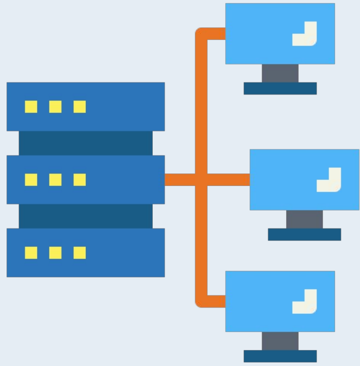
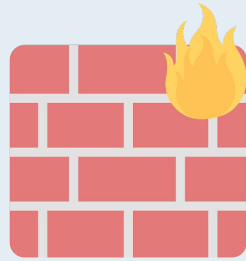
Automatic Parallelization of Software Network Functions

Francisco Pereira, Fernando Ramos, Luis Pedrosa

Middleboxes are pervasive in today's networks

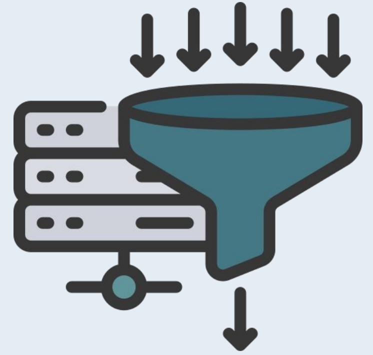


Load Balancers



Firewalls

DPI



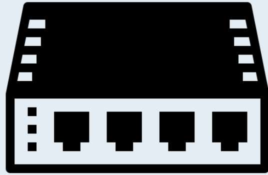
Rate Limiters

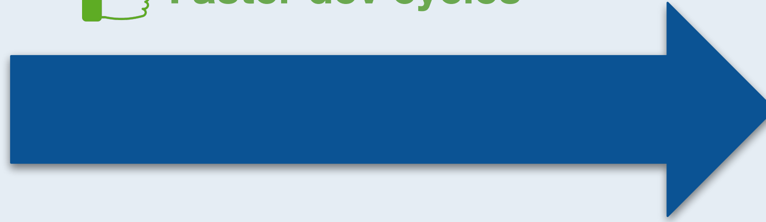
Trading performance for flexibility

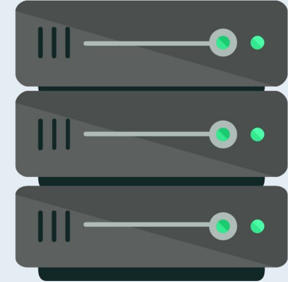
Lower costs

