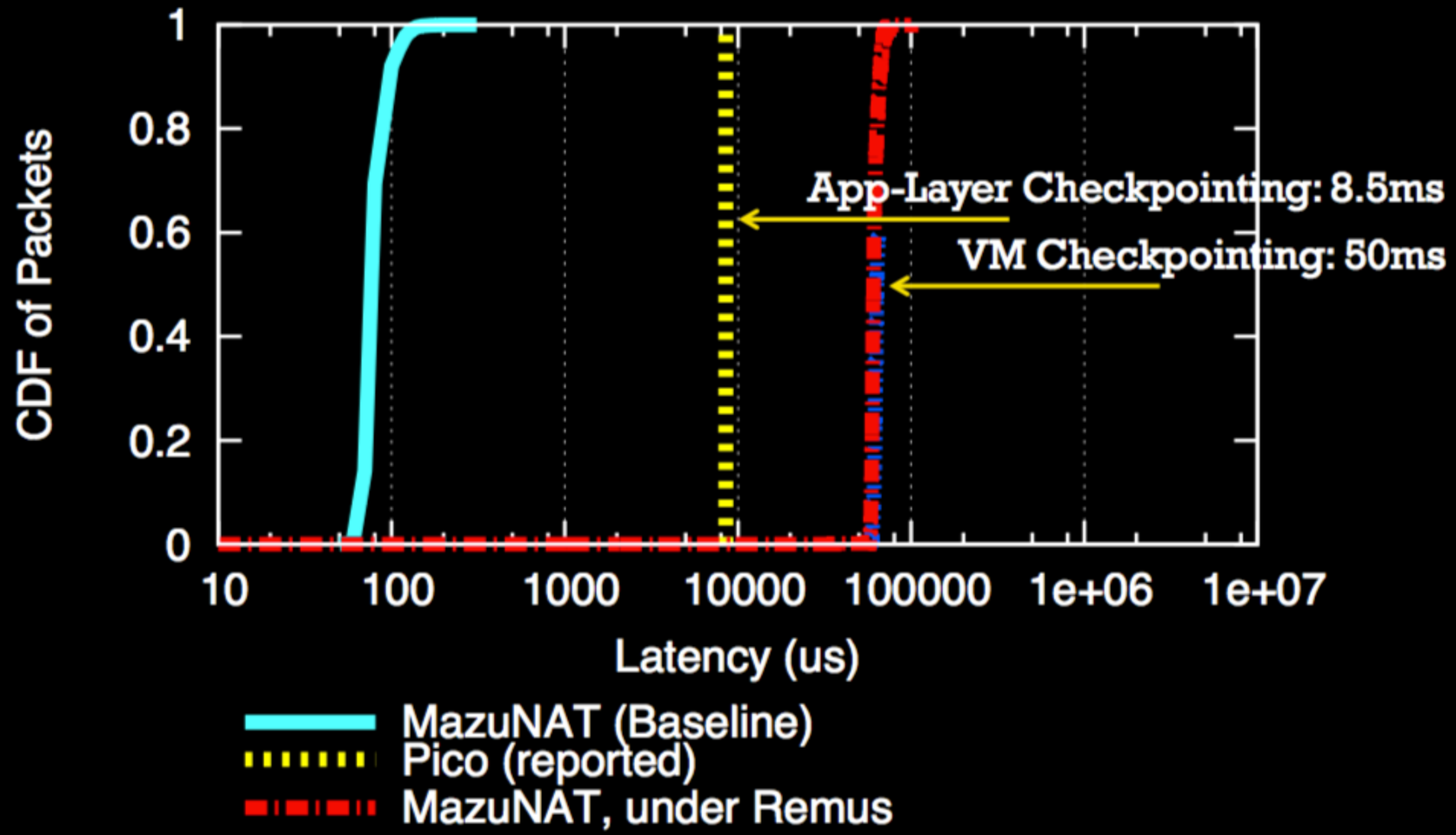
Performance Highlights

Latency	Throughput	Recovery Time
Remus [NSDI 2008]: 50,000us overhead	None higher than 200kpps	100s of ms
Colo [SOCC 2013]: 50,000us overhead	FTMB: **1.4-4Mpps**	FTMB: **increases recovery time by 50-300ms.**
Pico [SOCC 2013]: 8000us overhead	**5-30% reduction over baseline throughput**	**Still fast enough not to trigger TCP timeouts or errors!**
FTMB: **30us** overhead		

