IISc CNI Seminar, March 11 2025

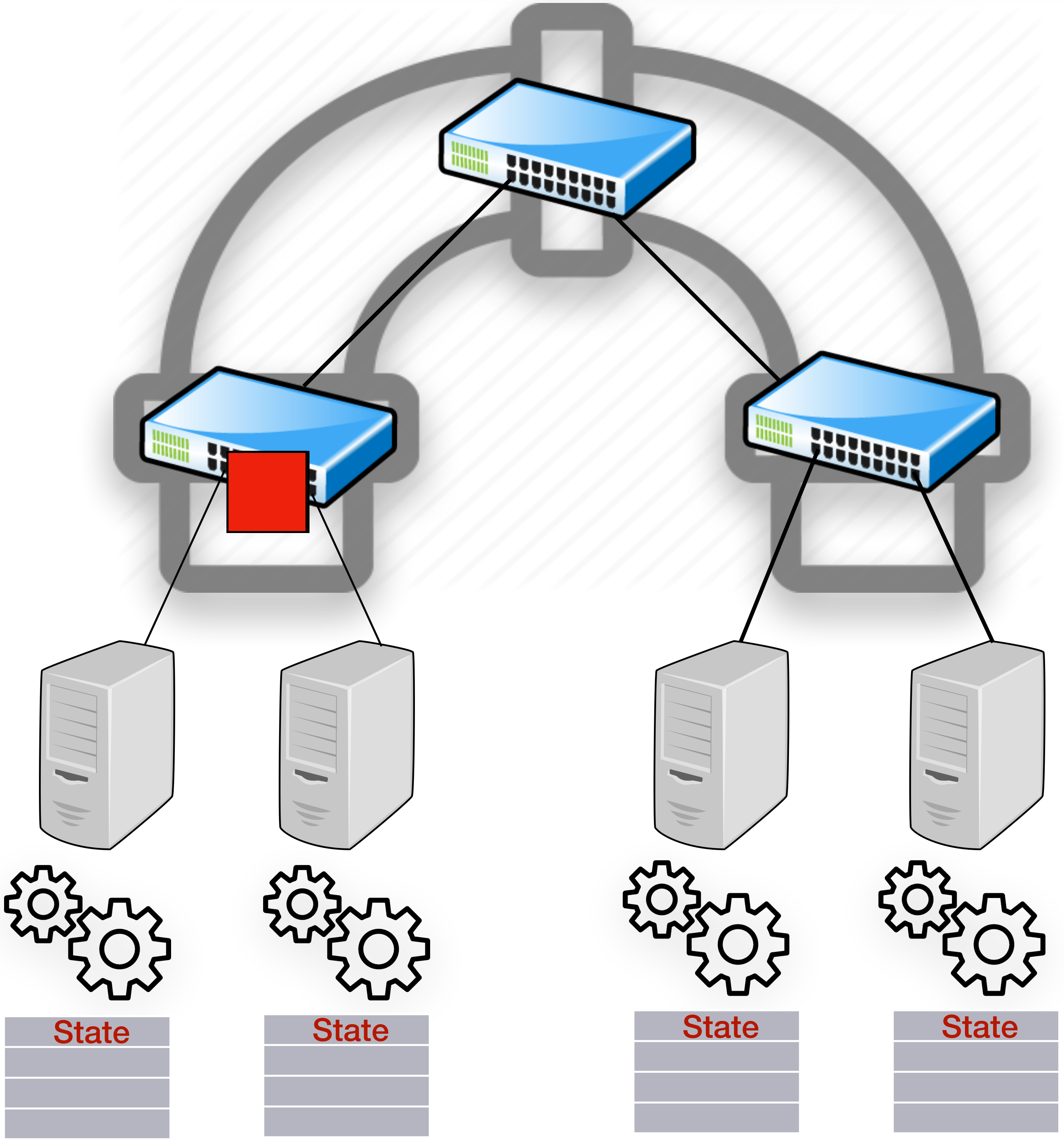
Enabling and Exploiting High-Speed In-Network Computing

Vishal Shrivastav

PURDUE UNIVERSITY®

Traditional Distributed Computing

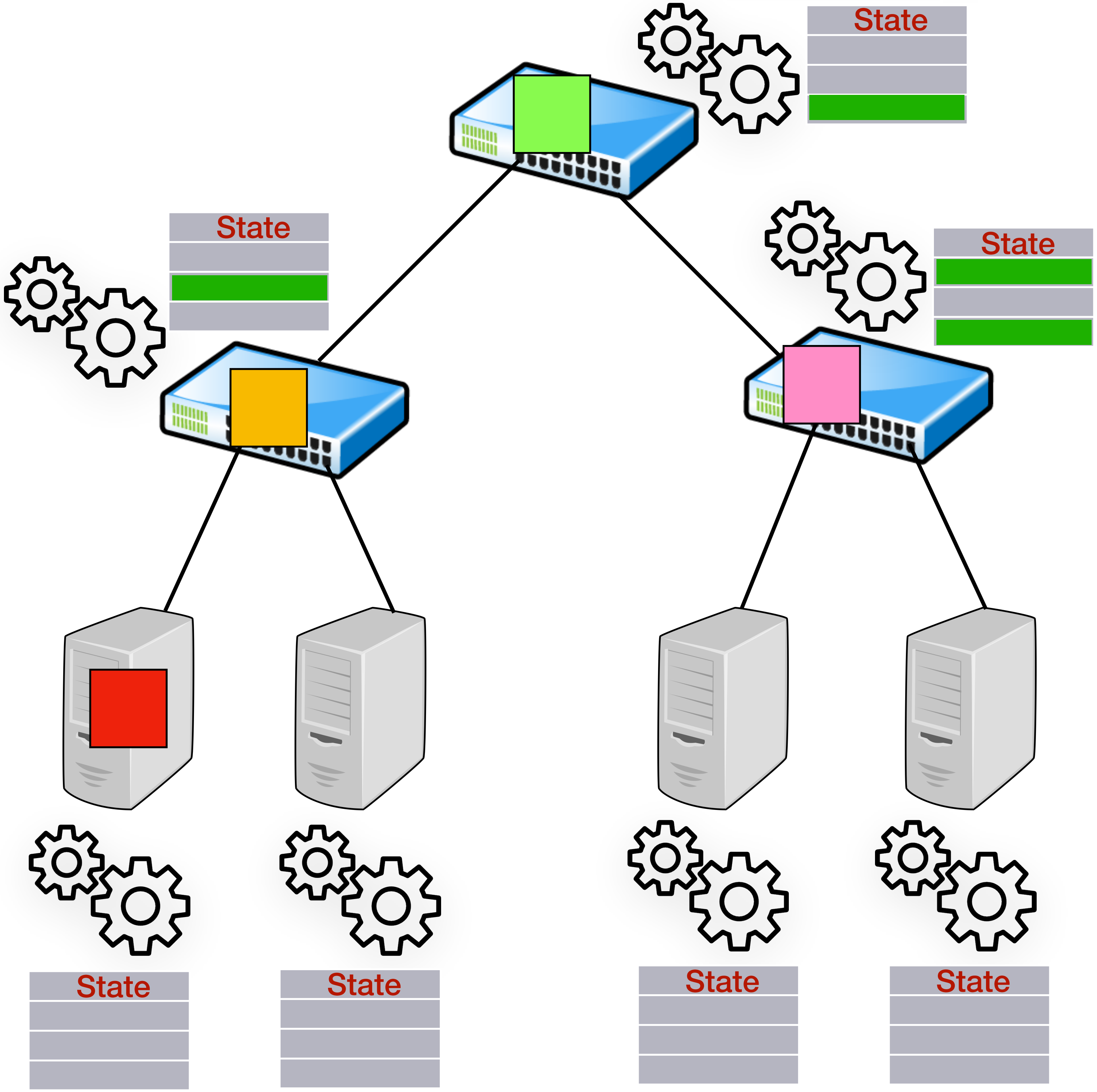
Network is a *passive* partner in communication



- Computation is distributed across the end-hosts
- End-hosts exchange messages to share state
- Network is abstracted as a “pipe”, responsible only for carrying messages between end-hosts

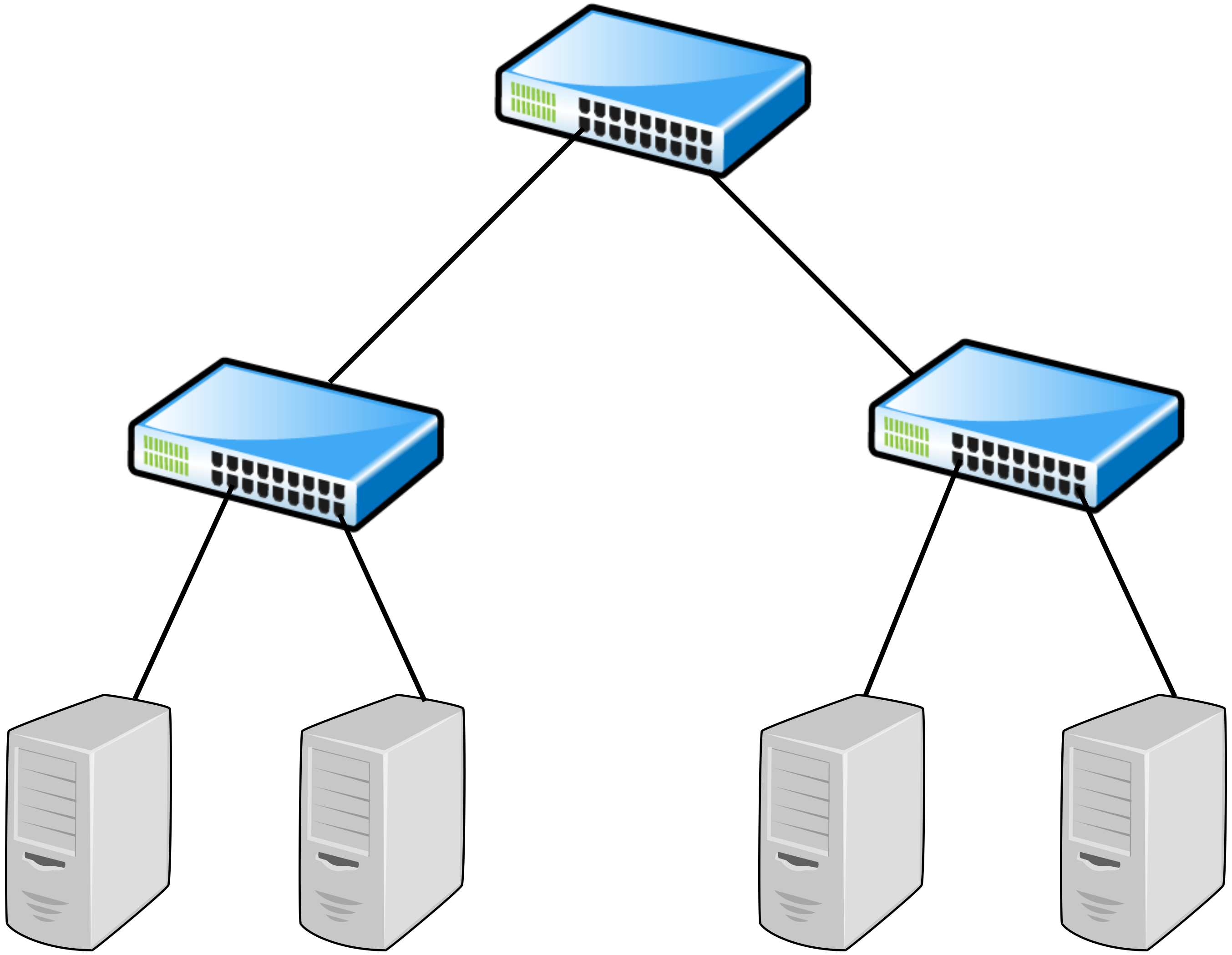
In-Network Computing

Make network an *active* partner in communication



- Custom computation offloaded to network devices (e.g., switches)
- Switches can maintain state
- Custom computation can be performed on both switch state and packet fields
- Both switch state and packet fields can be updated based on the custom computation

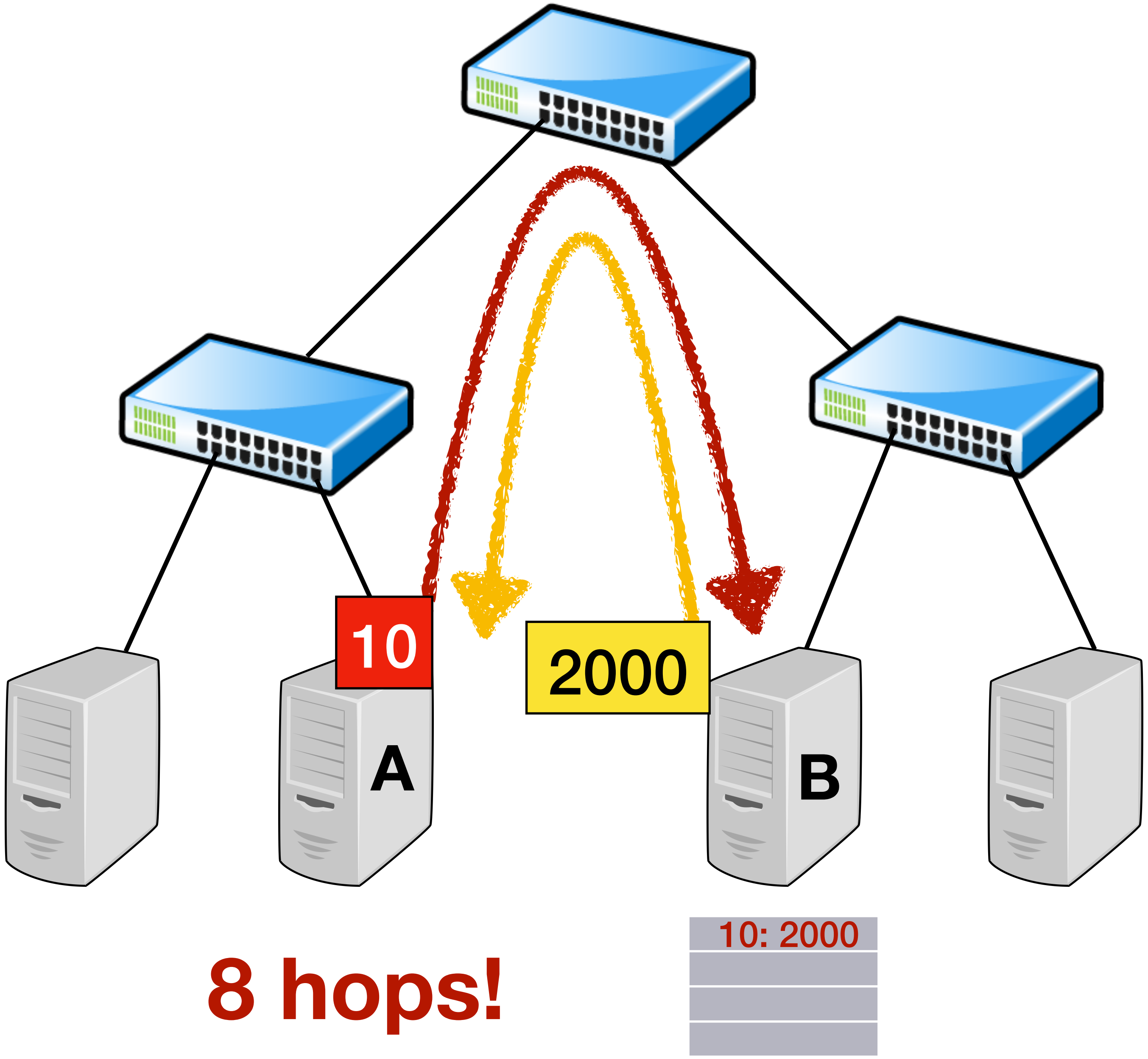
Rationale for In-Network Computing



- Switches sit at a “central” vantage point in the network
  - Well suited to implement partition / aggregate computations, caching, etc.

Rationale for In-Network Computing

Distributed Key Value store

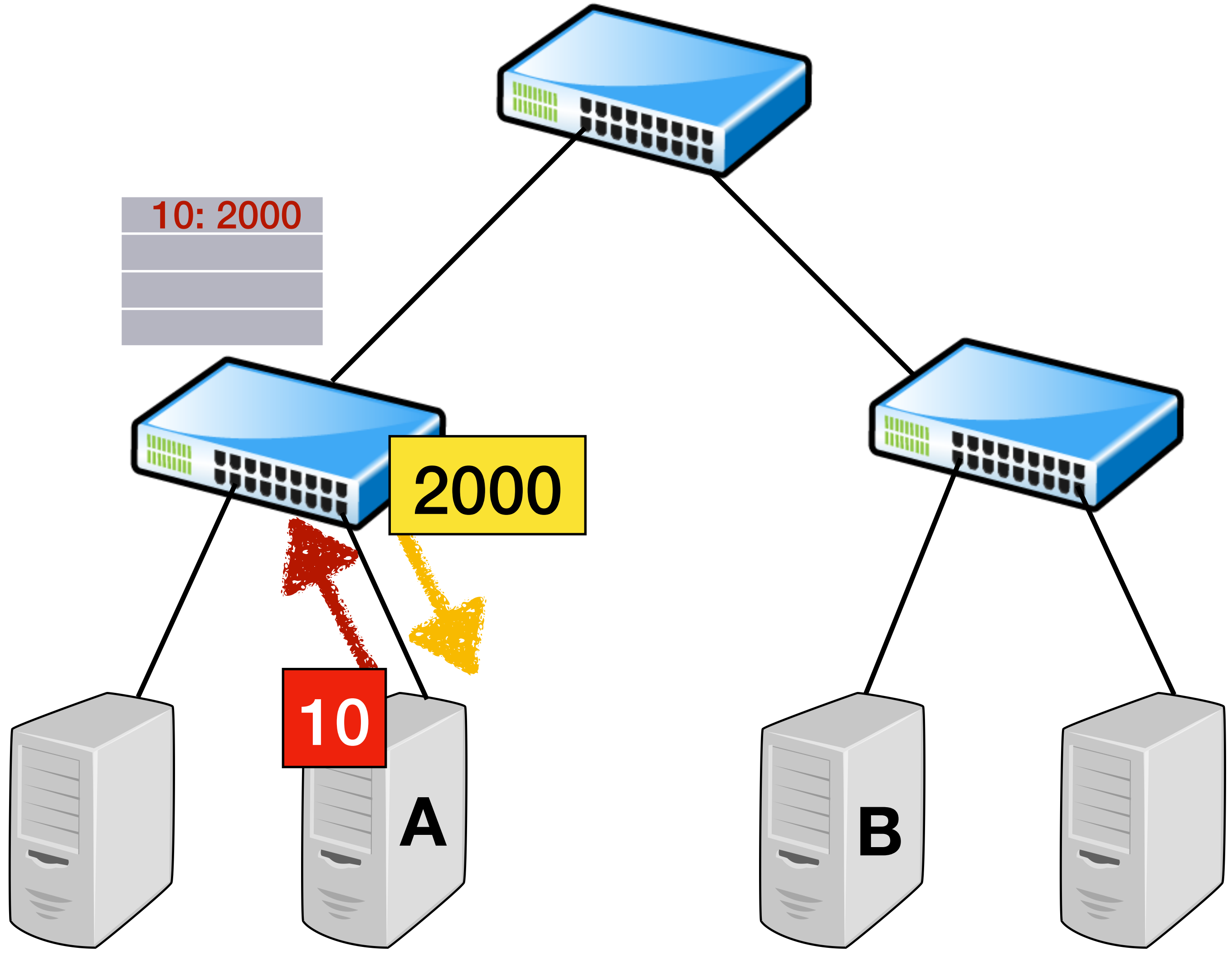


8 hops!