Easier management

Faster dev cycles







Fixed-function closed-source appliances

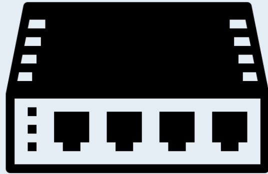
Software middleboxes

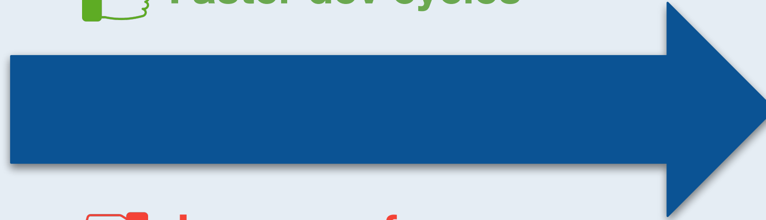
Trading performance for flexibility

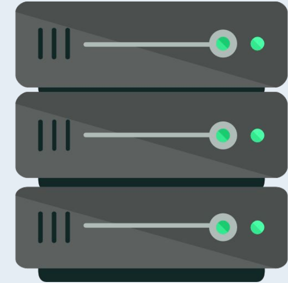
Lower costs

Easier management

Faster dev cycles





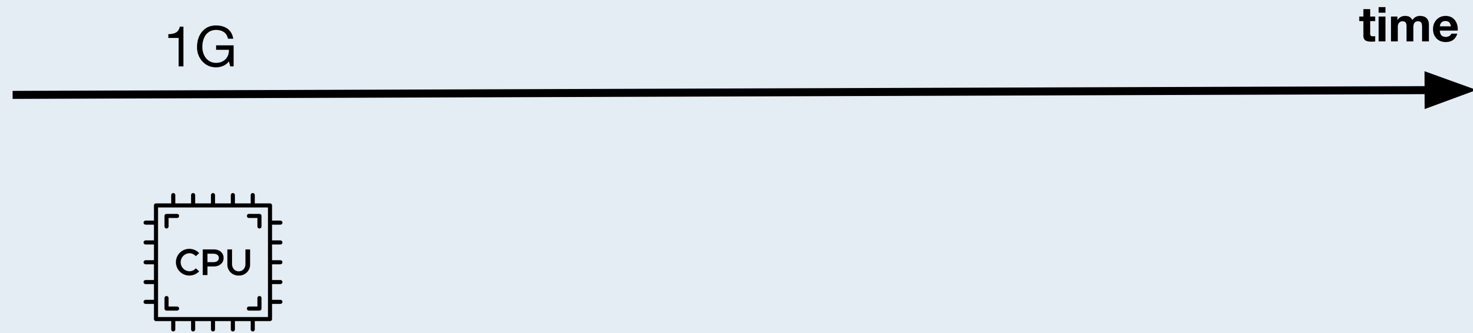


Lower performance

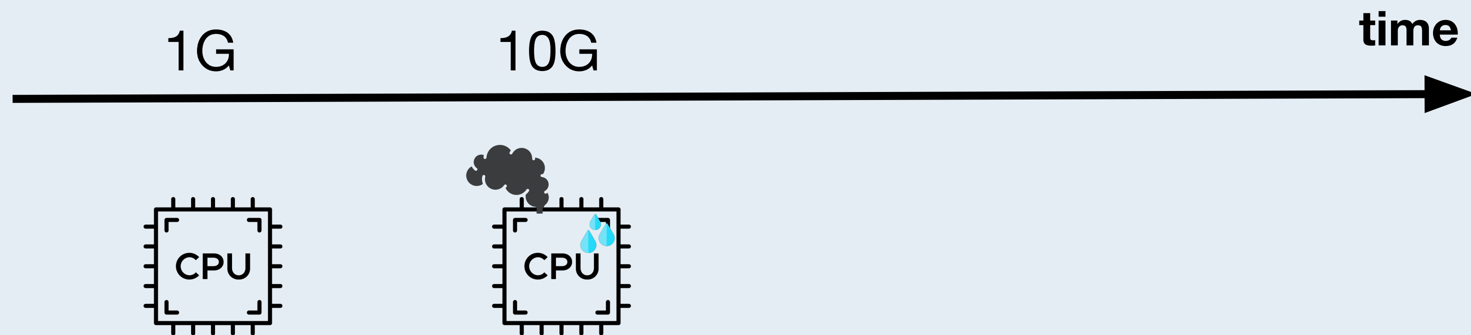
Fixed-function closed-source appliances

Software middleboxes

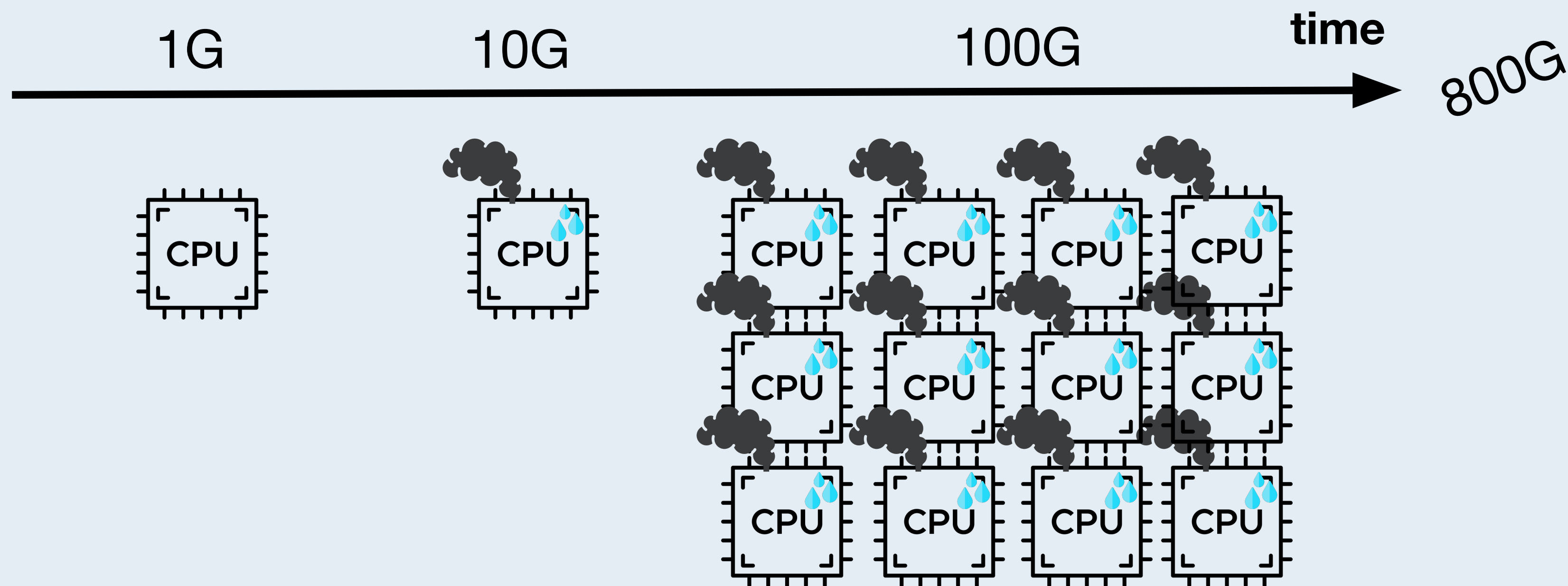