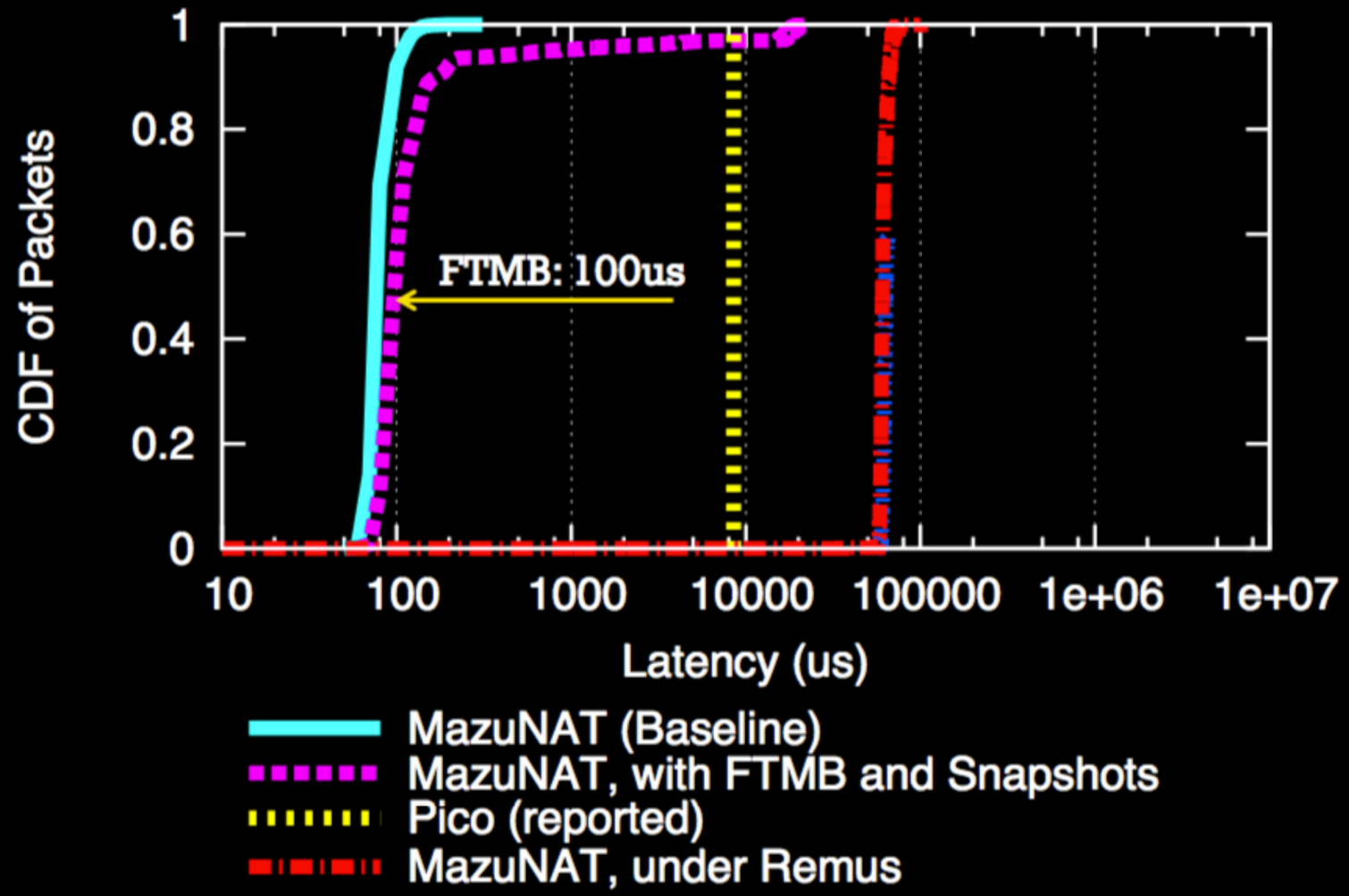
Latency under HA Solutions



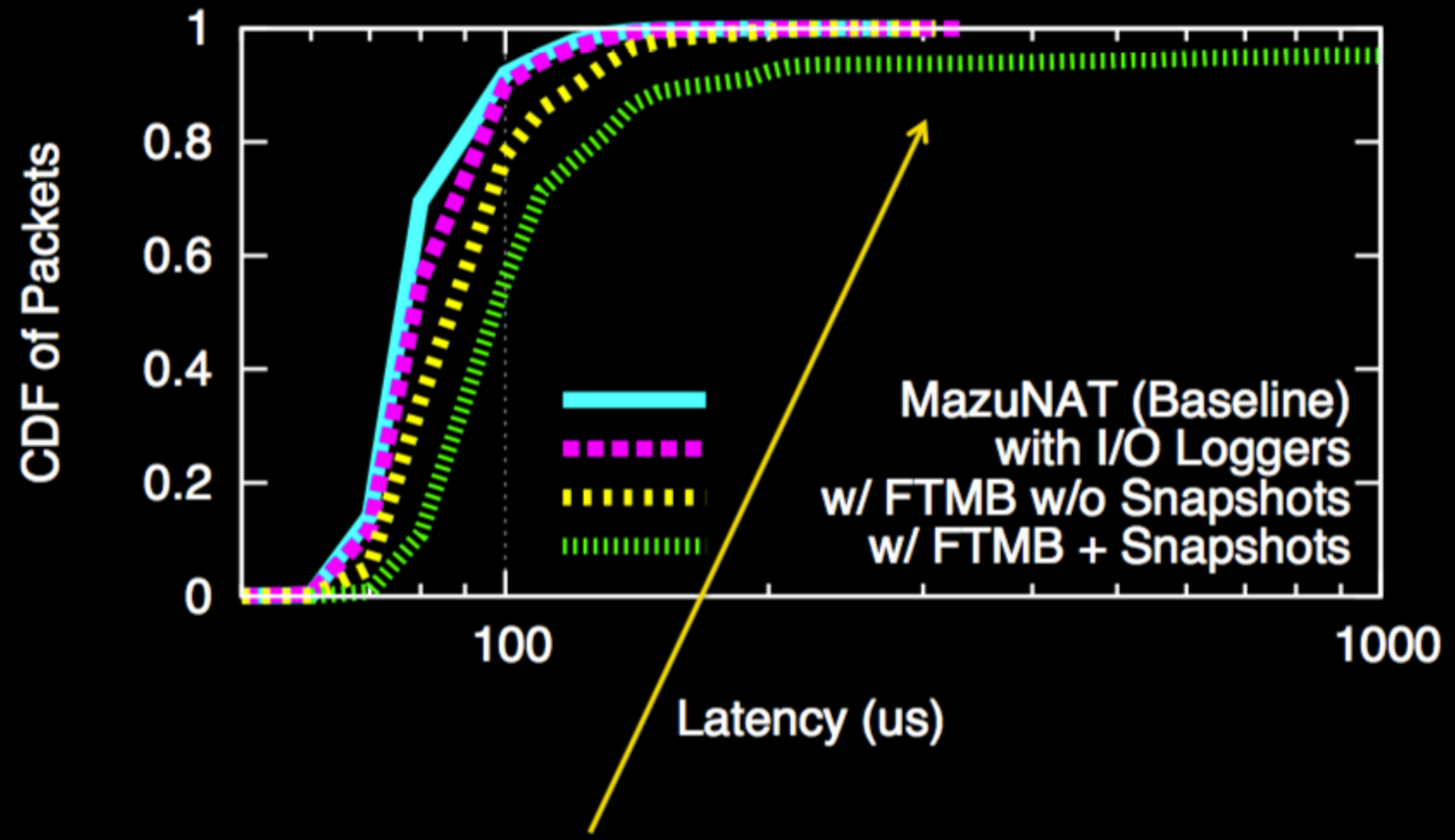
Latency under HA Solutions



Latency under HA Solutions