- Switches sit at a "central" vantage point in the network
  - Well suited to implement partition / aggregate computations, caching, etc.
    - Reduce network trip length

Rationale for In-Network Computing

Distributed Key Value store

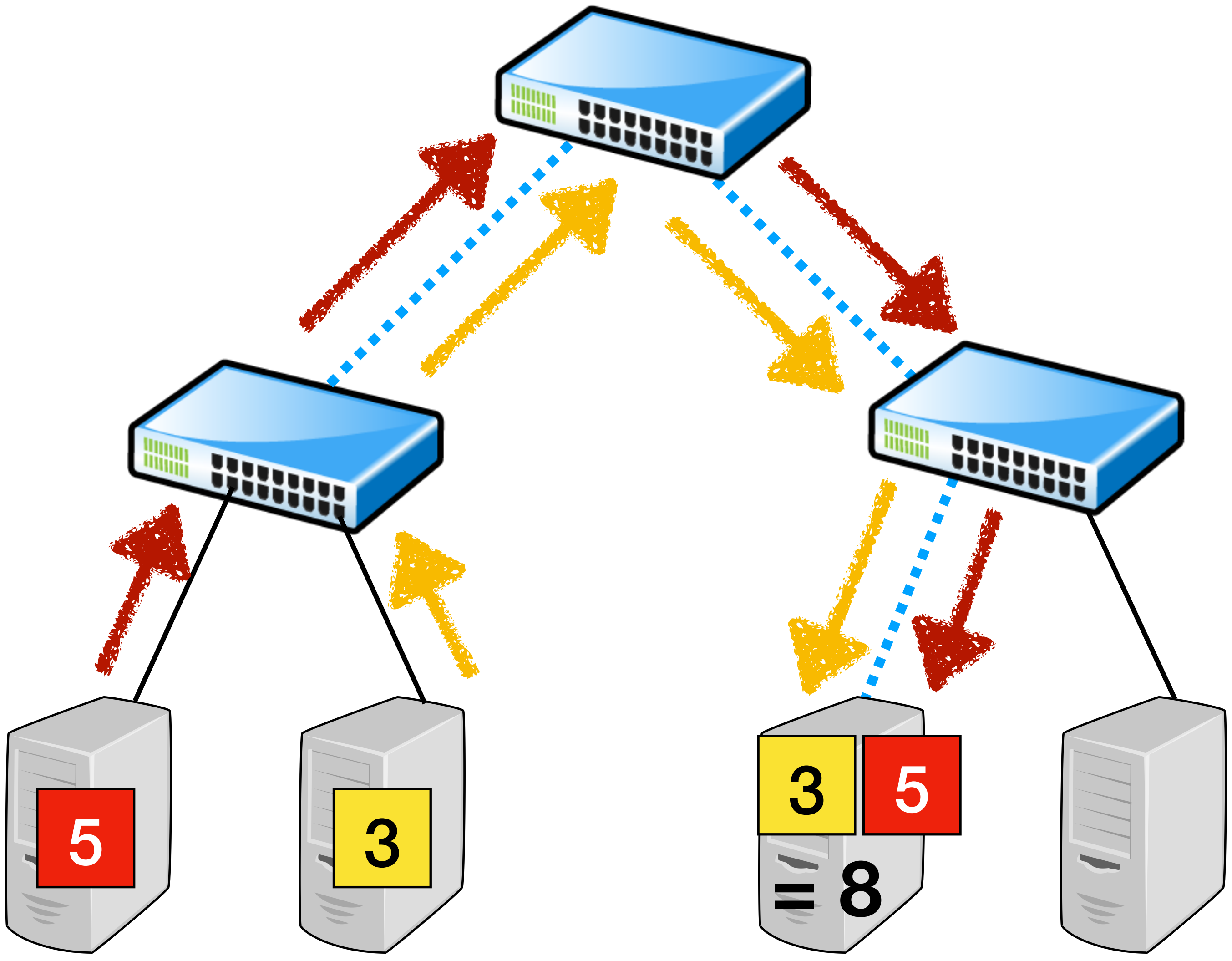


2 hops w/ In-network caching!

- Switches sit at a “central” vantage point in the network
  - Well suited to implement partition / aggregate computations, caching, etc.
    - Reduce network trip length

Rationale for In-Network Computing

Gradient aggregation in ML training

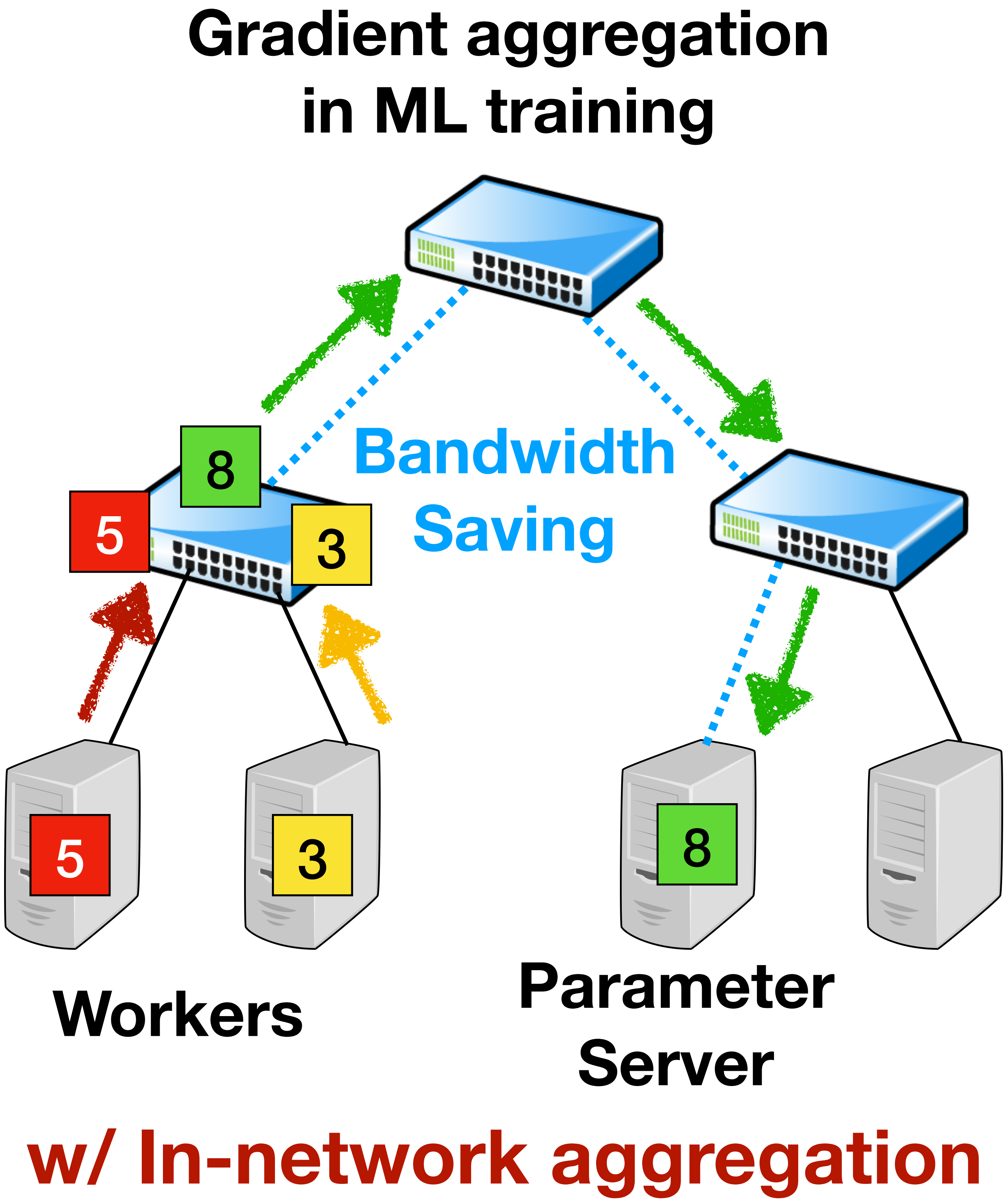


Workers

Parameter Server

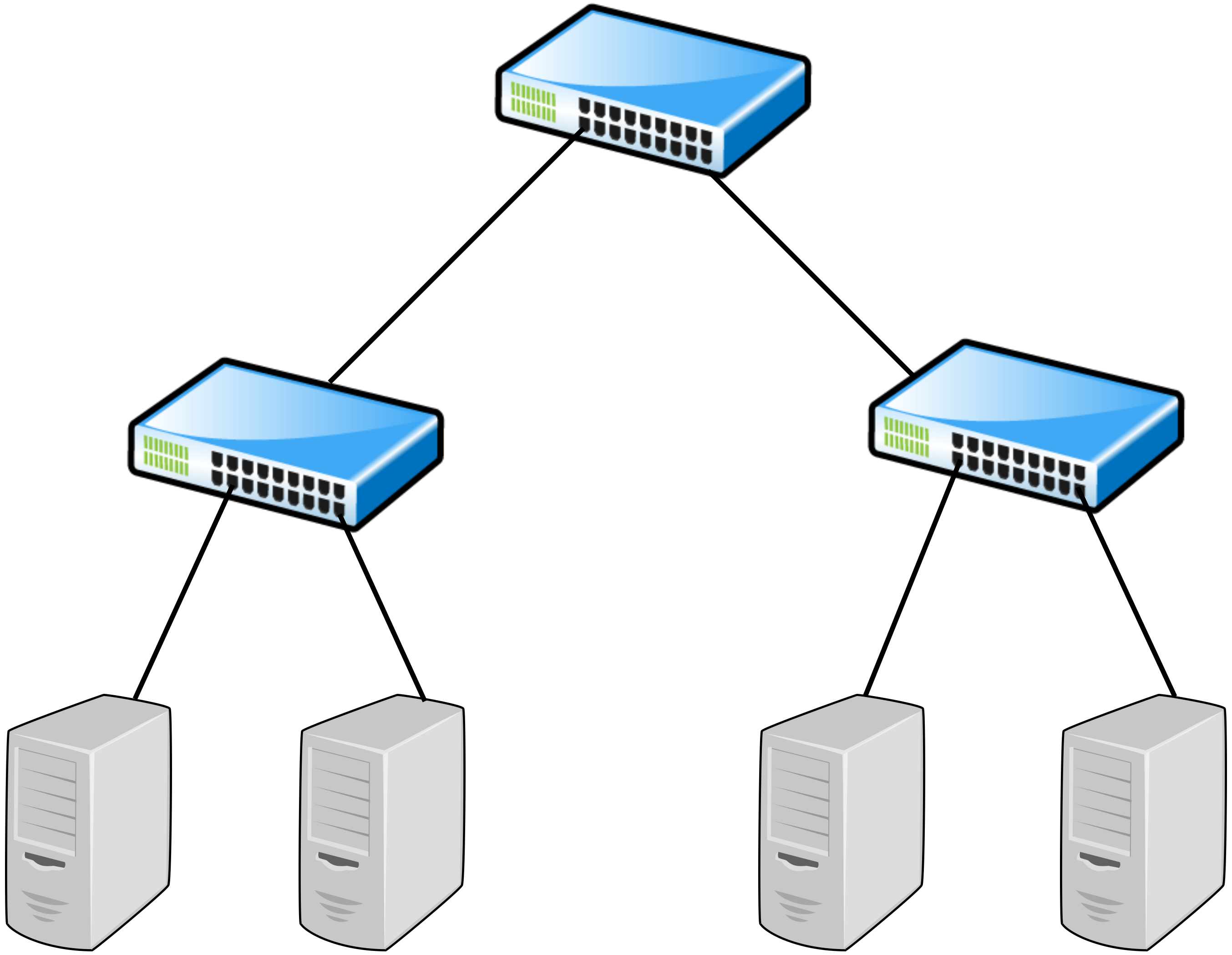
- Switches sit at a “central” vantage point in the network
  - Well suited to implement partition / aggregate computations, caching, etc.
    - Reduce network trip length
    - Save network bandwidth

Rationale for In-Network Computing



- Switches sit at a "central" vantage point in the network
  - Well suited to implement partition / aggregate computations, caching, etc.
    - Reduce network trip length
    - Save network bandwidth

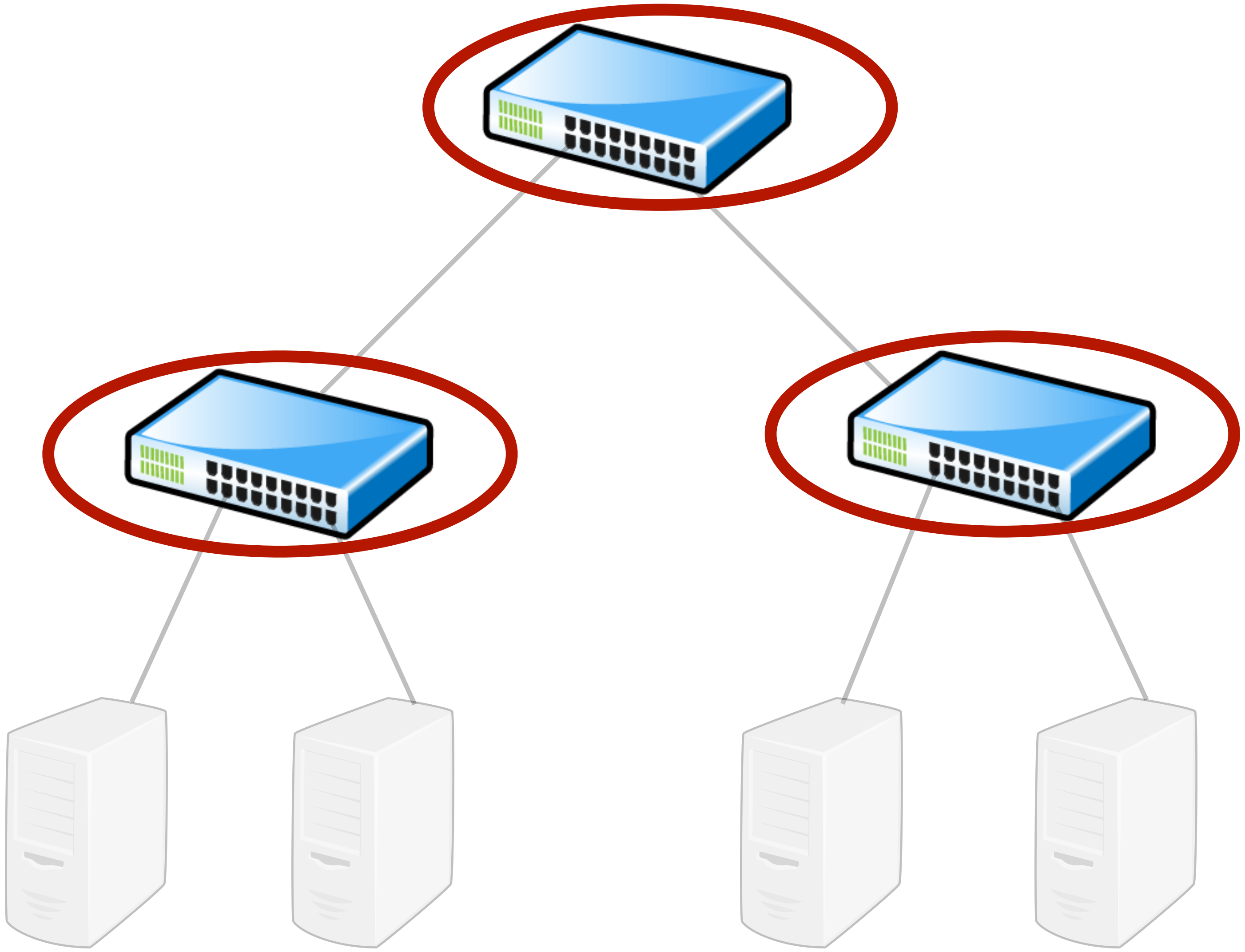
Rationale for In-Network Computing



- Switches sit at a “central” vantage point in the network
  - Well suited to implement partition / aggregate computations, caching, etc.
    - Reduce network trip length
    - Save network bandwidth
- Switches by design run at line rate (multi-Tbps)
  - Orders of magnitude higher throughput, lower latency and power efficiency compared to CPU-based servers