Line-rates just keep increasing



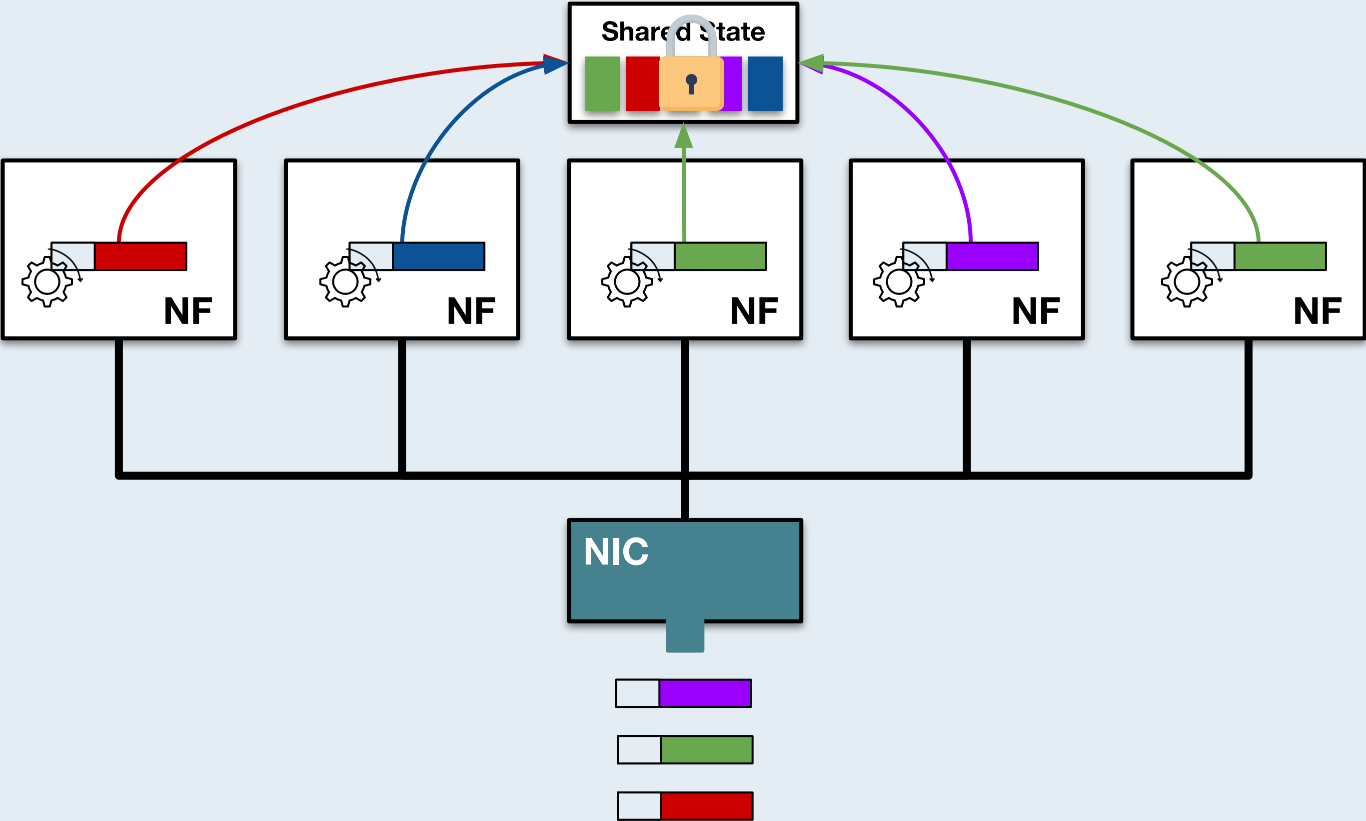
Line-rates just keep increasing



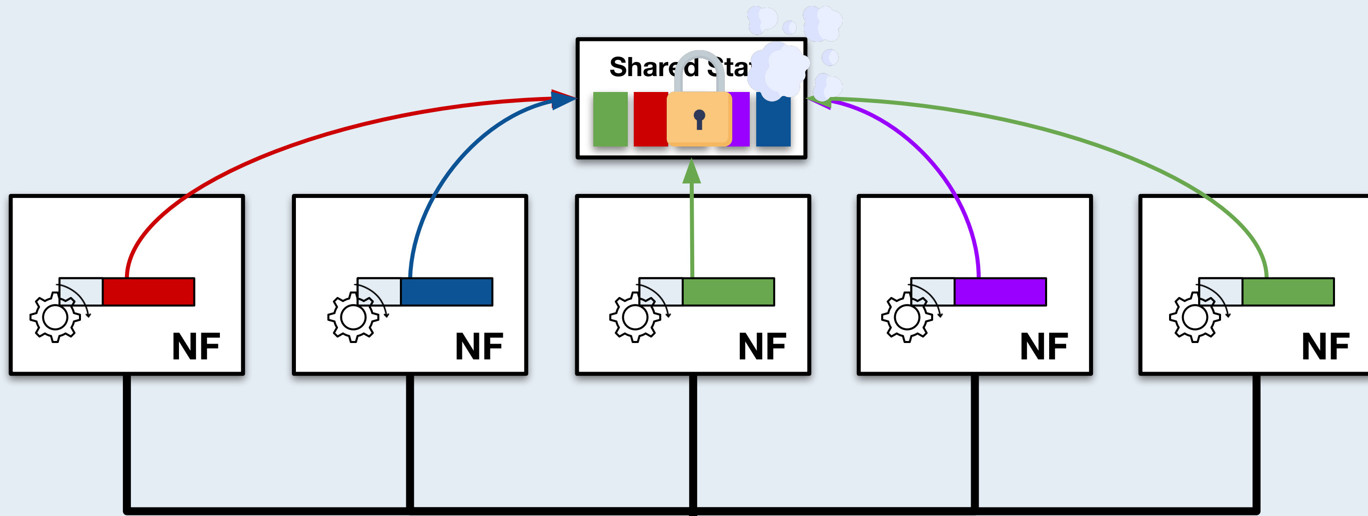
Line-rates just keep increasing



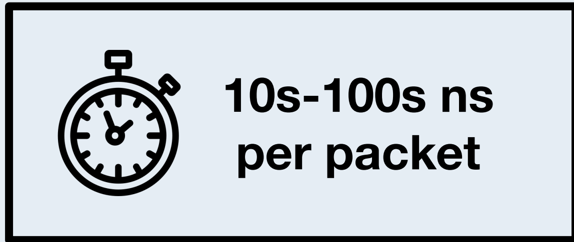
Parallelization in a nutshell



There is no time for synchronization

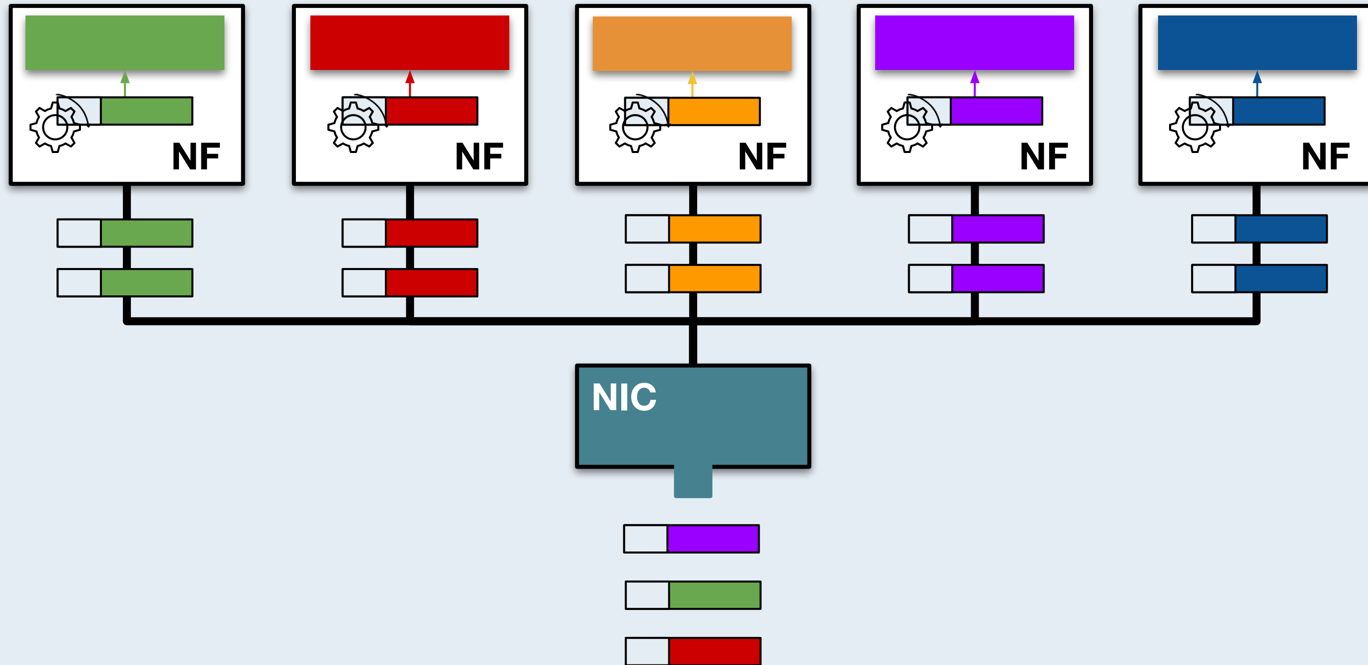


NIC



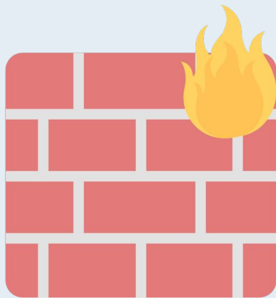
Avoiding inter-core coordination is paramount to achieving high performance in parallel implementations