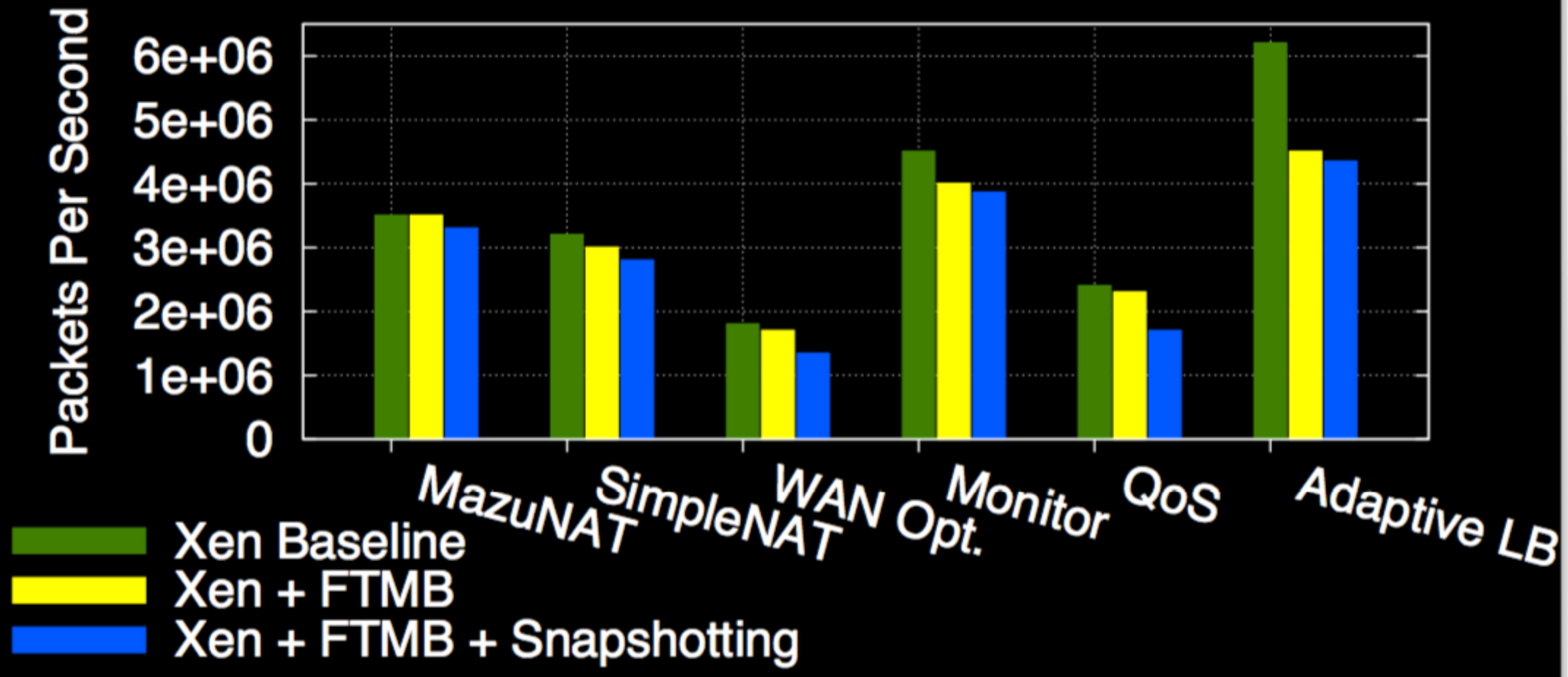


Latency



Tail comes from periodic VM suspension for snapshots

Throughput



5%-30% overhead, depending on the application.

Thank you!

FTMB: Correct Recovery and Performance

Obeys output commit using **ordered logging and parallel release.**

30us latency overhead

5-30% throughput reduction