Enabling Technology

Programmable Switches



Intel Tofino

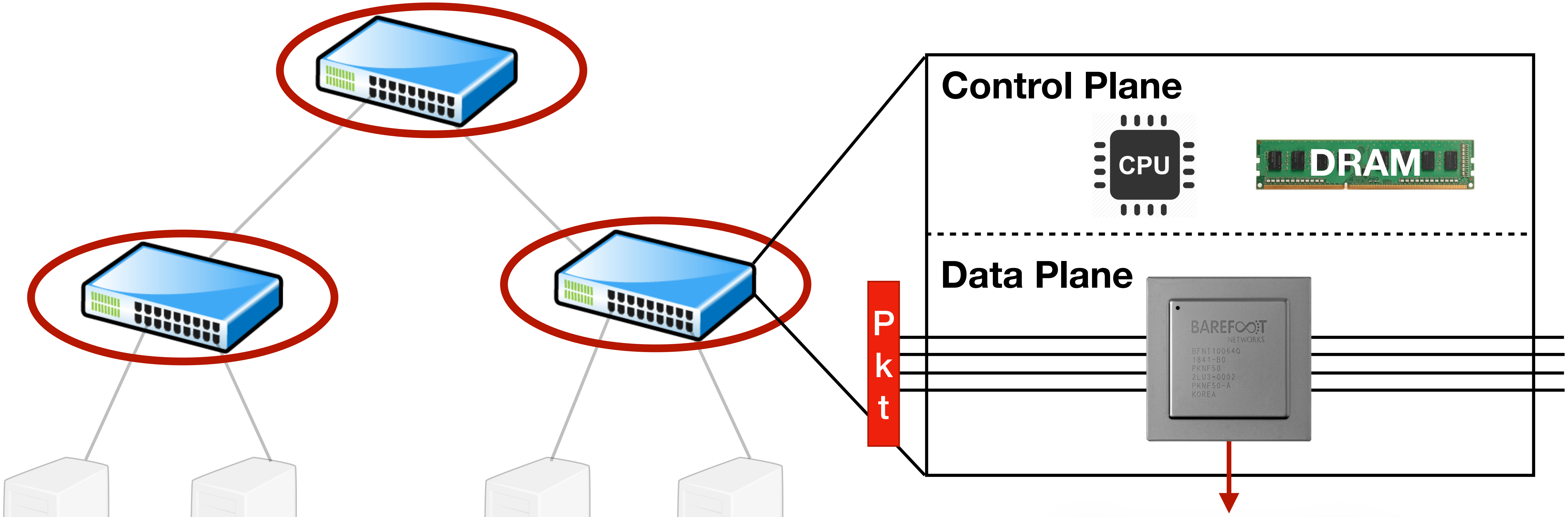
Cisco Nexus

Broadcom Trident

Mellanox Spectrum

Enabling Technology

Programmable Switches



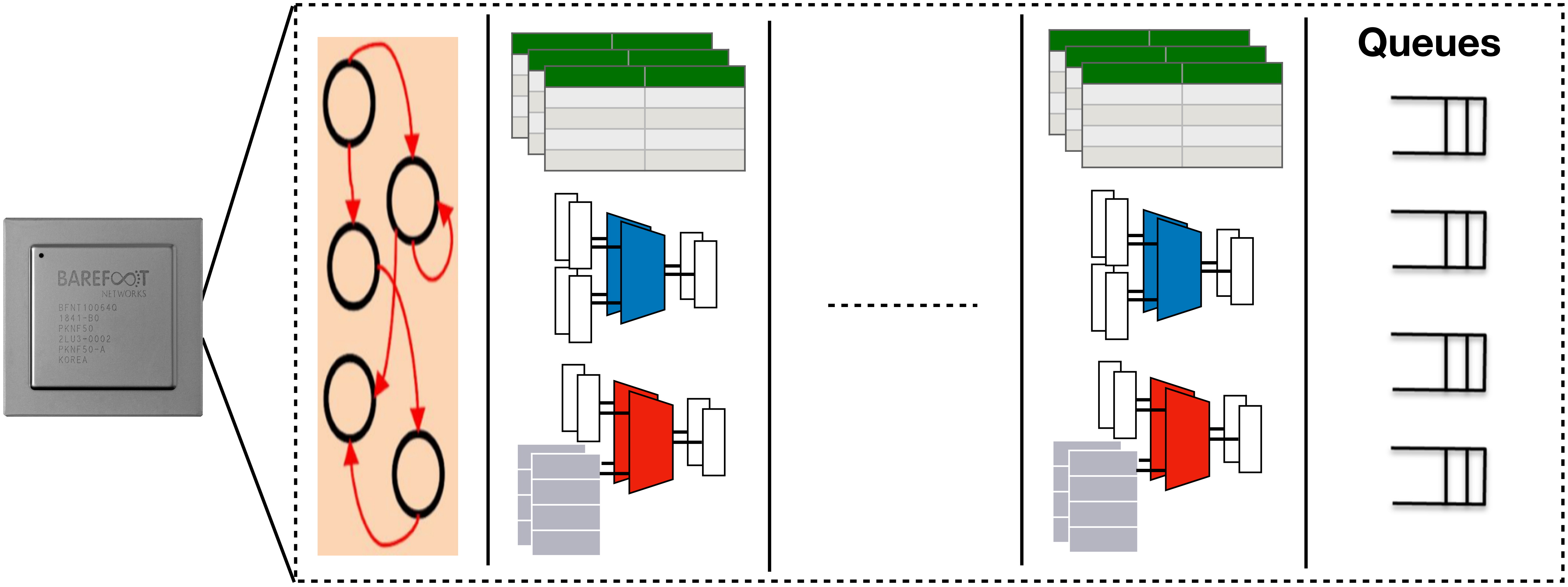
Programmable Switching Chip

Gen 1: 6.4 Tbps
Gen 2: 12.9 Tbps
Gen 3: 25.6 Tbps

Domain-specific Programming language P4

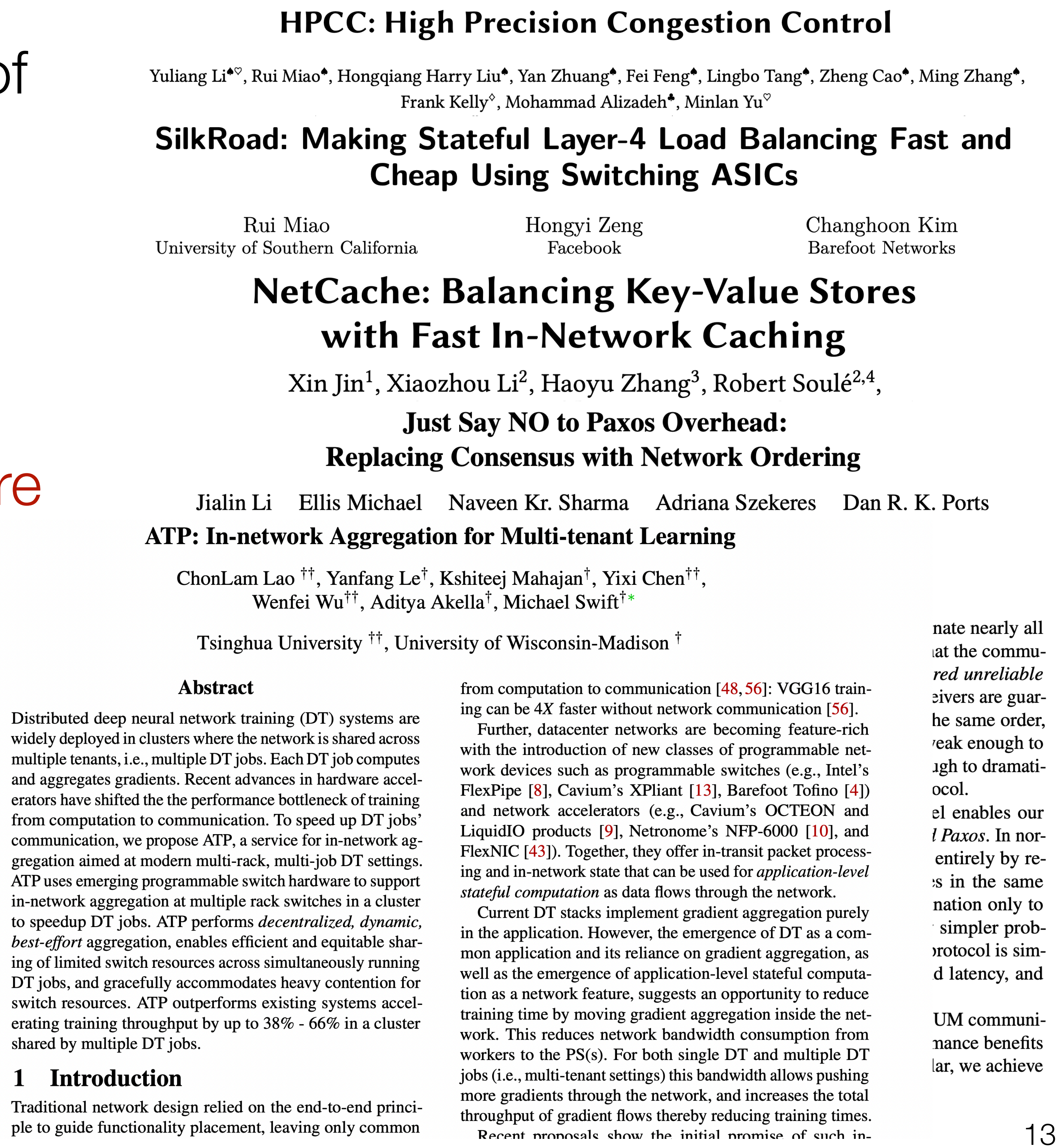
Programmable Switch Architecture

- **<u>R</u>econfigurable <u>M</u>atch <u>T</u>able (RMT) Architecture:**
  - Programmable parser (◌): Parse custom header fields
  - Match-Action Tables (▦) SRAM or TCAM: Match on custom header fields
  - Switch Registers (▤) SRAM: Store custom state
  - Stateless ALUs (🟦): Custom operations only on header fields
  - Stateful ALUs (🟥): Custom operations on header fields and switch registers



Power of Programmable Switch

- Improves the performance of several key distributed and network applications
    - Congestion control
    - Load balancing
    - Distributed Key Value store
    - Consensus protocol
    - ML training
    - …..



HPCC: High Precision Congestion Control

Yuliang Li♠♡, Rui Miao♠, Hongqiang Harry Liu♠, Yan Zhuang♠, Fei Feng♠, Lingbo Tang♠, Zheng Cao♠, Ming Zhang♠, Frank Kelly◊, Mohammad Alizadeh♣, Minlan Yu♡

SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs

Rui Miao
University of Southern California

Hongyi Zeng
Facebook

Changhoon Kim
Barefoot Networks

NetCache: Balancing Key-Value Stores with Fast In-Network Caching

Xin Jin[1], Xiaozhou Li[2], Haoyu Zhang[3], Robert Soulé[2,4],

Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering

Jialin Li Ellis Michael Naveen Kr. Sharma Adriana Szekeres Dan R. K. Ports

ATP: In-network Aggregation for Multi-tenant Learning

ChonLam Lao ††, Yanfang Le†, Kshiteej Mahajan†, Yixi Chen††,
Wenfei Wu††, Aditya Akella†, Michael Swift†*

Tsinghua University ††, University of Wisconsin-Madison †

Abstract

Distributed deep neural network training (DT) systems are widely deployed in clusters where the network is shared across multiple tenants, i.e., multiple DT jobs. Each DT job computes and aggregates gradients. Recent advances in hardware accelerators have shifted the the performance bottleneck of training from computation to communication. To speed up DT jobs' communication, we propose ATP, a service for in-network aggregation aimed at modern multi-rack, multi-job DT settings. ATP uses emerging programmable switch hardware to support in-network aggregation at multiple rack switches in a cluster to speedup DT jobs. ATP performs *decentralized, dynamic, best-effort* aggregation, enables efficient and equitable sharing of limited switch resources across simultaneously running DT jobs, and gracefully accommodates heavy contention for switch resources. ATP outperforms existing systems accelerating training throughput by up to 38% - 66% in a cluster shared by multiple DT jobs.

1 Introduction

Traditional network design relied on the end-to-end principle to guide functionality placement, leaving only common from computation to communication [48, 56]: VGG16 training can be 4X faster without network communication [56].

Further, datacenter networks are becoming feature-rich with the introduction of new classes of programmable network devices such as programmable switches (e.g., Intel's FlexPipe [8], Cavium's XPliant [13], Barefoot Tofino [4]) and network accelerators (e.g., Cavium's OCTEON and LiquidIO products [9], Netronome's NFP-6000 [10], and FlexNIC [43]). Together, they offer in-transit packet processing and in-network state that can be used for *application-level stateful computation* as data flows through the network.

Current DT stacks implement gradient aggregation purely in the application. However, the emergence of DT as a common application and its reliance on gradient aggregation, as well as the emergence of application-level stateful computation as a network feature, suggests an opportunity to reduce training time by moving gradient aggregation inside the network. This reduces network bandwidth consumption from workers to the PS(s). For both single DT and multiple DT jobs (i.e., multi-tenant settings) this bandwidth allows pushing more gradients through the network, and increases the total throughput of gradient flows thereby reducing training times.

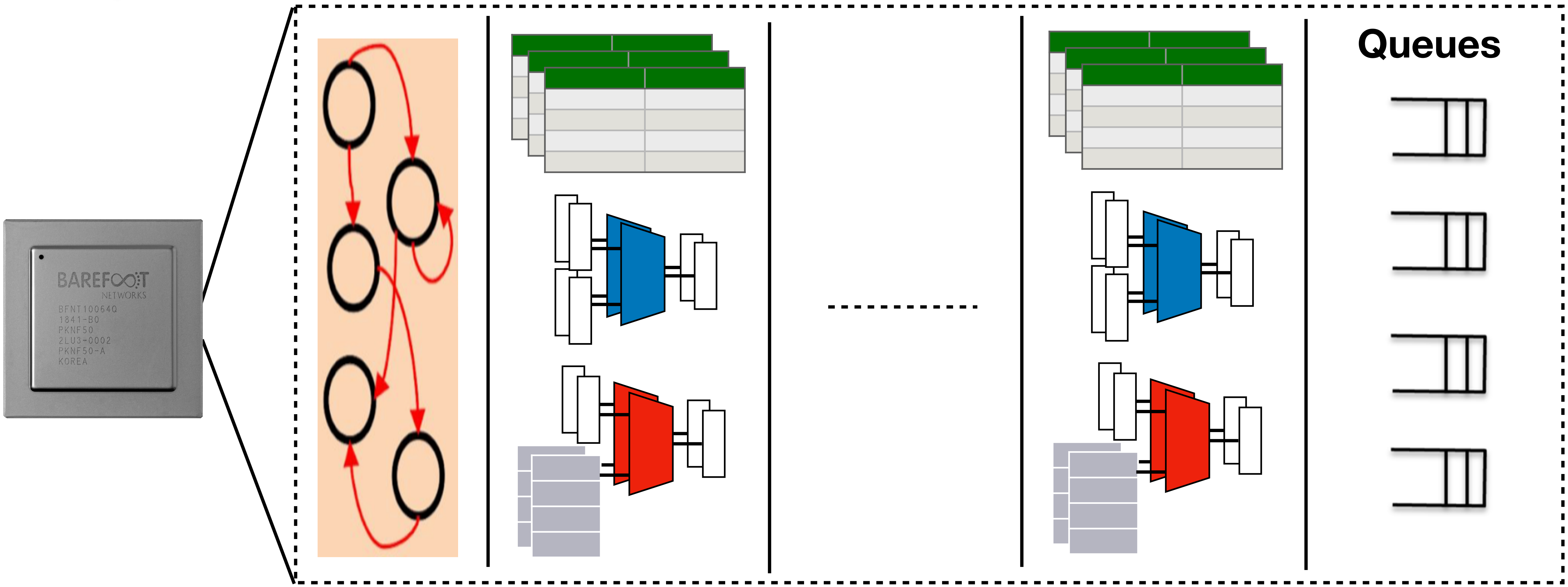
Recent proposals show the initial promise of such in-

nate nearly all
ıat the commu-
red unreliable
eivers are guar-
he same order,
/eak enough to
ıgh to dramati-
ocol.
el enables our
l Paxos. In nor-
entirely by re-
s in the same
nation only to
simpler prob-
rotocol is sim-
d latency, and

UM communi-
nance benefits
lar, we achieve

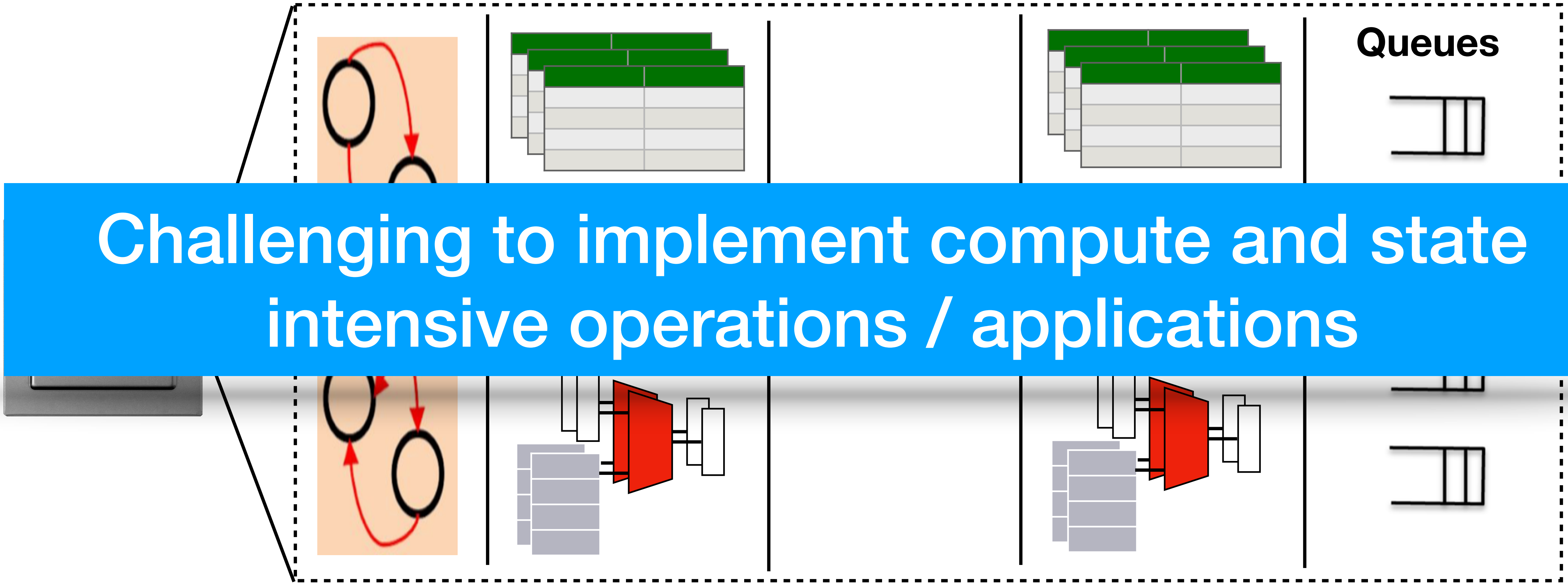
Limitations of Programmable Switch

- **RMT is a “domain-specific” architecture — Not general purpose**
  - *Sacrifices generality for line rate (multi-Tbps) processing*
  - *Also has power, chip area, and cost constraints …*
    - ✗ Limited # of switch registers (few MB for the entire switch)
    - ✗ Limited # of Match-Action stages (~10-20) & stateful ALUs / stage (~4-5)
    - ✗ Can only support simple ALU operations (add, bit shift, comparisons)



Limitations of Programmable Switch

- **RMT is a “domain-specific” architecture — Not general purpose**
  - *Sacrifices generality for line rate (multi-Tbps) processing*
  - *Also has power, chip area, and cost constraints …*
    - ✗ Limited # of switch registers (few MB for the entire switch)
    - ✗ Limited # of Match-Action stages (~10-20) & stateful ALUs / stage (~4-5)
    - ✗ Can only support simple ALU operations (add, bit shift, comparisons)



Challenging to implement compute and state intensive operations / applications

An Illustrative Example

- **Network Intrusion Detection System (NIDS)**
  - *Classifies network traffic as malicious or benign*

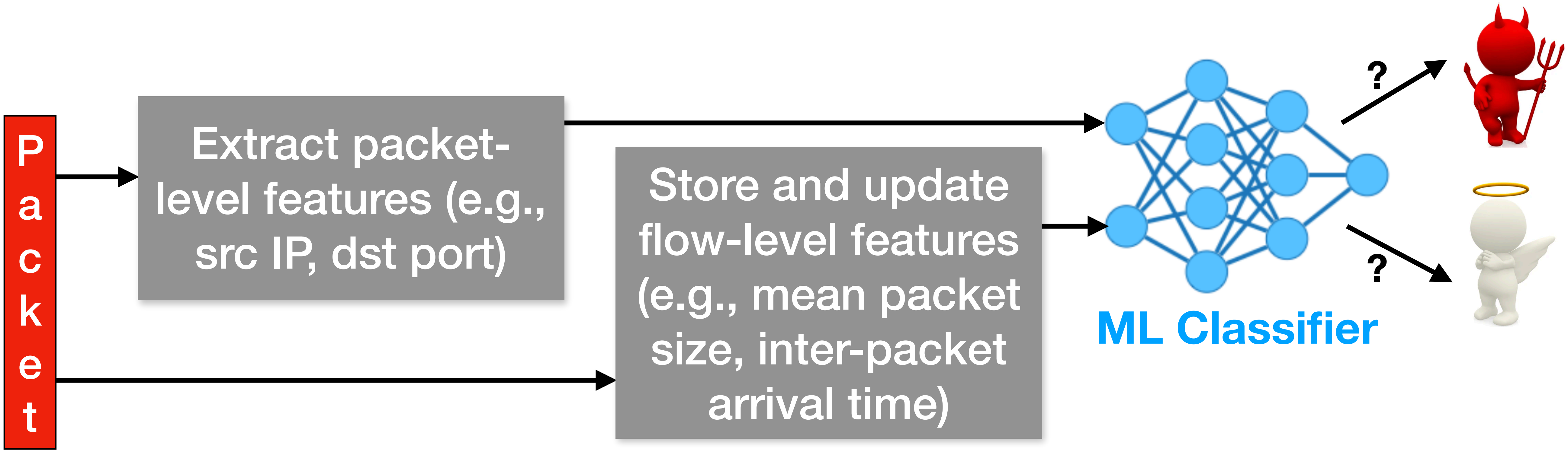
- **Traditional NIDS:** Rule-based (e.g., Snort)
  - E.g., if `dst port` = 80 and `payload` contains `a*b#` : `Malicious`

  ✗ Cannot detect complex malicious patterns in traffic

  ✗ Payload inspection not possible in encrypted traffic

- **Recent NIDS:** ML-based
  - Extracts packet- and flow-level features from network traffic
    - Does not rely on payload, only inspects the header
  - Feeds the features to an ML classifier to detect malicious traffic

Workflow of ML-based NIDS



Packet

Extract packet-level features (e.g., src IP, dst port)

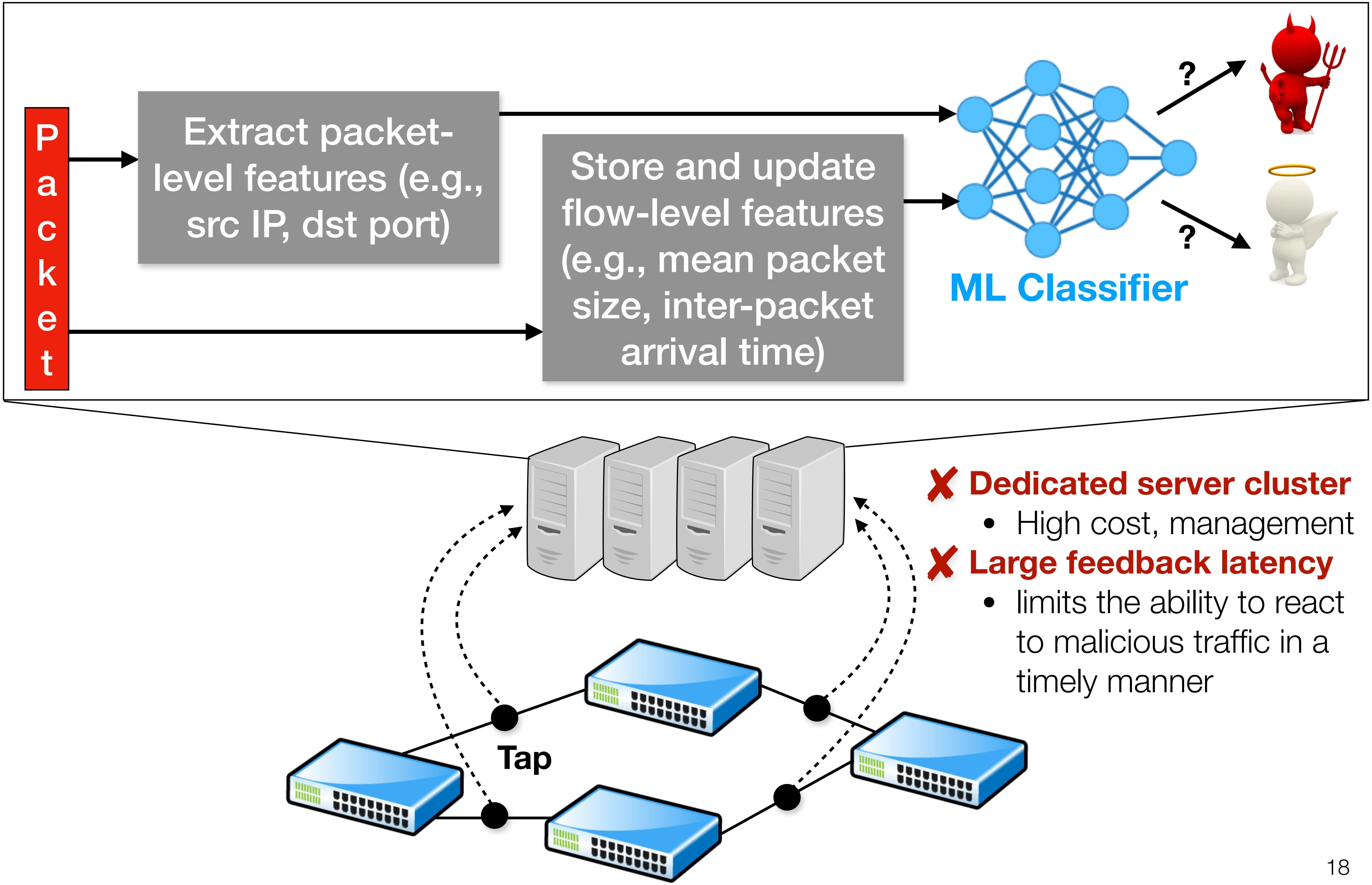
Store and update flow-level features (e.g., mean packet size, inter-packet arrival time)

ML Classifier

?