Shared-nothing architecture

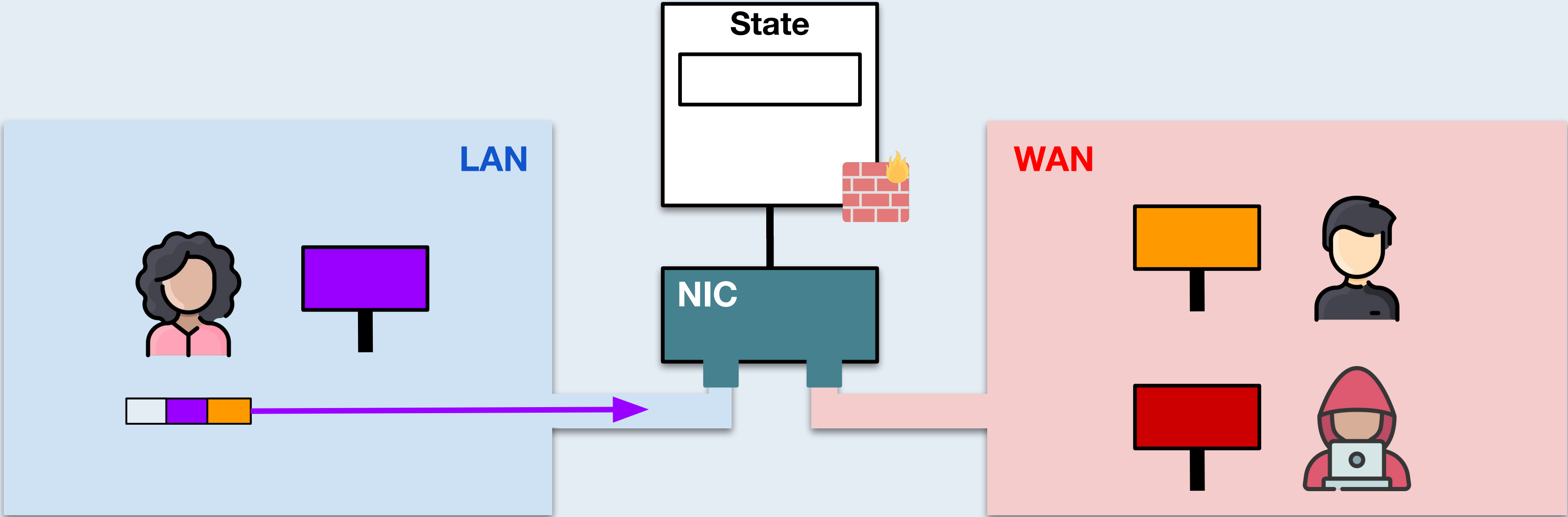


Why is parallelization hard?