?

Existing ML-based NIDS



Packet

Extract packet-level features (e.g., src IP, dst port)

Store and update flow-level features (e.g., mean packet size, inter-packet arrival time)

ML Classifier

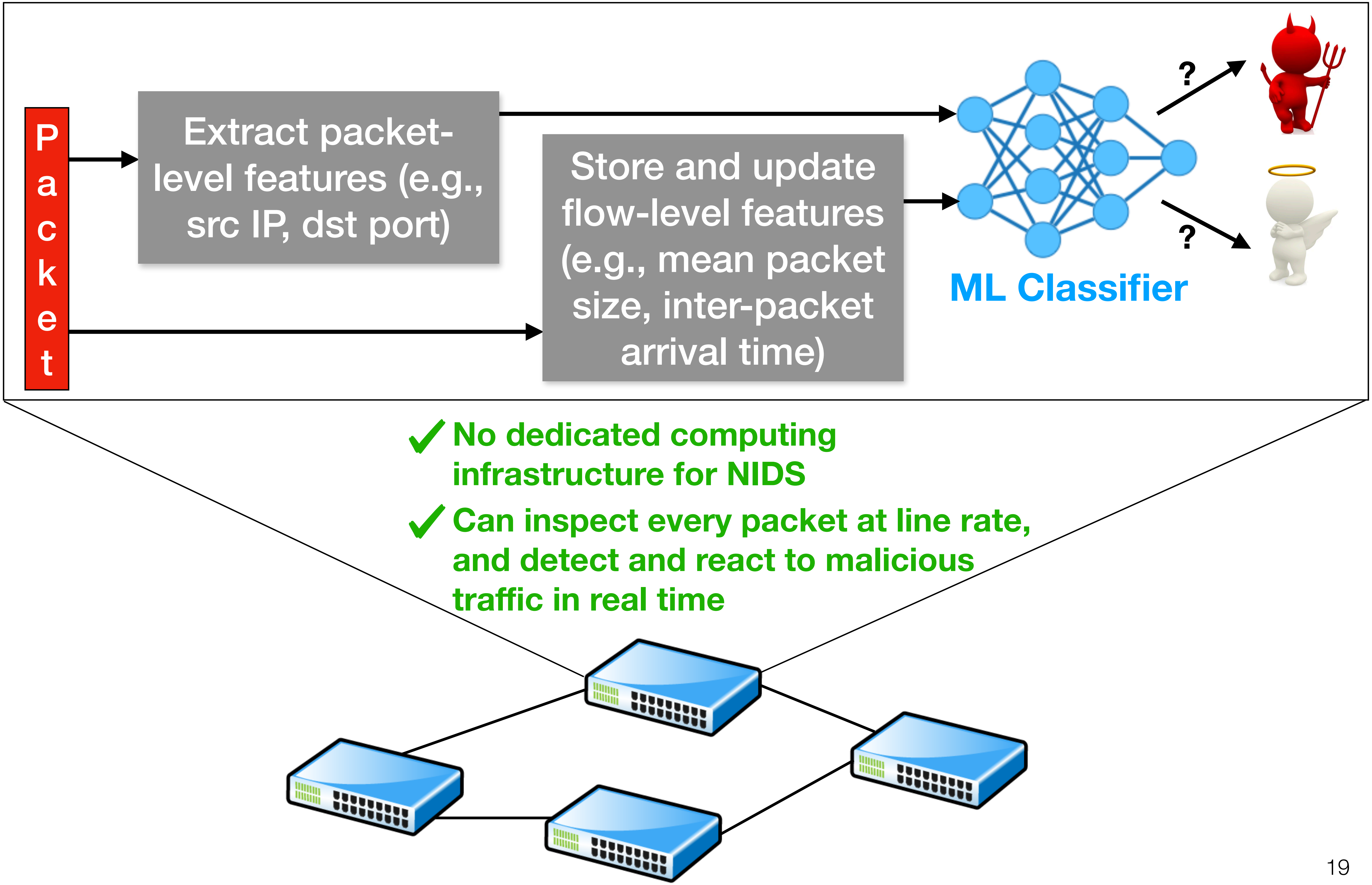
✗ **Dedicated server cluster**

- High cost, management

✗ **Large feedback latency**

- limits the ability to react to malicious traffic in a timely manner

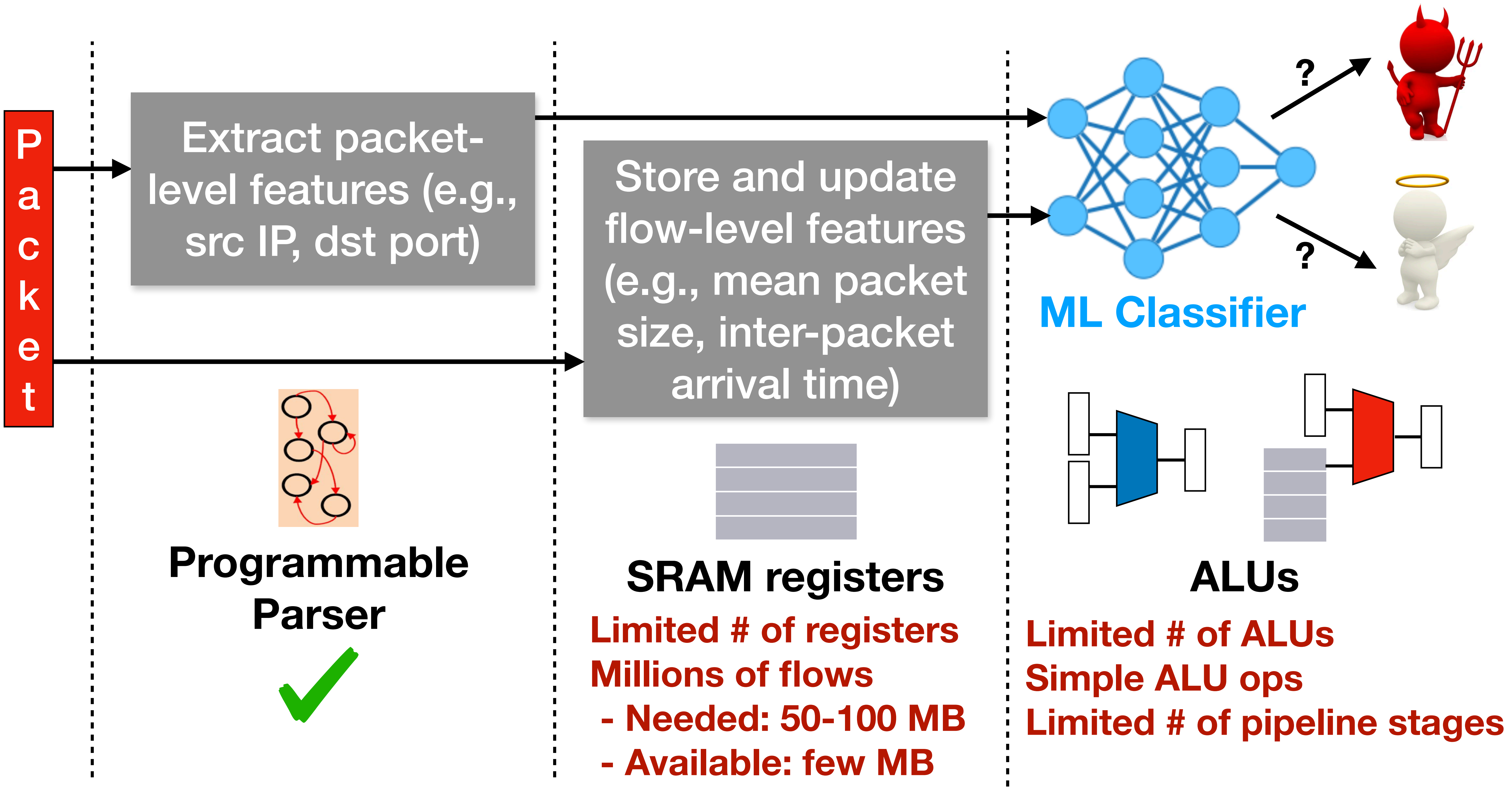
In-Network ML-based NIDS


Packet
Extract packet-level features (e.g., src IP, dst port)
Store and update flow-level features (e.g., mean packet size, inter-packet arrival time)
ML Classifier
?
?

✓ No dedicated computing infrastructure for NIDS

✓ Can inspect every packet at line rate, and detect and react to malicious traffic in real time

ML-based NIDS on Programmable Switch



Packet

Extract packet-level features (e.g., src IP, dst port)

Store and update flow-level features (e.g., mean packet size, inter-packet arrival time)

ML Classifier

Programmable Parser

✓

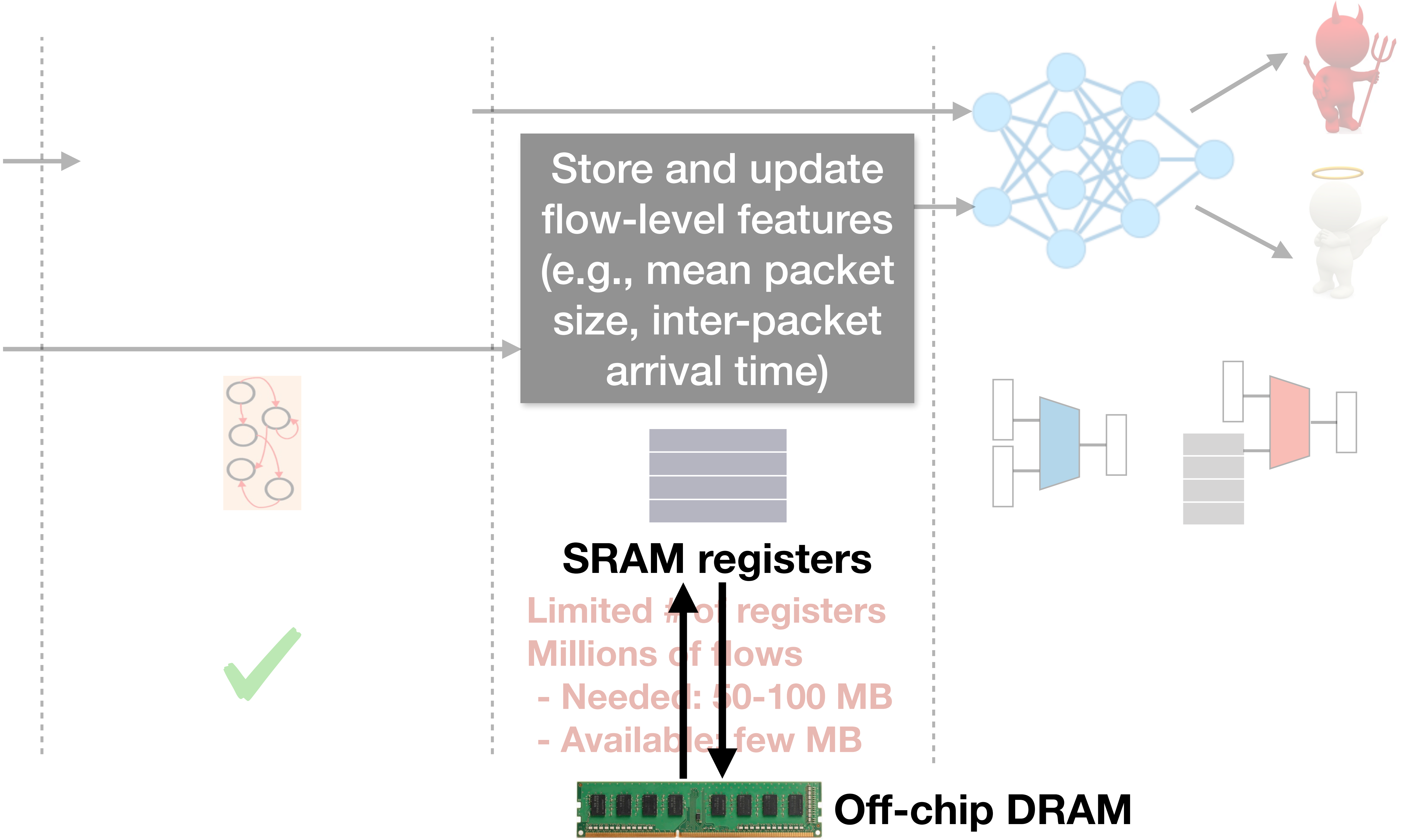
SRAM registers

Limited # of registers
Millions of flows
- Needed: 50-100 MB
- Available: few MB

ALUs

Limited # of ALUs
Simple ALU ops
Limited # of pipeline stages

Challenge (1): State-Intensive



Store and update flow-level features (e.g., mean packet size, inter-packet arrival time)

SRAM registers

Limited # of registers

Millions of flows

- Needed: 50-100 MB
- Available: few MB

Off-chip DRAM

✓

Design efficient caching for state-intensive network applications

Limitation of existing caching solutions

- Several online caching heuristics used in practice:
  - LRU, LFU, ARC, CLOCK, S3-FIFO, SIEVE, etc.
- … but all fall short of the optimal offline caching algorithm **(Belady)**

Fundamental Cause of Performance Gap:

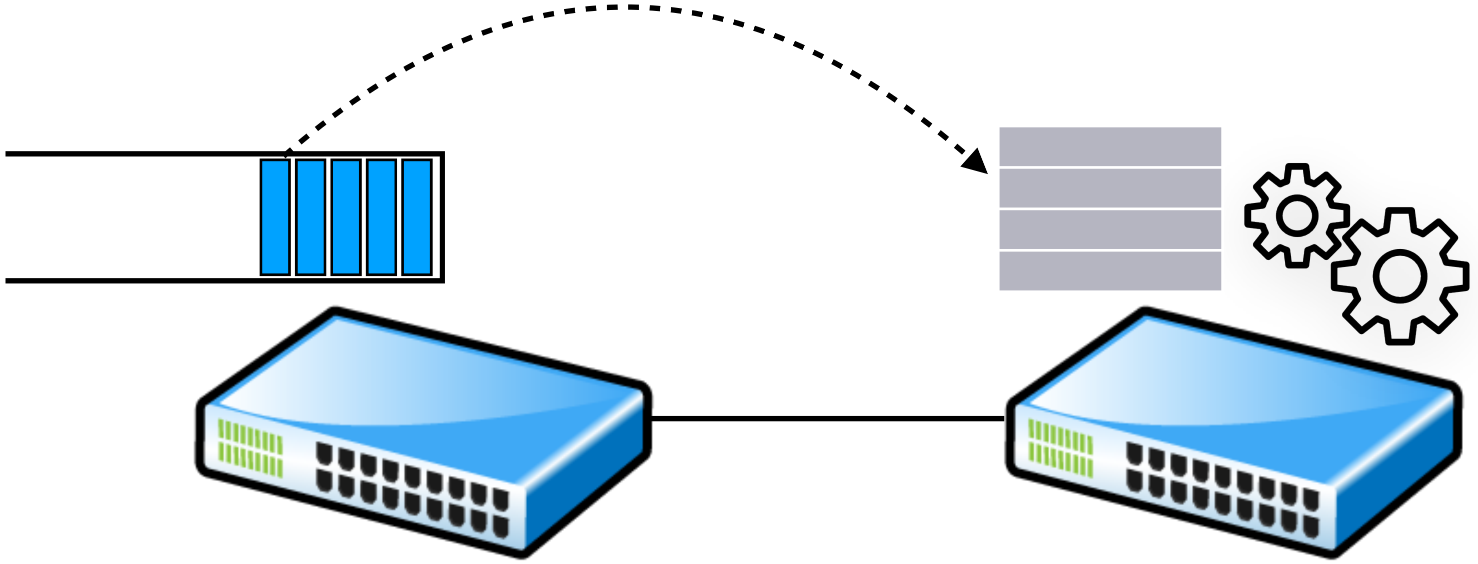
Offline algorithm (Belady) uses knowledge of future state accesses to make optimal decisions, but …

Existing online caching heuristics
lack awareness of future state accesses!

Research Question

Can future-aware online caching be realized in a network setting?

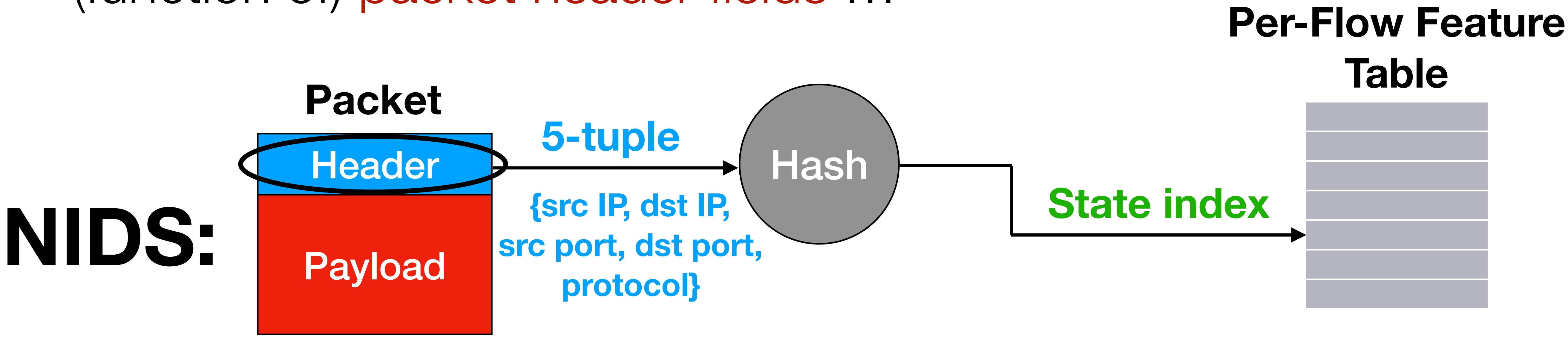
Seer[1]



[1] Jason Lei, Vishal Shrivastav. “Seer: Enabling Future-Aware Online Caching in Networked Systems”. USENIX NSDI 2024.

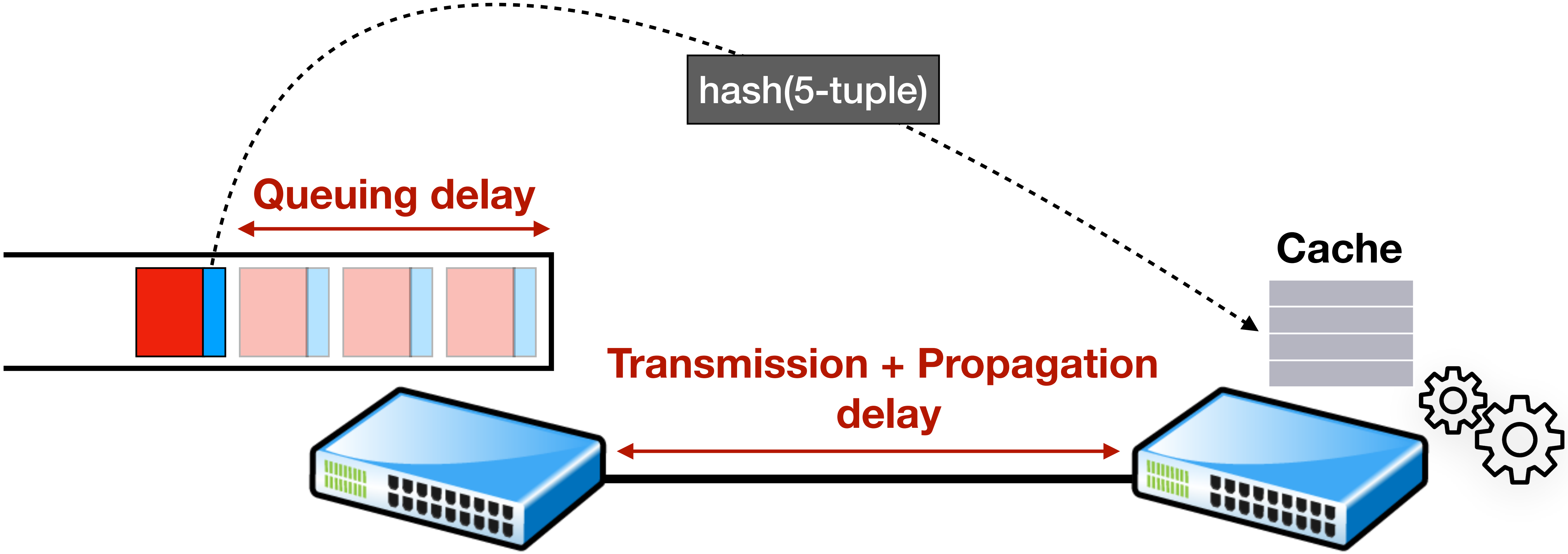
Insight 1: Header-based State Indexing

- State accesses in network applications are often triggered by incoming packets, and the state is often indexed by some (function of) packet header fields …



State access indices are carried in incoming packet headers, and can be encoded using a small number of bits (e.g., hash(5-tuple))

Insight 2: Leverage Network Delays



Delays in the network can be leveraged to forward state index information in advance

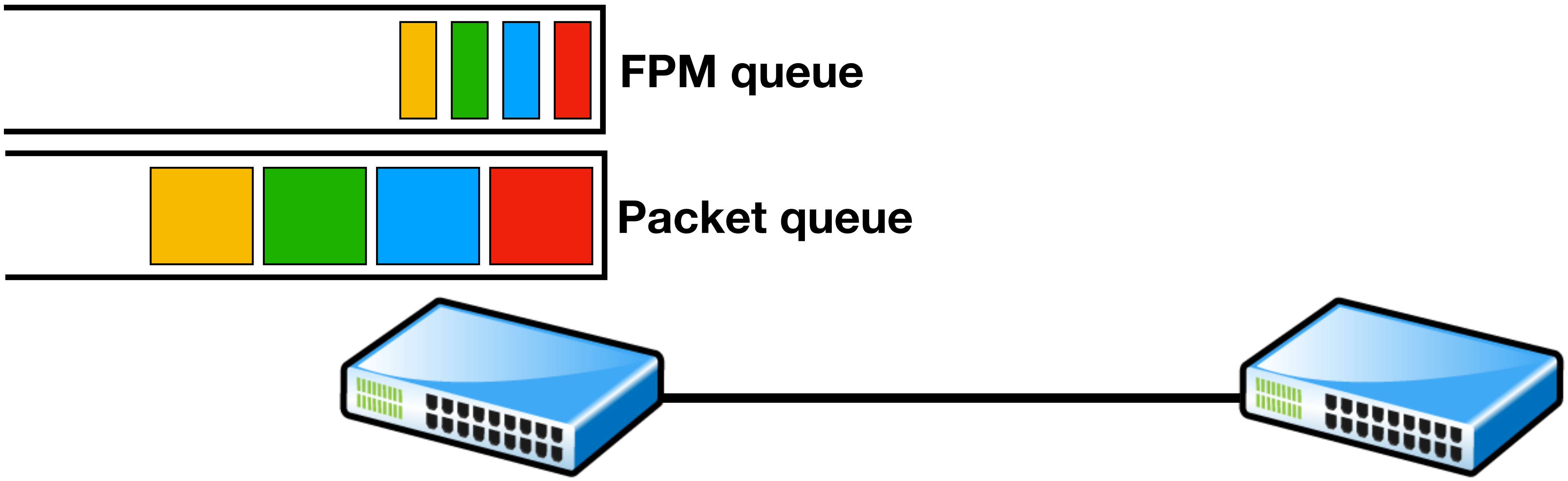
Two Challenges

1. How to forward future state access indices in a timely manner and at low bandwidth overhead?

2. How to leverage (partial) future state access information to make smarter cache prefetching and eviction decisions?

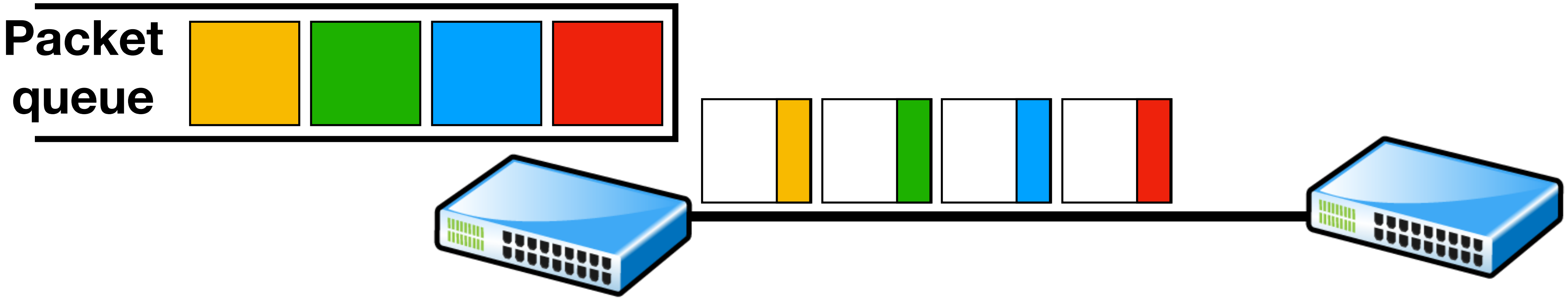
Low Overhead Notification

- For each queued packet, switch stores a **metadata**
  - Future Packet Metadata (FPM) for a given packet p:
    - Stores (i) State access index carried in packet p (ii) future time at which packet p will arrive at the next hop
- Seer only exchanges FPMs between **directly-connected nodes**
  - Results in most accurate estimation of future packet arrival time



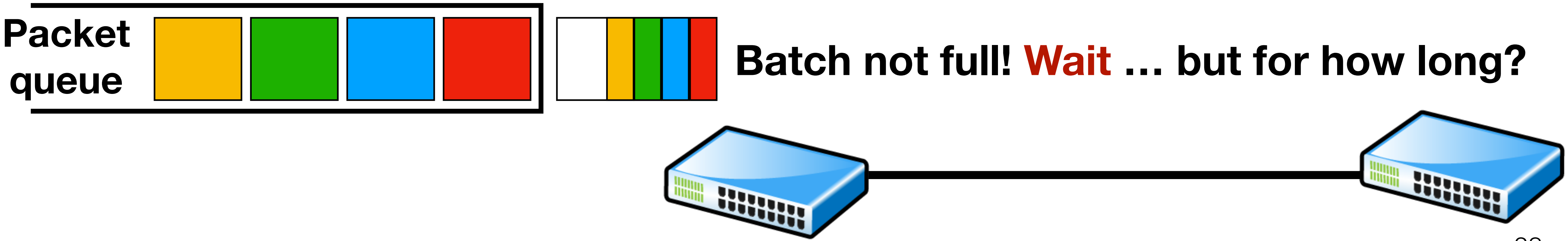
How to exchange FPMs?

- **Option 1:** Create a control packet for each FPM
  - Bandwidth wastage
    - Ethernet packets are at least 64B; FPMs are ~5-6B



Packet queue

- **Option 2:** Batch multiple FPMs into a single control packet
  - Non-deterministic batching latency

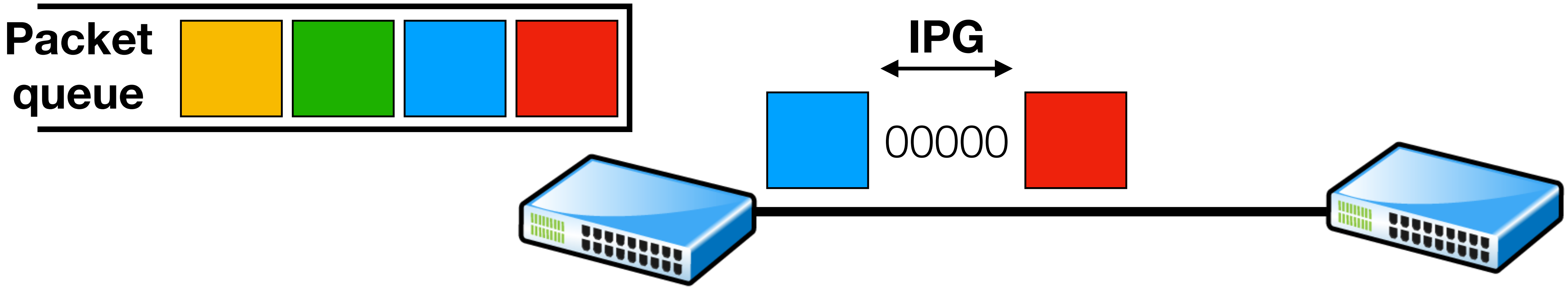


Packet queue

Batch not full! Wait … but for how long?

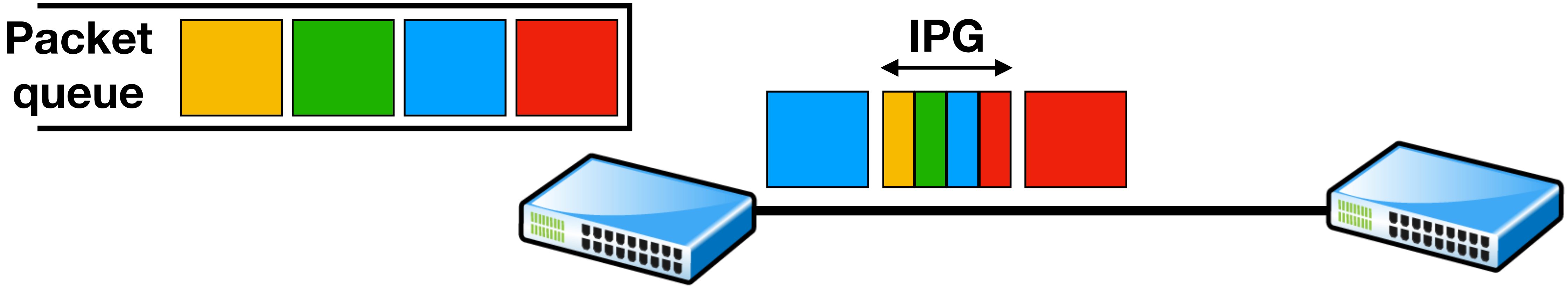
How to exchange FPMs?

- **Seer's solution:**
  - Ethernet mandates a minimum of 96 bits IPG (filled w/ 0s)



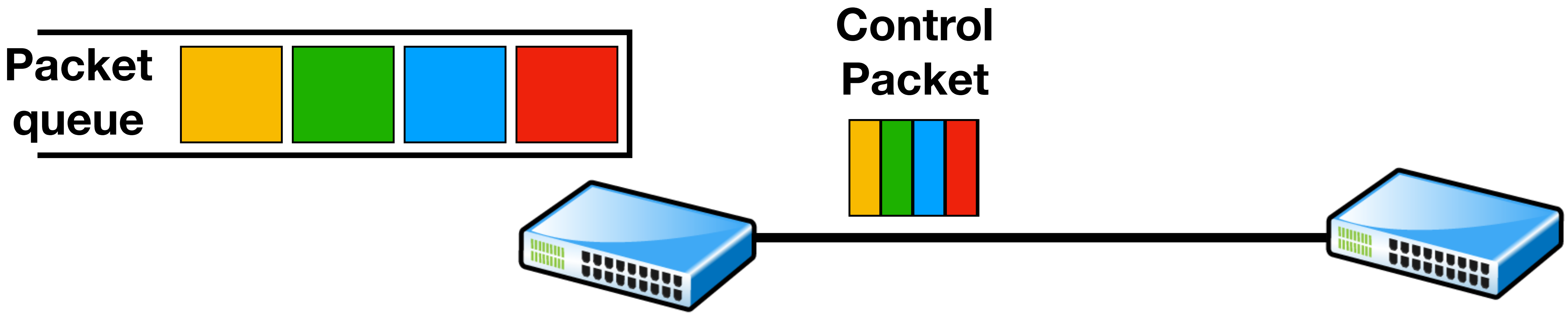
How to exchange FPMs?

- **Seer's solution:** Set a configurable batch parameter **m**
  - If # of FPMs < m:
    - Send FPMs in the **inter-packet gap (IPG)**
      - Ethernet mandates a minimum of 96 bits IPG (filled w/ 0s)
      - Seer replaces 0s with FPMs! **Zero bandwidth overhead!**



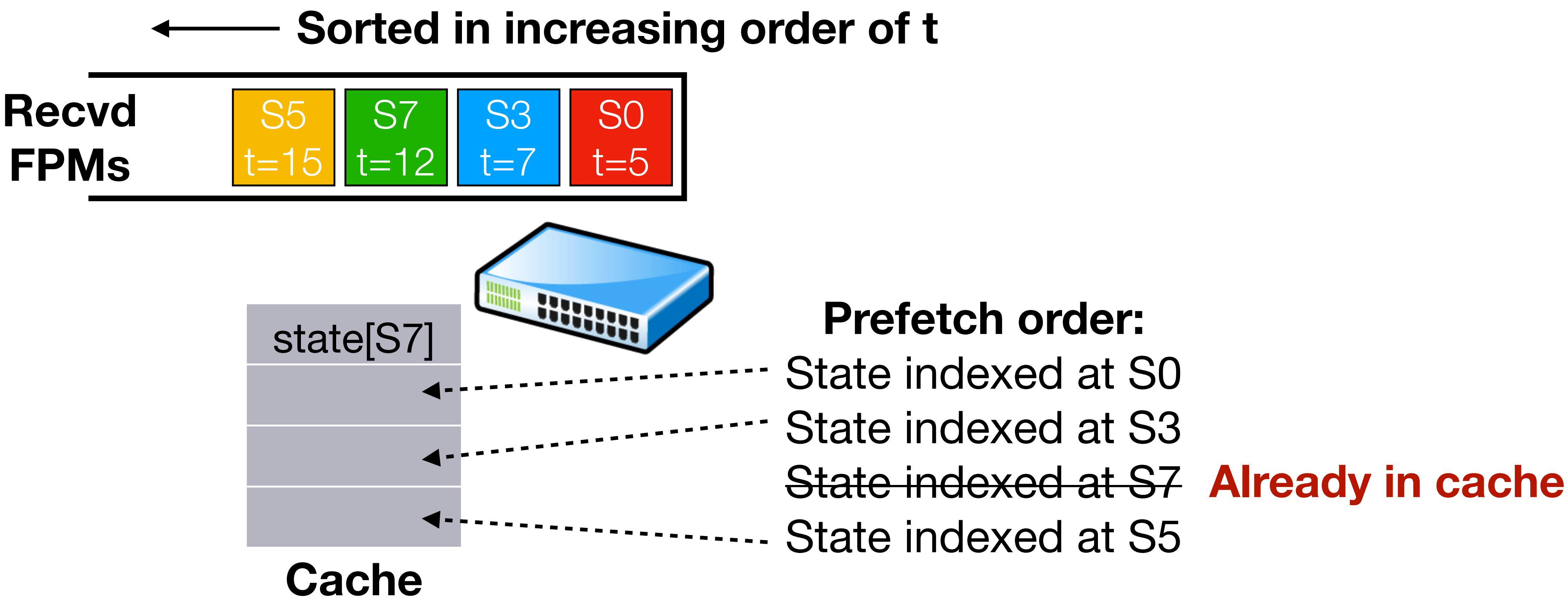
How to exchange FPMs?