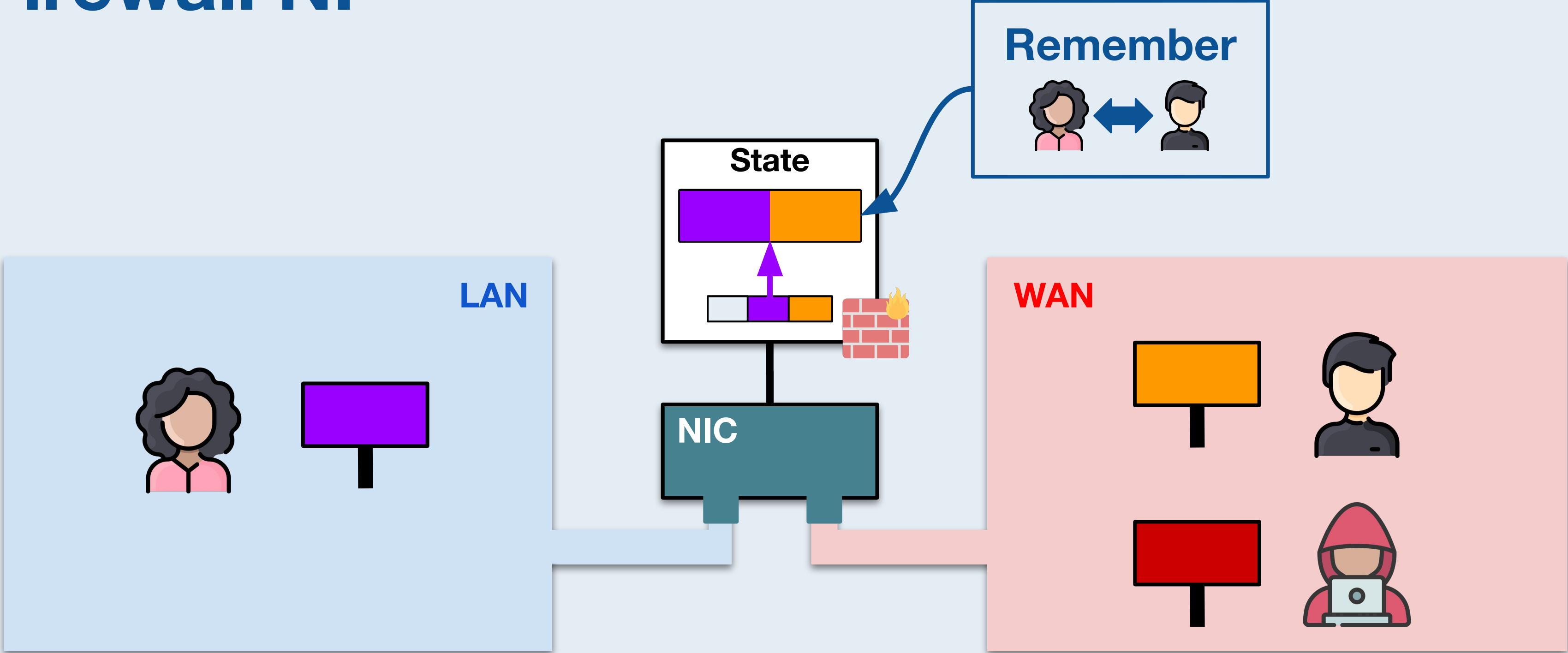
Let’s use a firewall as
an example



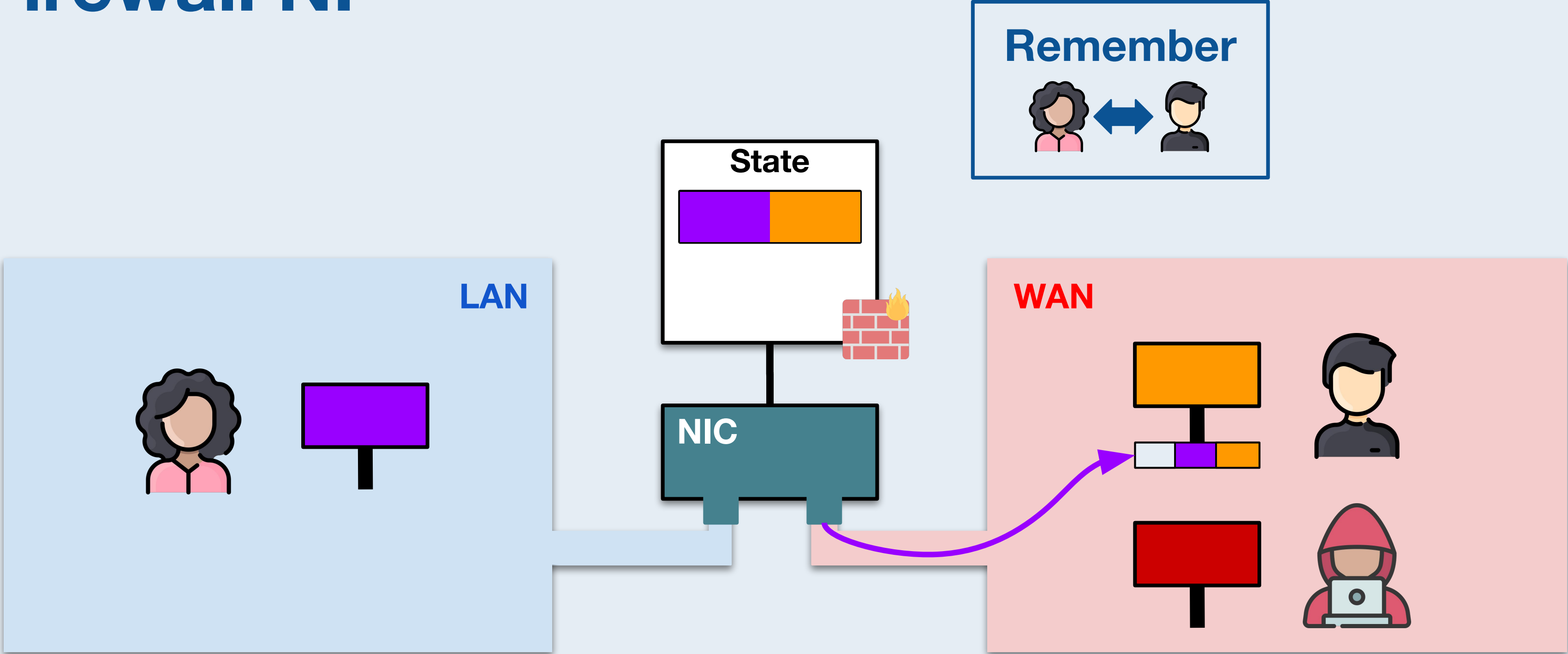
Firewall NF



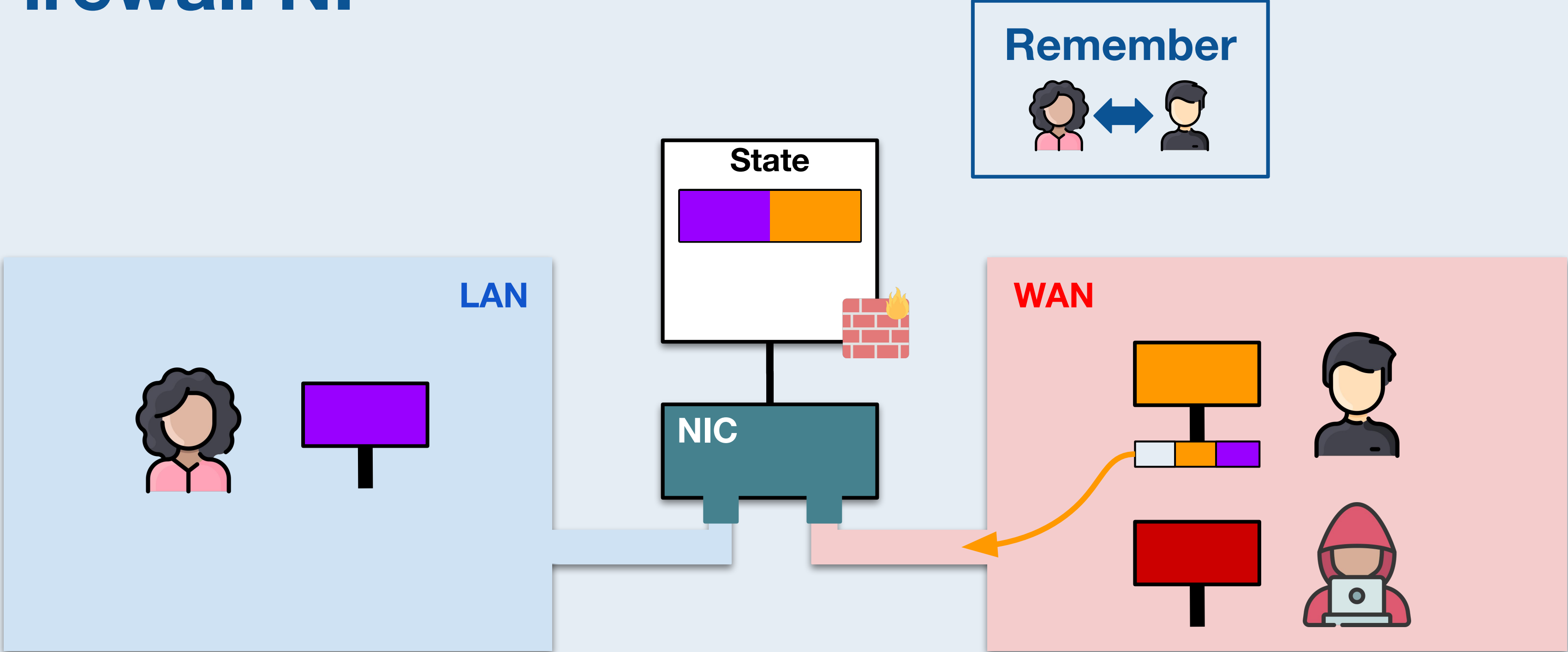
Firewall NF



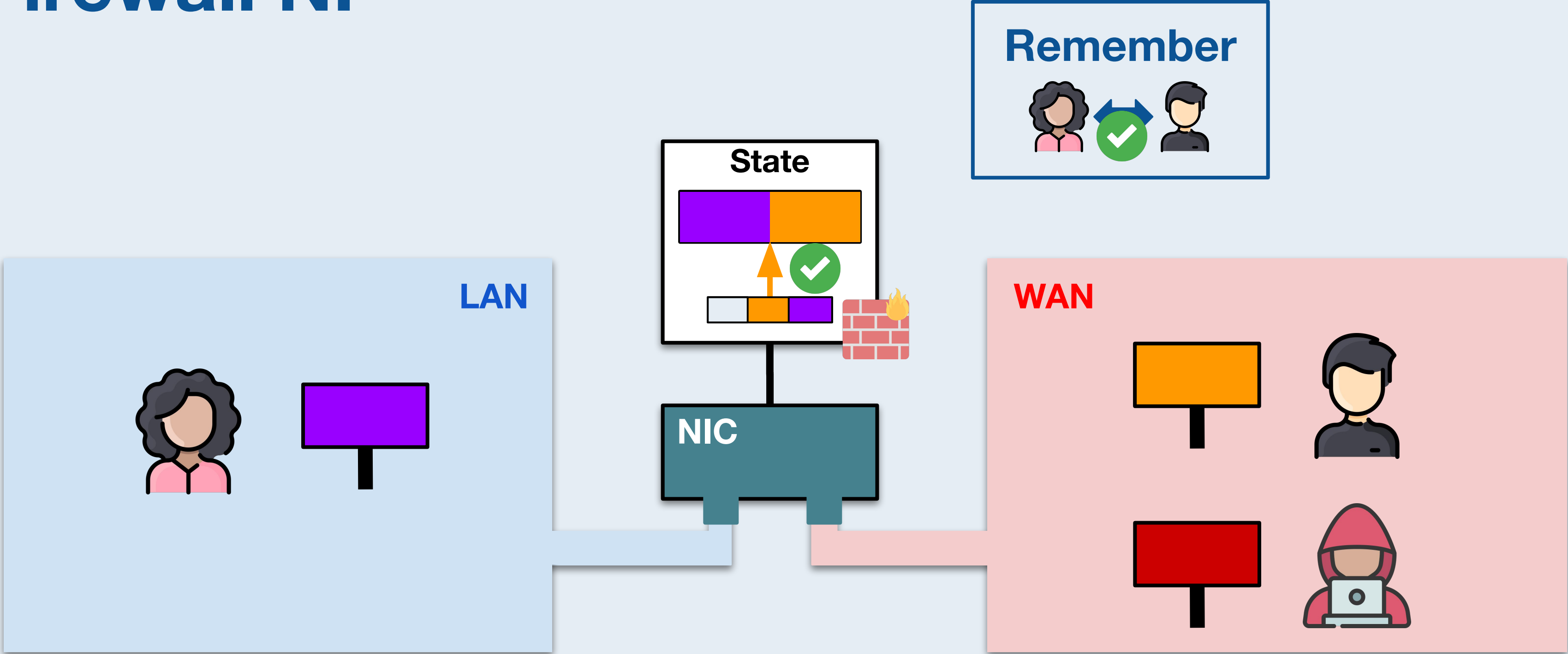
Firewall NF



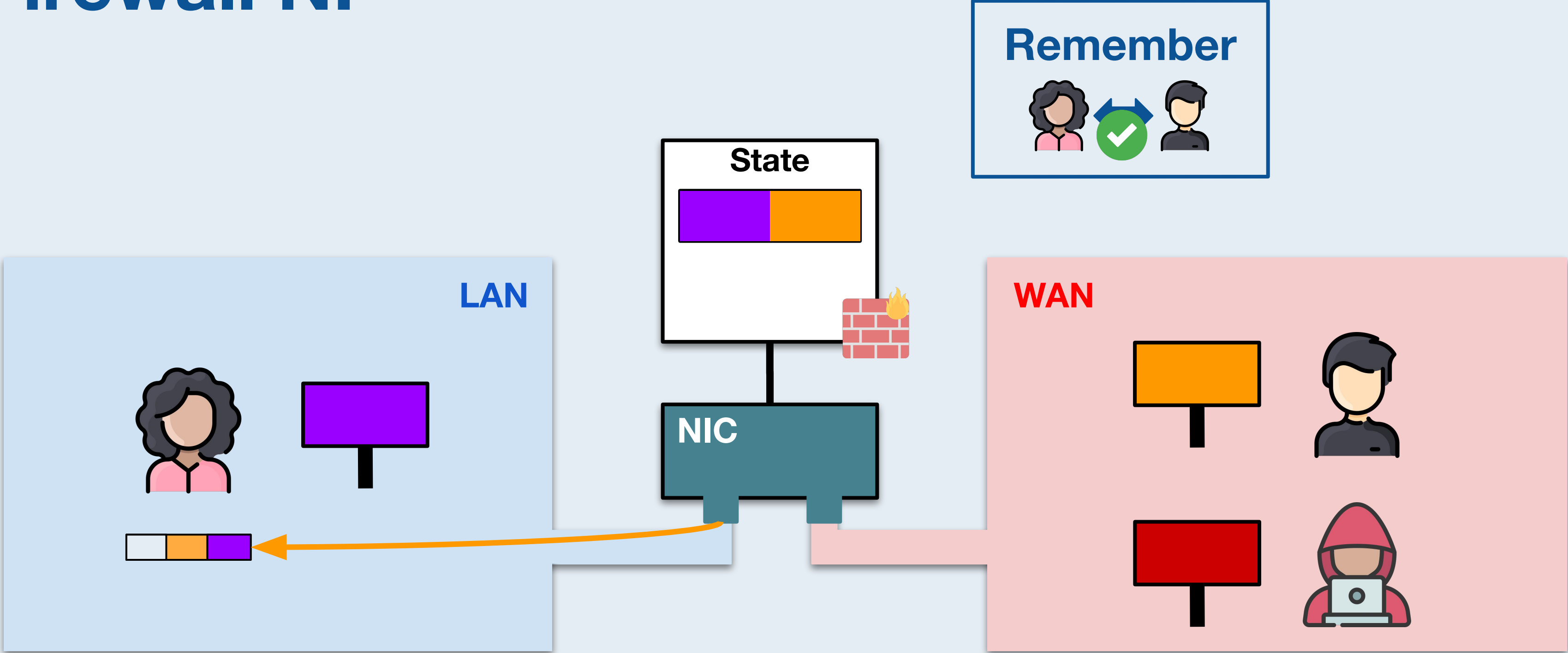
Firewall NF



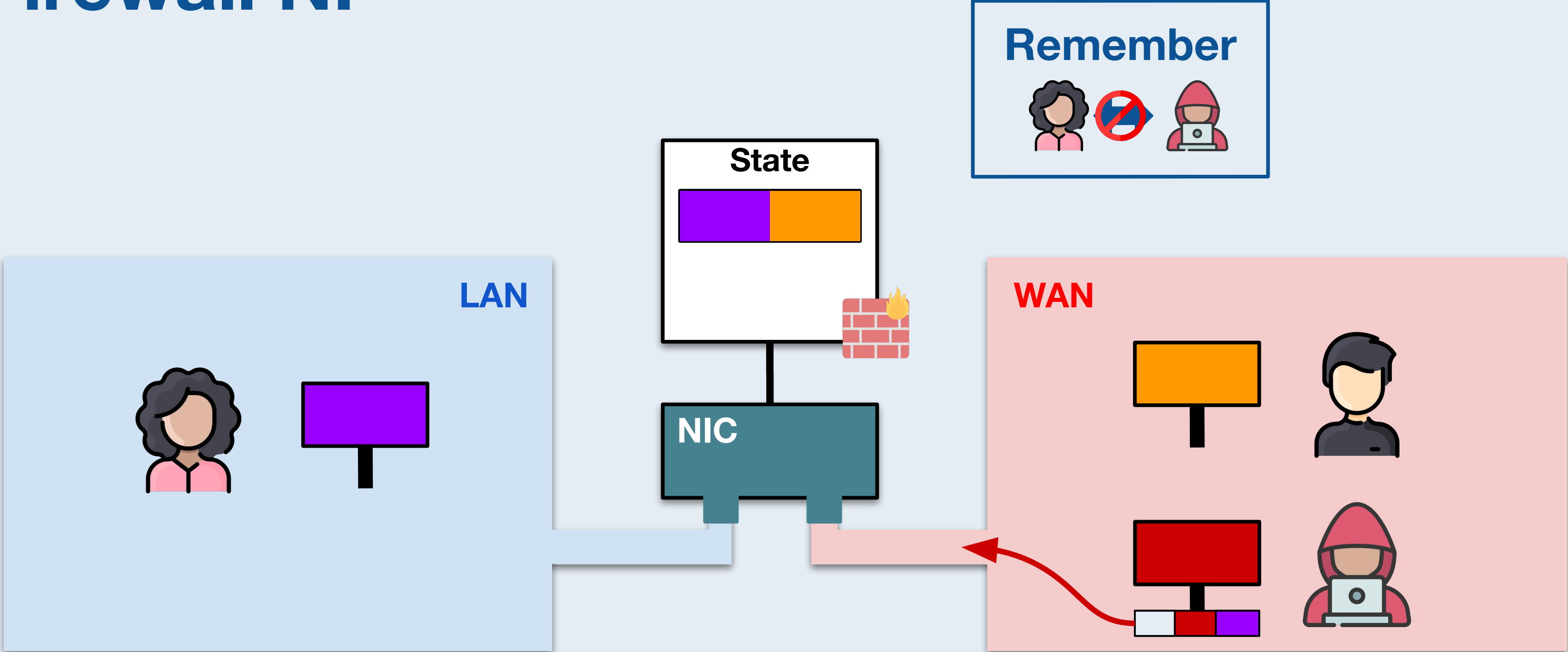
Firewall NF