- **Seer's solution:** Set a configurable batch parameter **m**
  - If # of queued FPMs < m:
    - Send FPMs in the **inter-packet gap (IPG)**
      - Ethernet mandates a minimum of 96 bits IPG (filled w/ 0s)
      - Seer replaces 0s with FPMs! **Zero bandwidth overhead!**
  - If # of queued FPMs >= m:
    - Batch FPMs and send in a single control packet
  - **No latency overhead!**



Future-Aware Prefetching & Eviction

- **Prefetching**
  - Goal: Fetch state in order of predicted time of access
    - Seer sorts received FPMs in the increasing order based on the future time of packet arrival field in FPM
    - Seer prefetches state (not in cache) in the order of sorted FPMs



Prefetch order:
State indexed at S0
State indexed at S3
~~State indexed at S7~~ **Already in cache**
State indexed at S5

Future-Aware Prefetching & Eviction

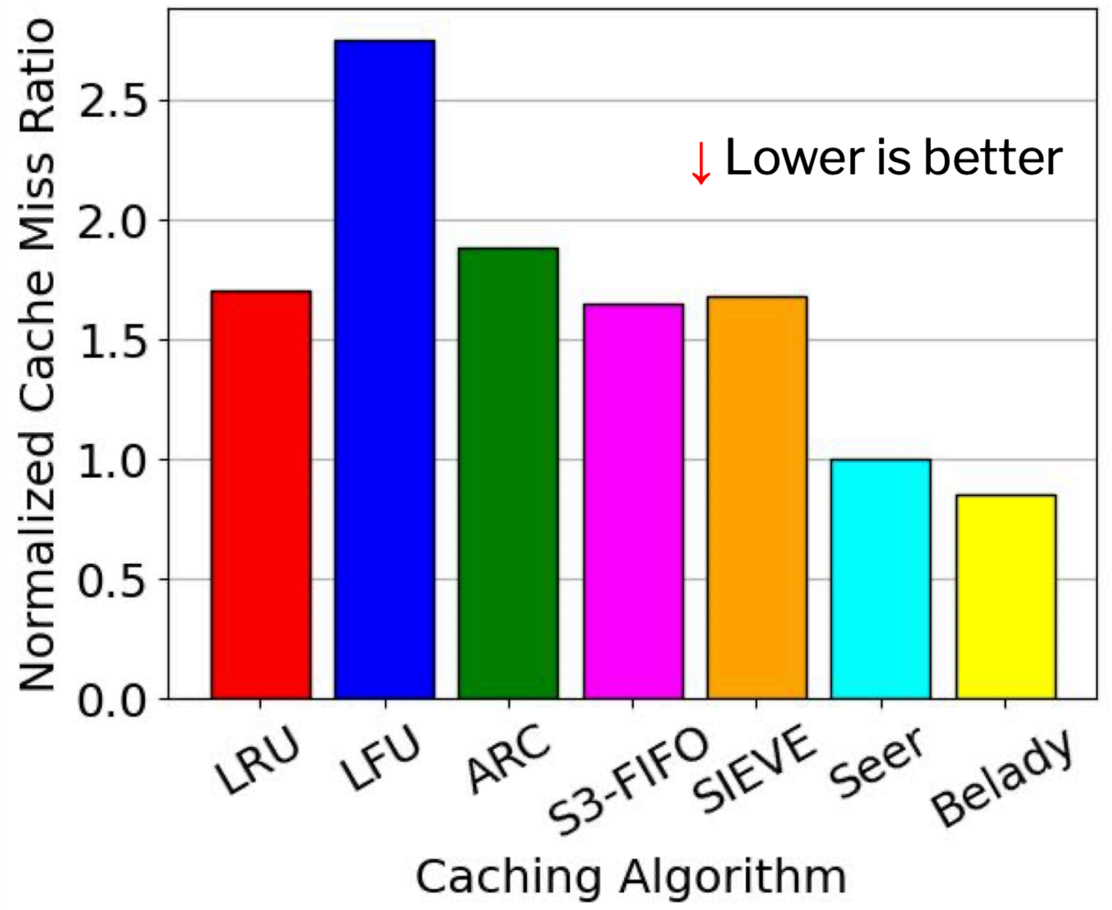
- **Cache Eviction**
  - Goal: Emulate Belady, i.e., evict the state that will be accessed farthest in the future
    - **Split cache into two sets:** states with known future access time (i.e., for which FPMs were received) vs. unknown future access time
    - Prioritize evicting states with unknown future access time
      - Can use ***any*** caching heuristic for such evictions
      - **Worst case:** Seer reduces to the caching heuristic
    - If all states in the cache have known future access time
      - Evict state to be accessed farthest in the future (Belady)
      - **Best case:** Seer reduces to Belady

Seer's Performance

Setup:

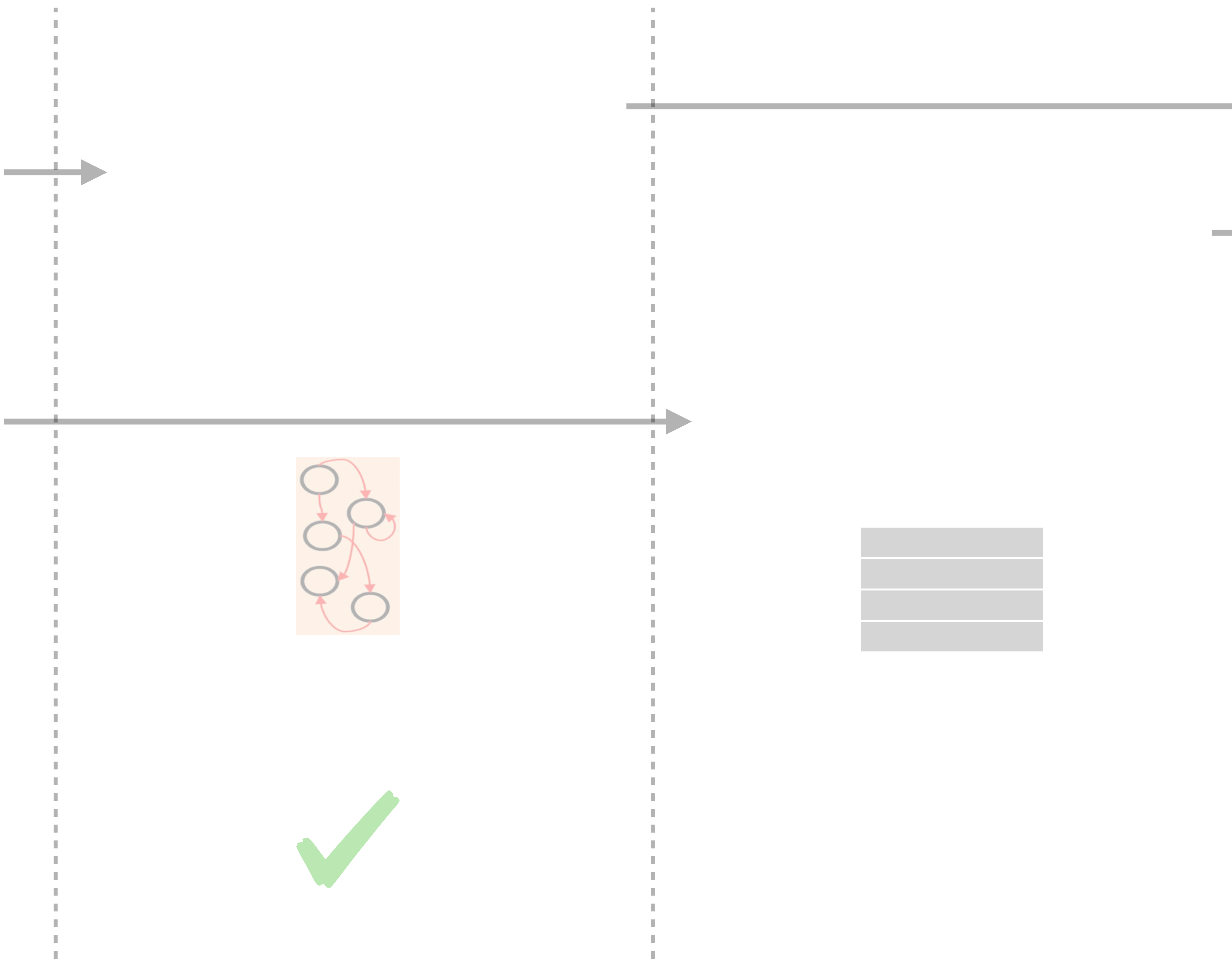
- Simulated Fattree topology with 144 hosts, 9 ToRs, 16 spine switches
- All links 100 Gbps, 100ns per hop latency, 100ns DRAM access latency
- DCTCP congestion control, ECMP load balancing
- Two in-network applications:
  - ToR switches running intrusion detection
  - Spine switches running load balancing

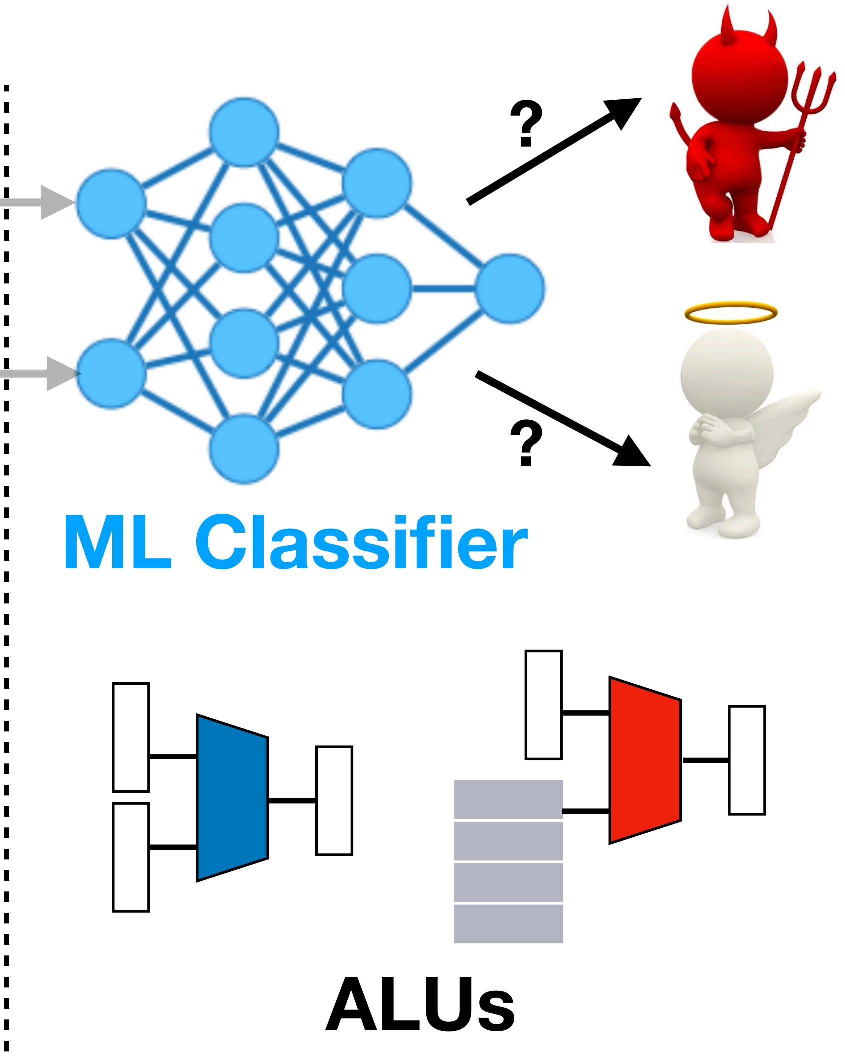
- Seer achieves **60-180% lower** cache miss ratio that state-of-the-art caching heuristics
- Seer's cache miss ratio is **within 15%** of Belady (optimal)



Normalized w.r.t. Seer

Challenge (2): Compute-Intensive



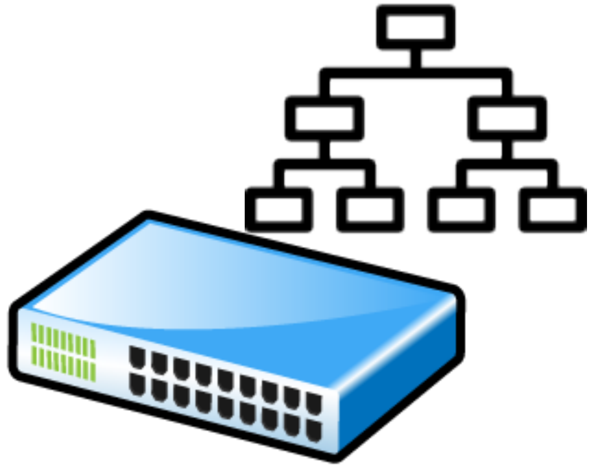


Limited # of ALUs
Simple ALU ops
Limited # of pipeline stages

Design efficient compute utilization and multiplexing techniques

ML Classifier on Programmable Switch

Leo[2]

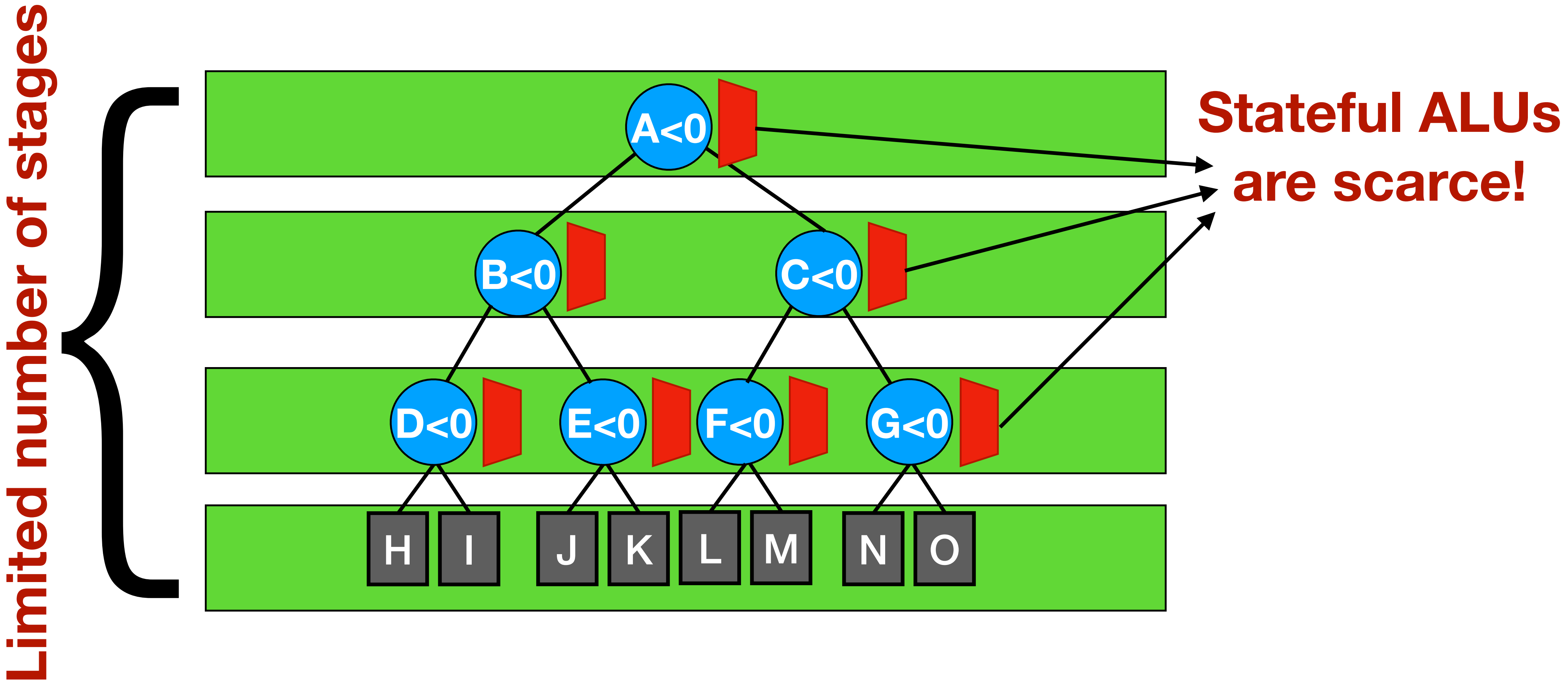


- Leo focuses on **decision tree** based ML classifiers
  - Easily interpretable — liked by network operators
  - Competitive classification accuracy compared to DNNs for intrusion detection application
  - Simple ALU operations (e.g., comparison) — supported on programmable switch

[2] Syed Usman Jafri, Sanjay Rao, *Vishal Shrivastav,* Mohit Tawarmalani. "Leo: Online ML-based Traffic Classification at Multi-Terabit Line Rate". USENIX NSDI 2024.

Naive Mapping

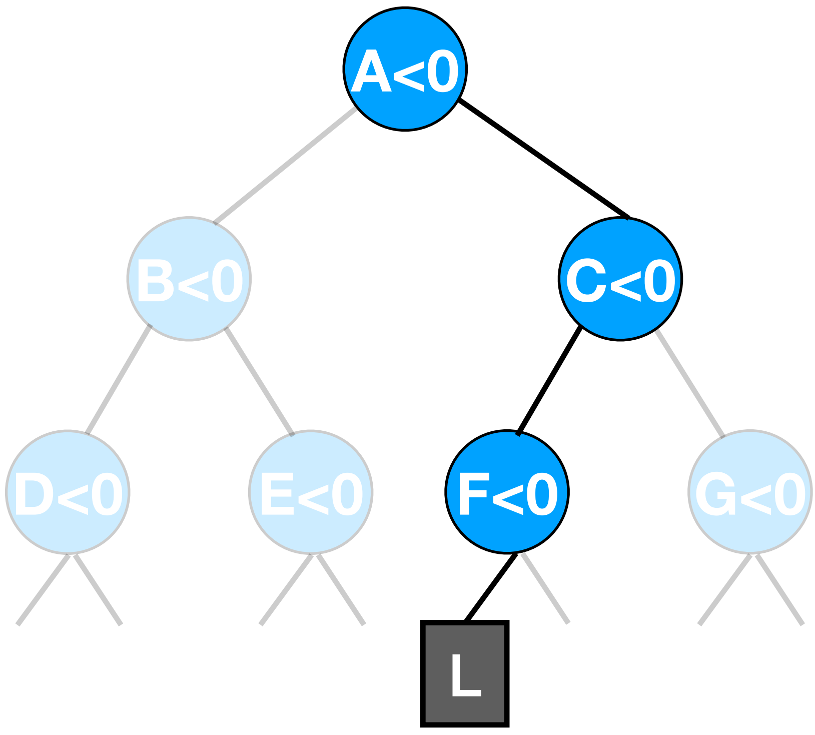
- **Decision tree has a natural mapping to Match-Action stages**
  - Each level of the tree can be mapped to one Match-Action stage



Fundamentally limits the size of a decision tree!