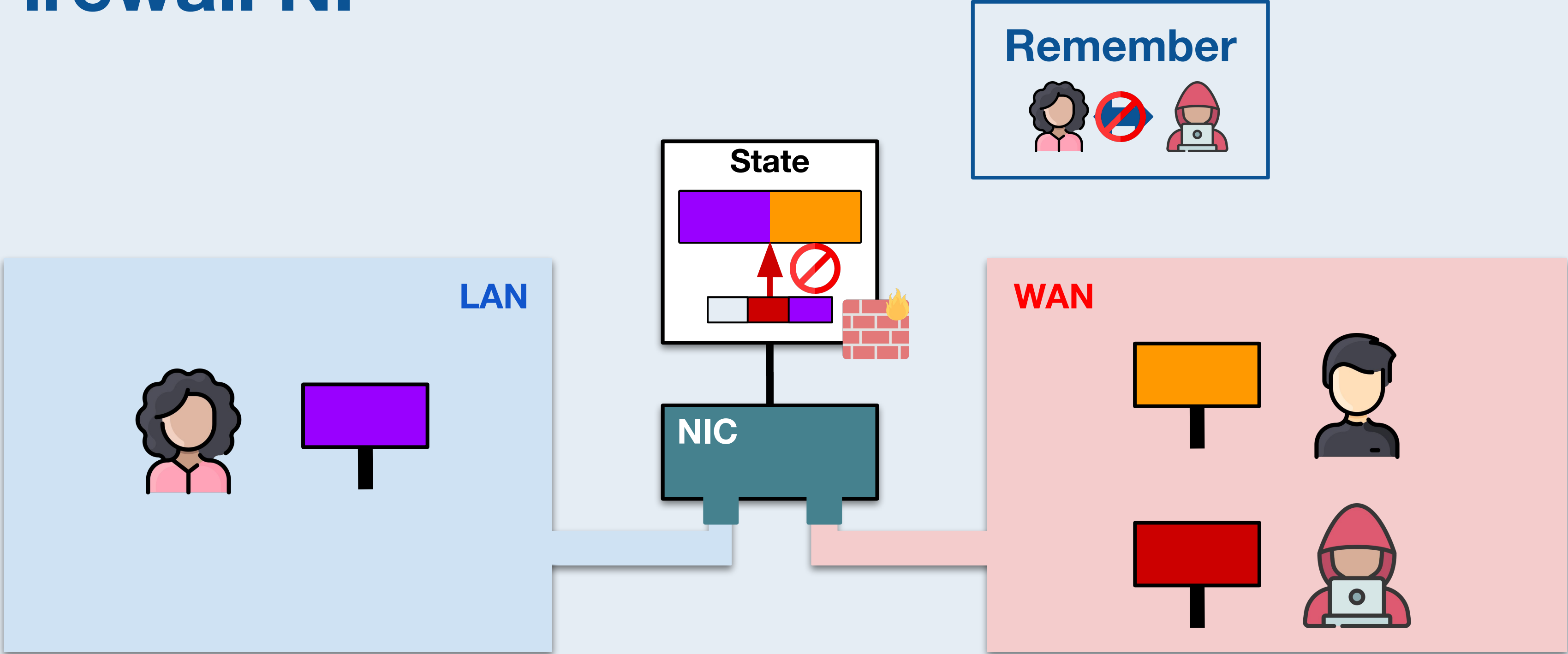
Firewall NF



Firewall NF



Firewall NF

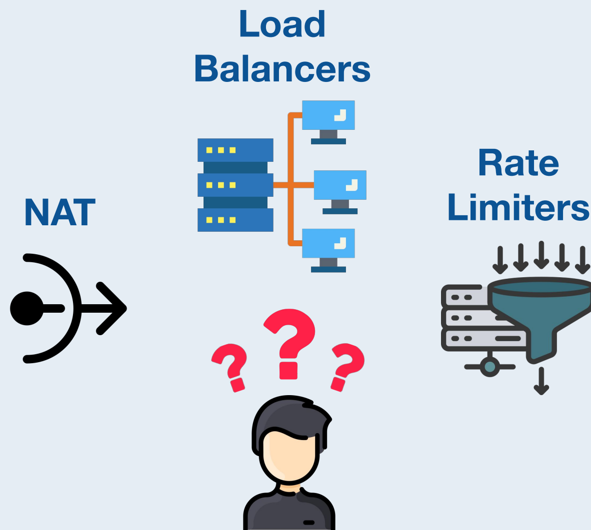


Why is parallelization hard?

Why is parallelization hard?