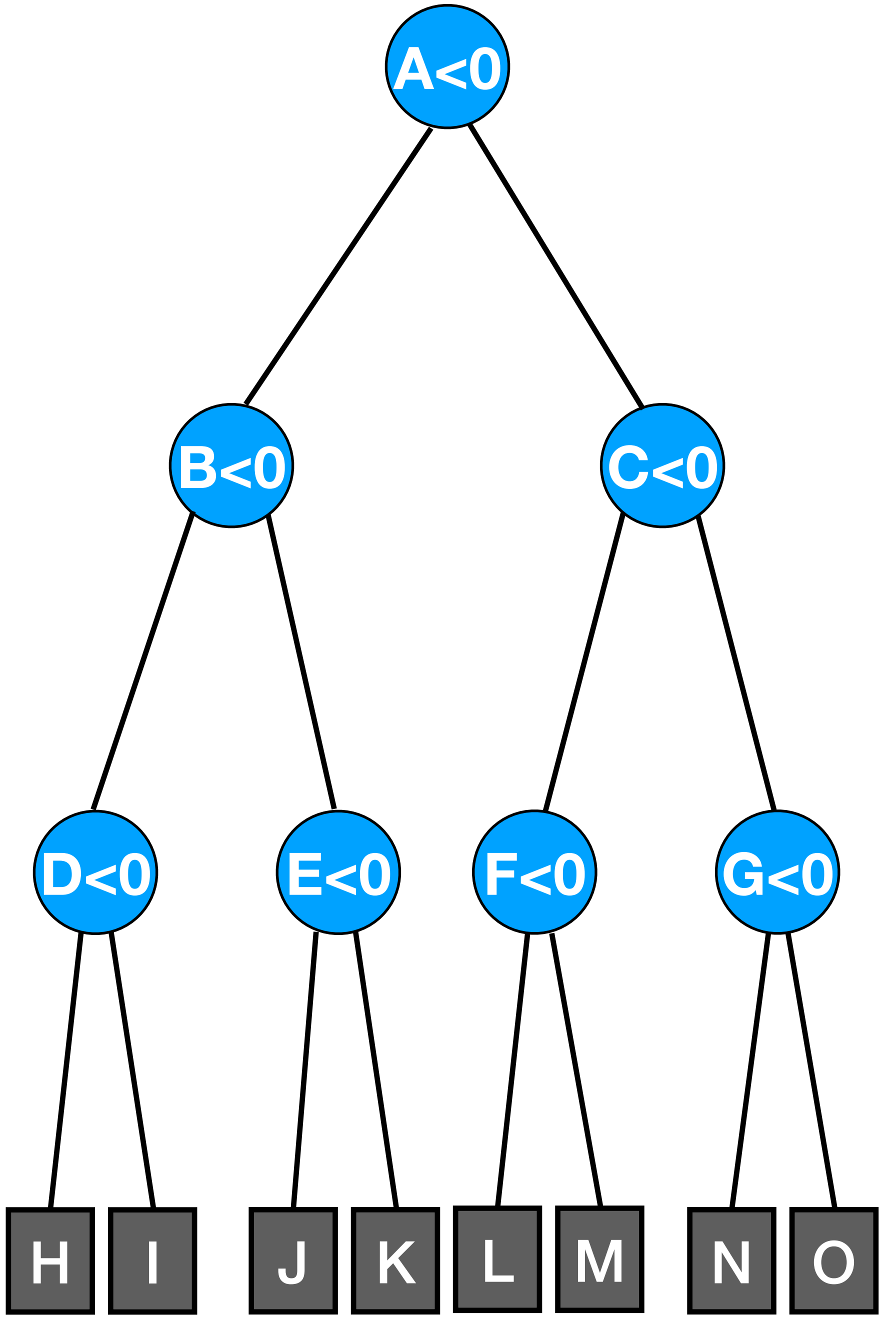
Idea (1): Node Multiplexing

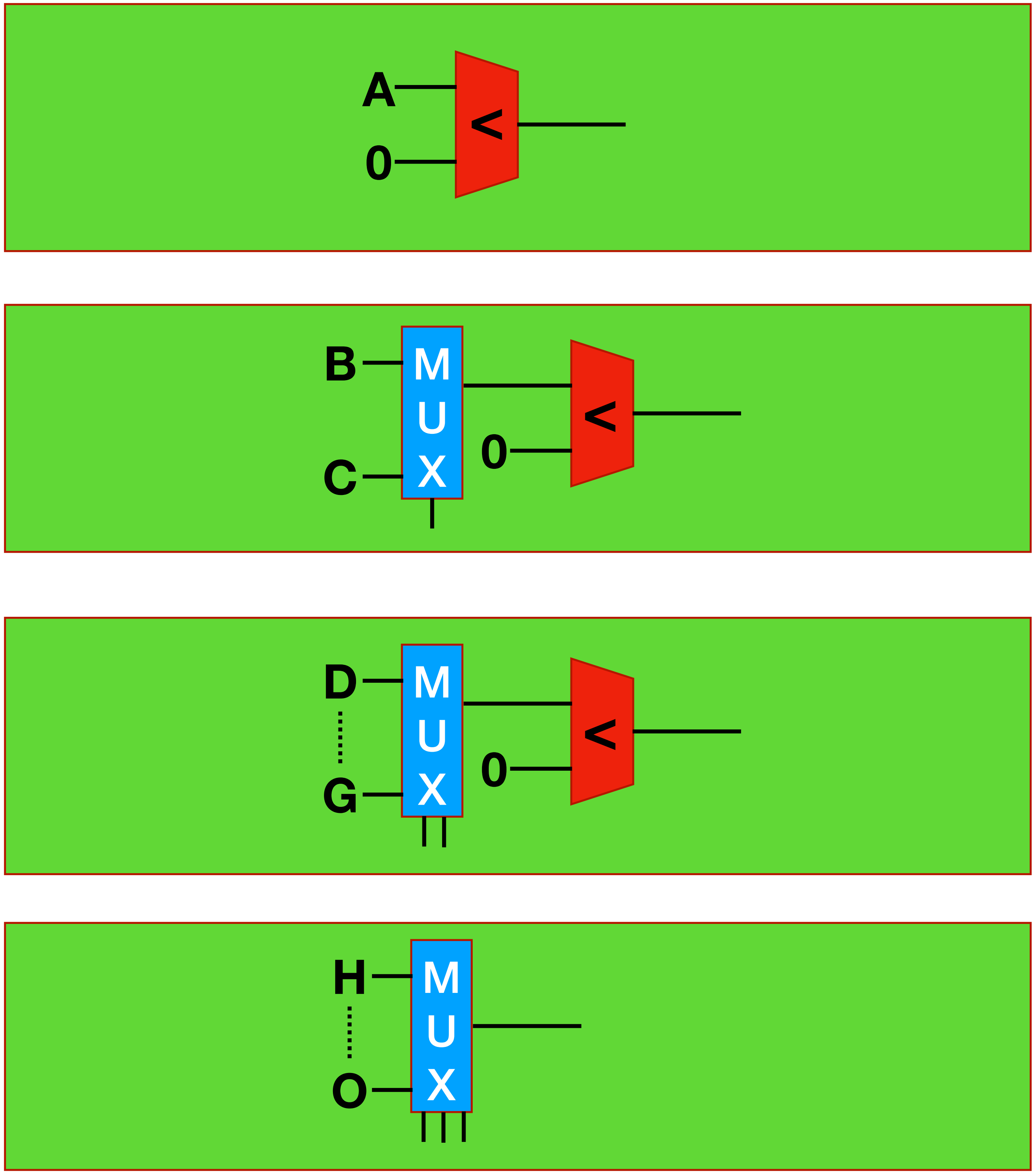
- **Key Observation: At runtime, only 1 node per level is accessed**
  - Allocate resources for only one node per level
  - At runtime, multiplex the feature comparisons at each node



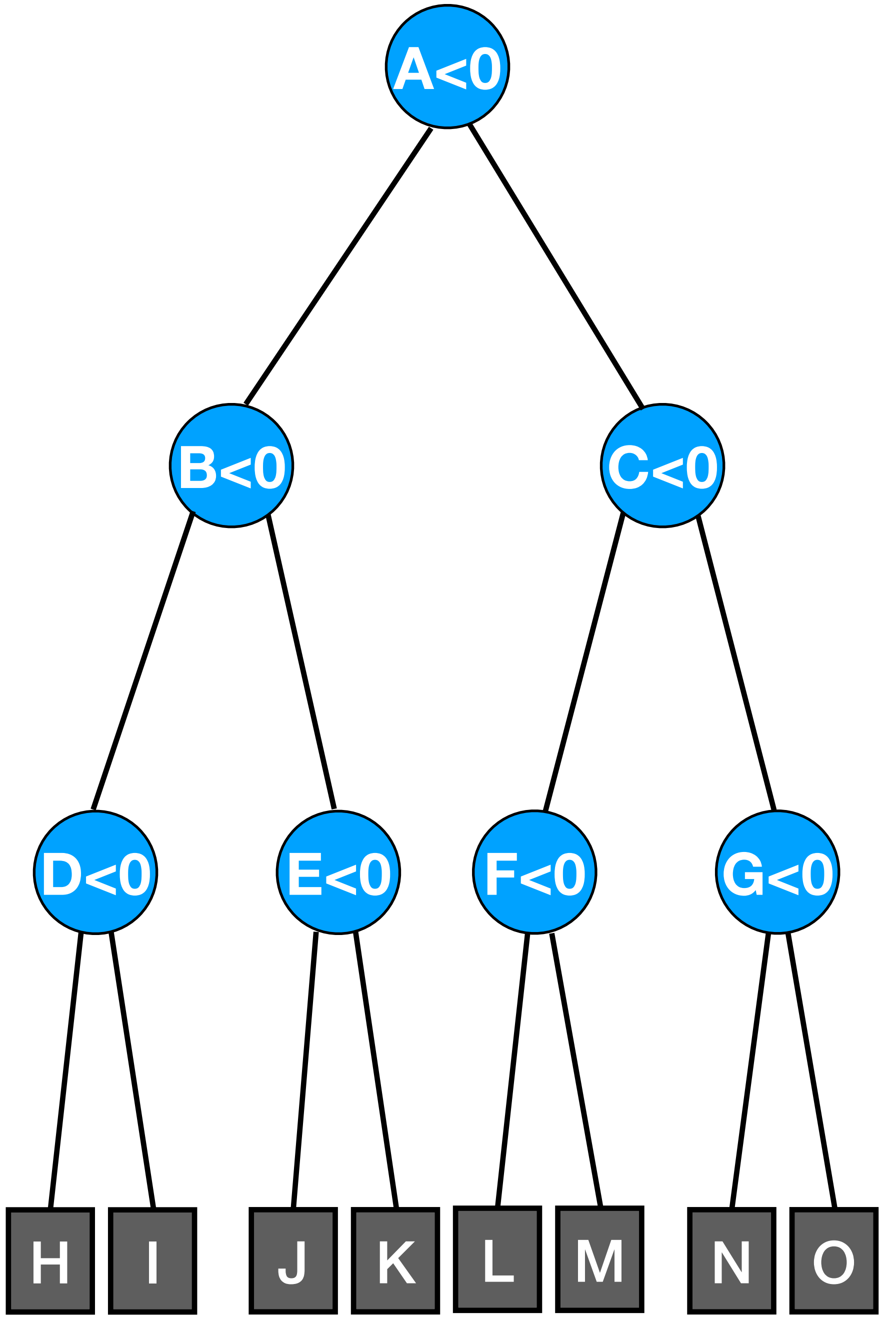
Results in compute (ALU) efficiency!

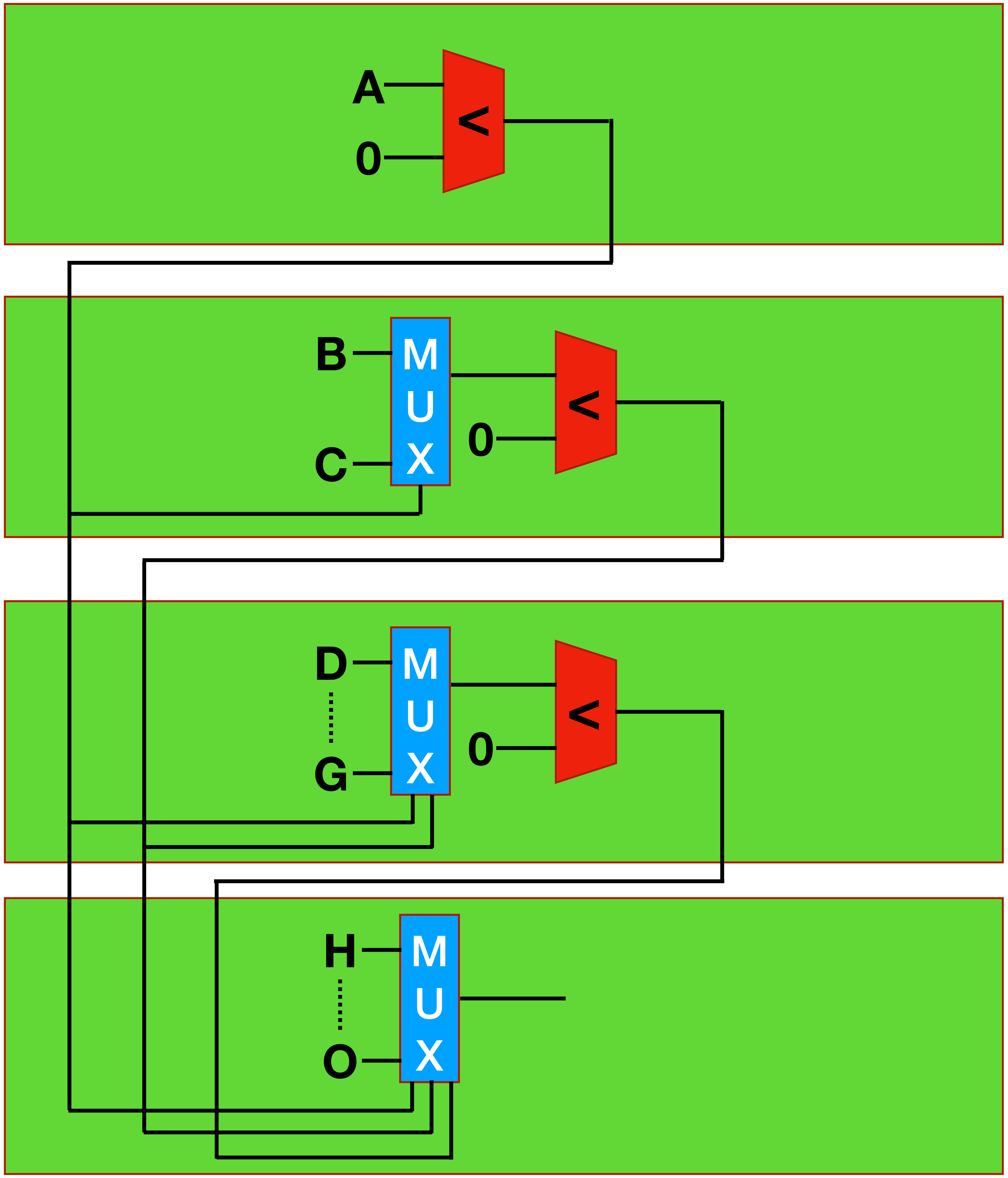
Idea (1): Node Multiplexing





Idea (1): Node Multiplexing



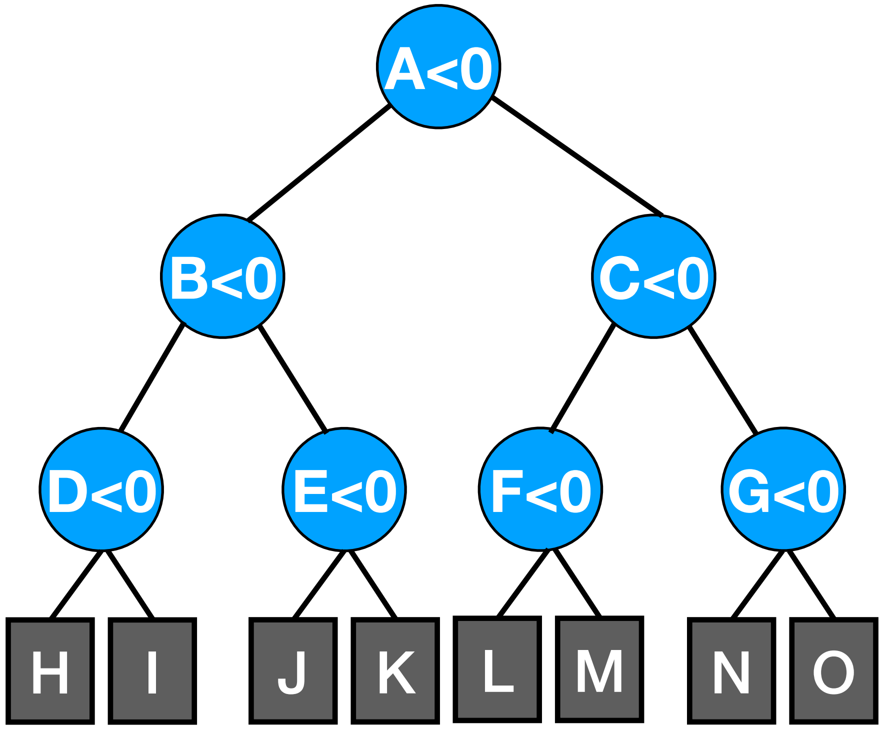


Limitation of Node Multiplexing

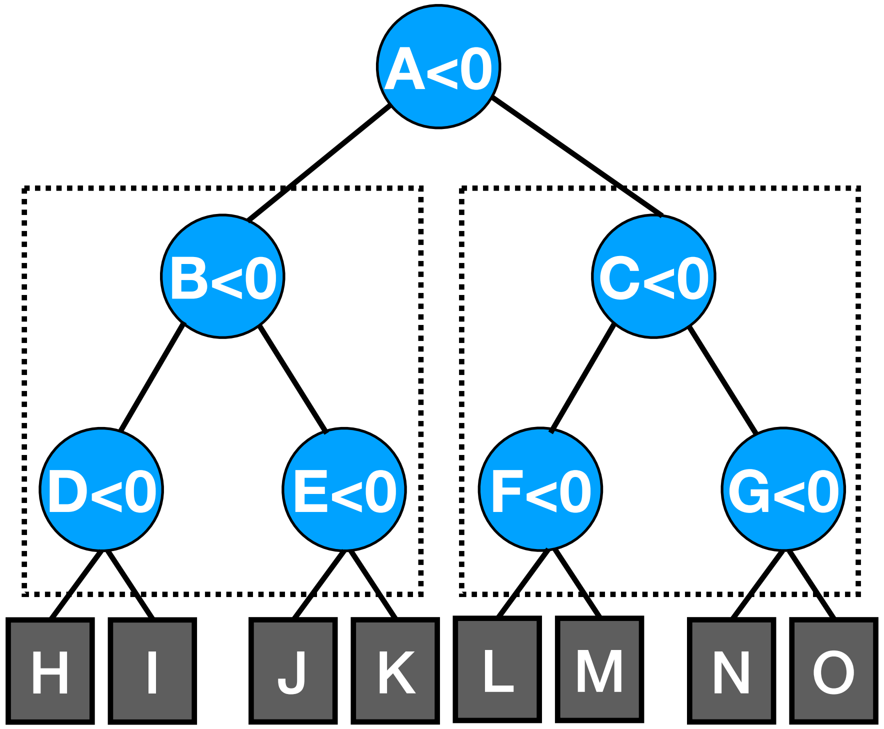
- **Depth of the decision tree is limited by # of switch stages**
    - A switch with *D* Match-Action stages can support a decision tree of depth at most *D*!

Question:
How to scale tree depth?

Idea (2): Subtree Flattening & Multiplexing



Idea (2): Subtree Flattening & Multiplexing

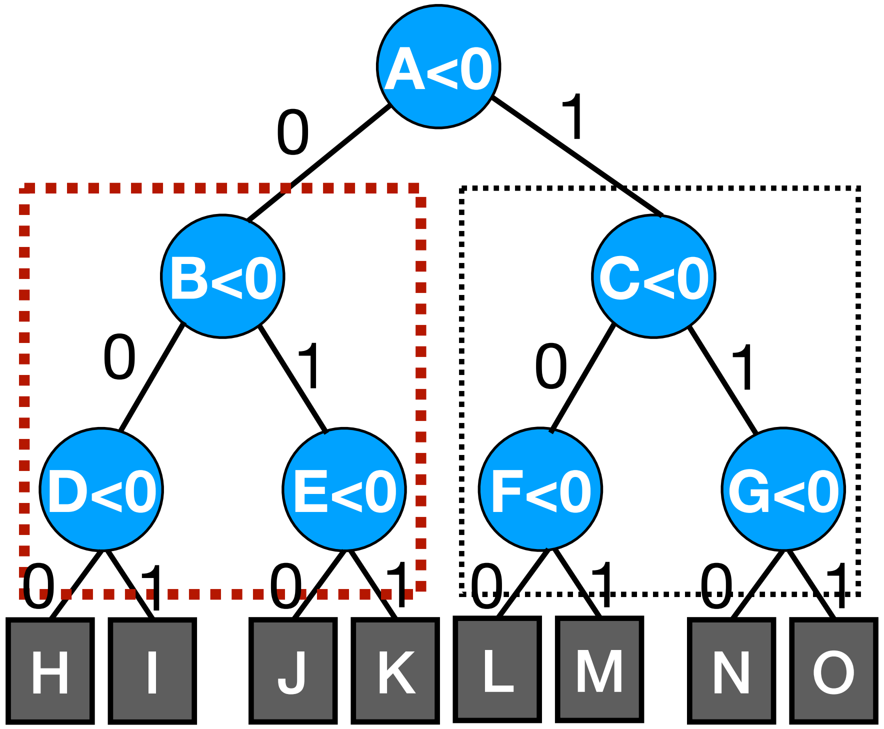


Insight: A decision tree can be represented as a boolean truth table

1. **Identify common subtree structures**

2. **Flatten the subtree**

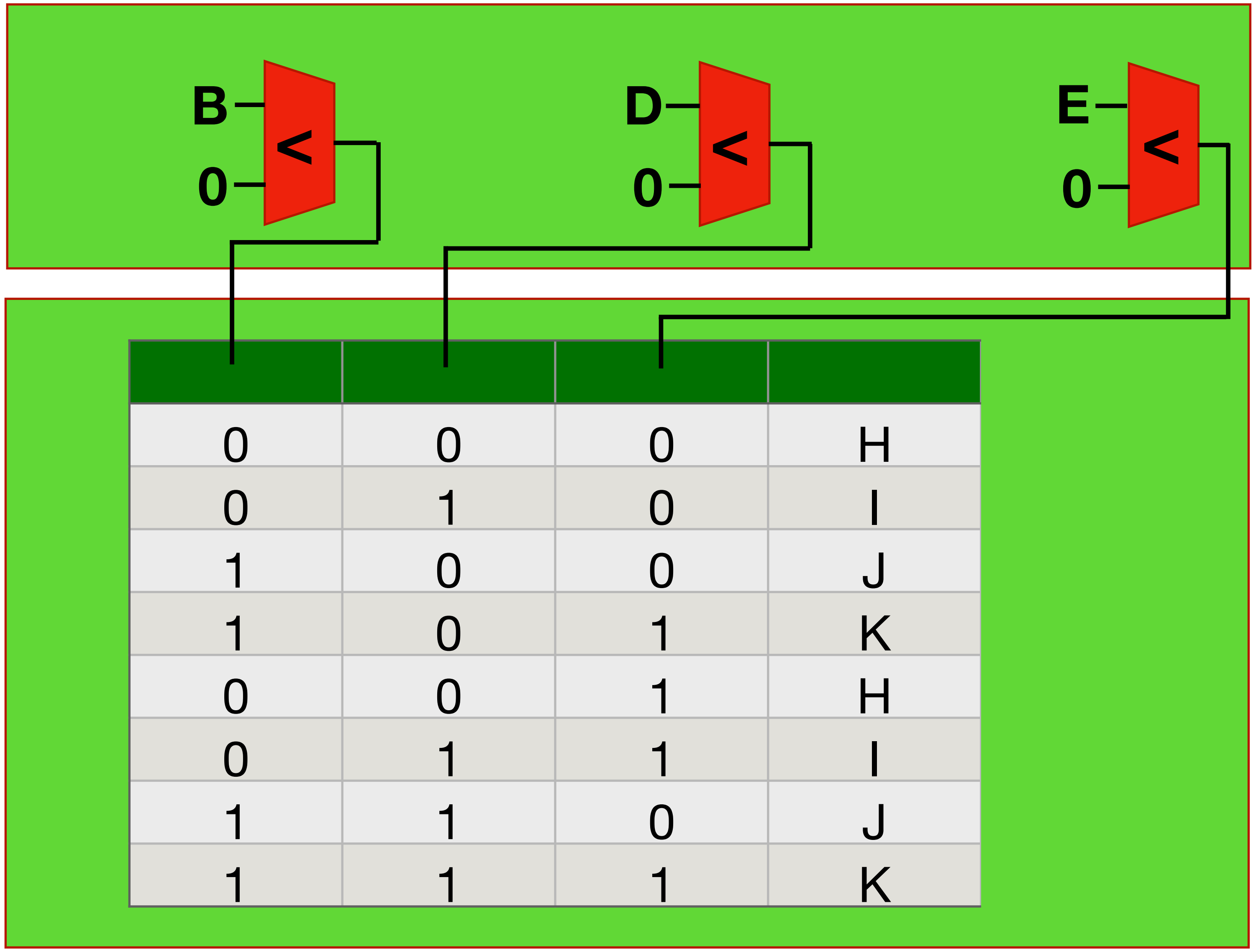
Idea (2): Subtree Flattening & Multiplexing



Insight: A decision tree can be represented as a boolean truth table

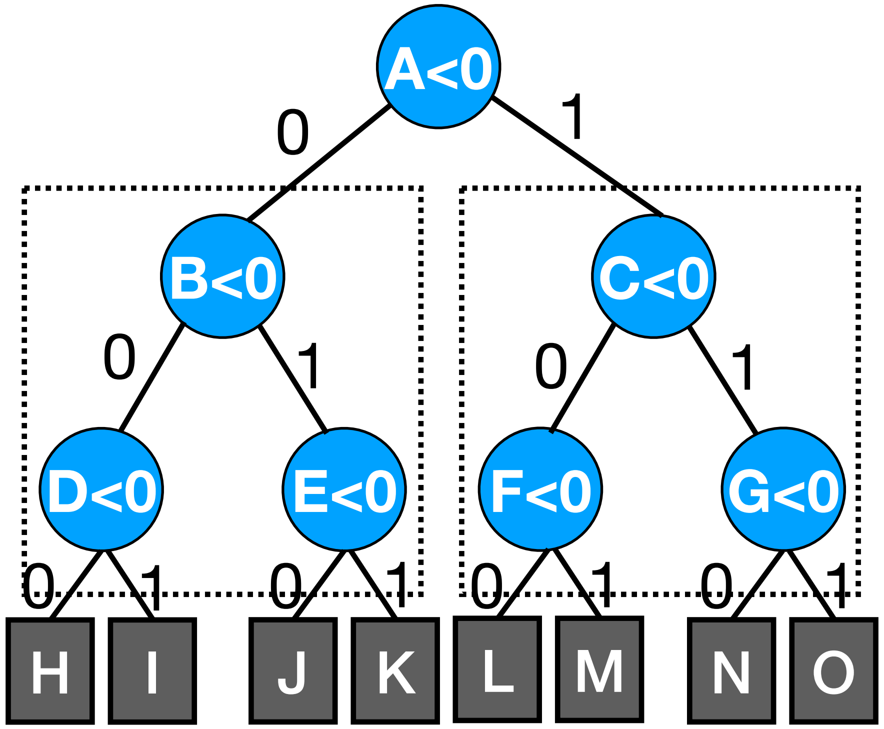
1. Identify common subtree structures

2. Flatten the subtree



0	0	0	H
0	1	0	I
1	0	0	J
1	0	1	K
0	0	1	H
0	1	1	I
1	1	0	J
1	1	1	K

Idea (2): Subtree Flattening & Multiplexing

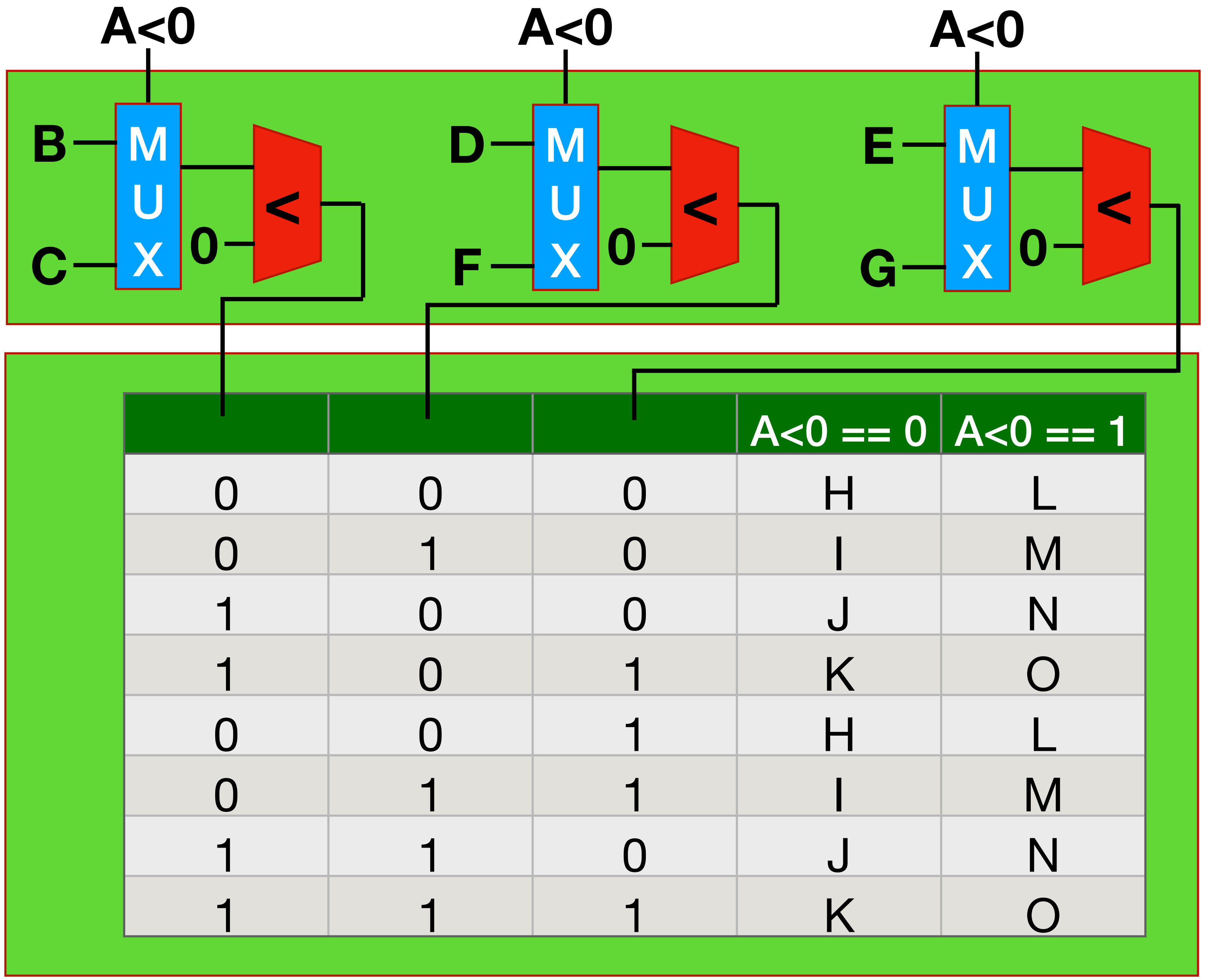


Only need 3 stages to support depth 4 tree!

(1 stage to implement A<0 plus 2 stages shown in the figure)

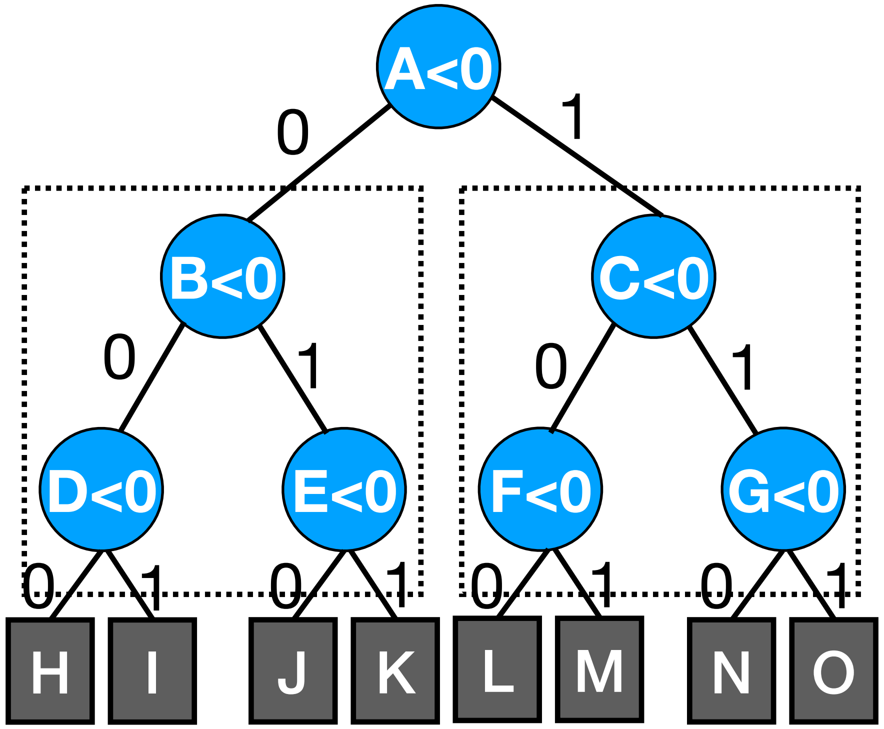
1. Identify common subtree structures

2. Flatten the subtree & Multiplex



			A<0 == 0	A<0 == 1
0	0	0	H	L
0	1	0	I	M
1	0	0	J	N
1	0	1	K	O
0	0	1	H	L
0	1	1	I	M
1	1	0	J	N
1	1	1	K	O

Why not flatten the entire tree?



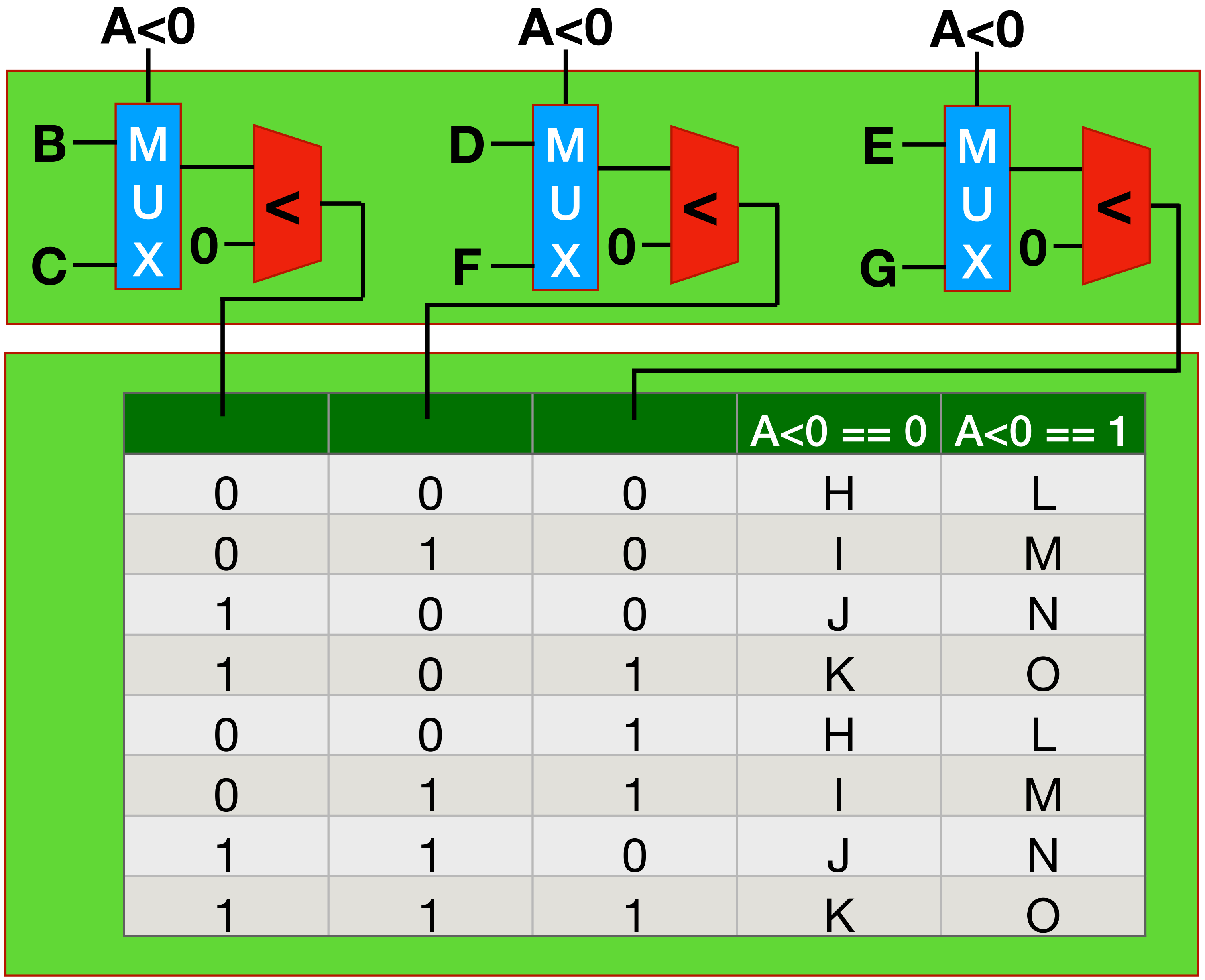
… because,

small # of stateful ALUs per stage

the size of table grows *exponentially* w/ tree size

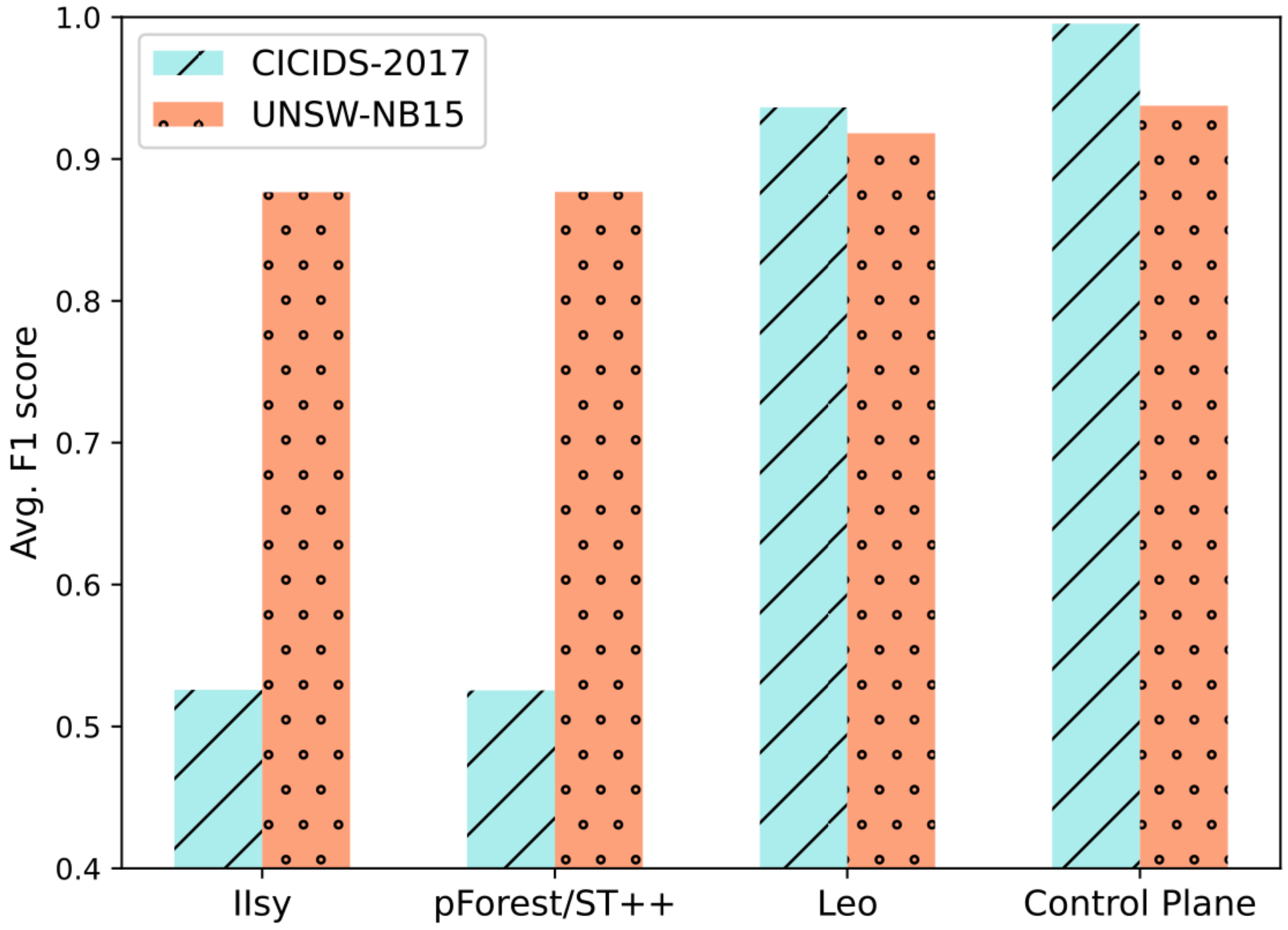
1. Identify common subtree structures

2. Flatten the subtree & Multiplex



			A<0 == 0	A<0 == 1
0	0	0	H	L
0	1	0	I	M
1	0	0	J	N
1	0	1	K	O
0	0	1	H	L
0	1	1	I	M
1	1	0	J	N
1	1	1	K	O

Classification Accuracy



↑ Higher is better

- Leo achieves significantly better F1 score than existing programmable switch based decision tree implementations (IIsy, pForest, ST) for CICIDS-2017 dataset — **because Leo can support larger trees!**
- Leo’s F1 score is close to control plane solution, where optimal sized decision tree is implemented on general purpose CPU

Closing Thoughts

- In-network computing presents a new paradigm for distributed computing, with the potential to improve the performance of many distributed systems and networked applications

- The challenge lies in the **domain-specific** processing of programmable network devices

- **Two broad research directions:**

  1. Enhance the **hardware** capability of programmable network devices
     - Seer improves the state caching subsystem
     - Richer compute units, improved state sharing, multi-pipeline processing
  2. Clever **algorithmic and systems** design choices to work-around the limitations of programmable network devices
     - Leo uses efficient compute multiplexing to implement ML classifier
     - Approximate algorithms and computations, distributed multi-switch computations, division of labor b/w switch, NIC, host

Thank you!

Acknowledgments:

The presented work was generously supported by grants from

NSF National Science Foundation

Google

cisco