1 Finding the right sharding solution

Symmetry

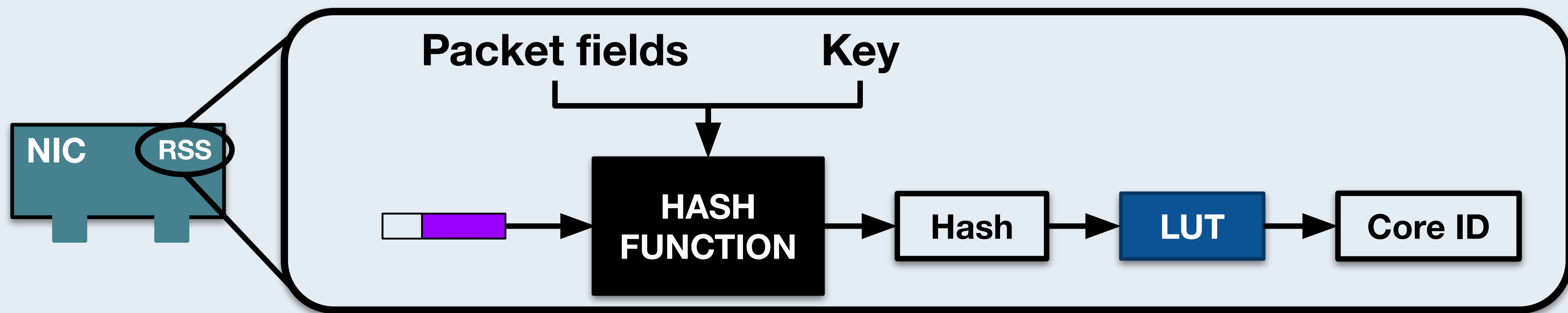


Why is parallelization hard?

1 Finding the right sharding solution

2 Finding the right NIC configuration

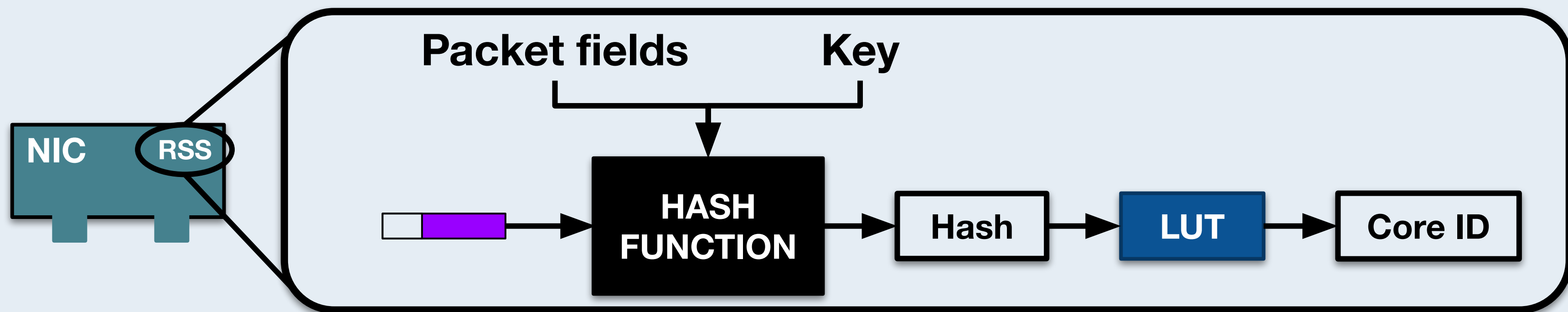
Why is parallelization hard?



Why is parallelization hard?

1 Finding the right sharding solution

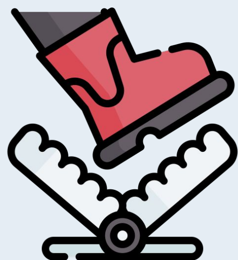
2 Finding the right NIC configuration



Which packet fields & key enforce the required sharding solution?

Why is parallelization hard?

1. Finding the right sharding solution
2. Finding the right NIC configuration
3. **Writing performant parallel code**



Why is parallelization hard?

1 Finding the right sharding solution

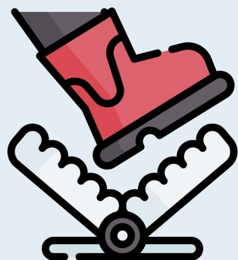
2 Finding the right NIC configuration

3 **Writing performant parallel code**

False sharing
Cache alignment

Concurrent memory accesses
Synchronization

Load imbalance
Load balancing



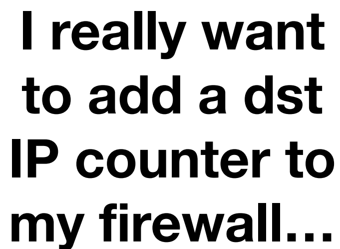
Slow remote memory accesses
NUMA awareness

Contention for locks
Avoiding locks

Why is parallelization hard?

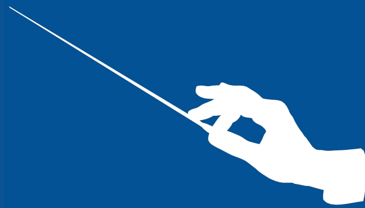
1. Finding the right sharding solution
2. Finding the right NIC configuration
3. Writing performant parallel code

I really want to add a dst IP counter to my firewall...



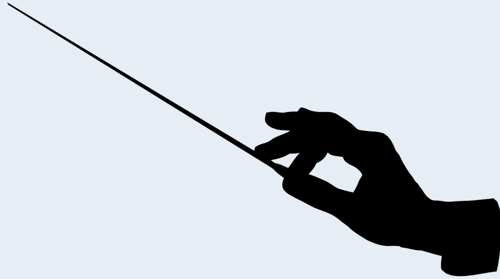
Repeat the whole process

Typical constraints found on NFs makes automatic parallelization possible



We propose Maestro, a solution for automatic parallelization

Automatic parallelization



Push-button parallelization

Favors shared-nothing architectures

Provides a highly-optimized lock-based alternative

Can also generate parallel implementations using hardware transactional memory (HTM)

The 3 ideas supporting Maestro

Static analysis

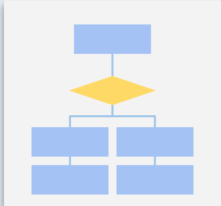
Infer state manipulation

SMT solver

Find the RSS configuration

Automate code generation

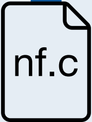
Maestro’s pipeline

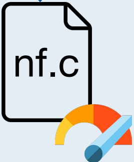


$$p_0[w] = p_1[x] \wedge p_0[y] = p_1[z]$$

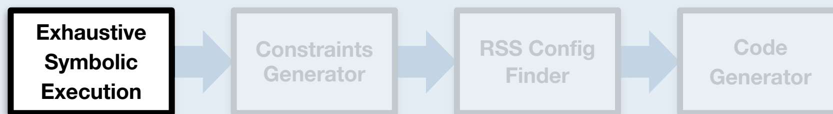
0x14	0x05	0x84	0x82	0x84	0x83	0x14	0x04
0x14	0x05	0x84	0x83	0x14	0x05	0x84	0x83
0x20	0x0d	0xc5	0x2e	0xfa	0xa6	0xe1	0xf6
0x28	0xe3	0x60	0xfb	0x20	0xfb	0x63	0x4a
0x78	0xd8	0x09	0x2b	0x70	0x94	0xf1	0x29
0xac	0x60	0xc0	0x64	0xb8	0xa2	0xff	0xd8

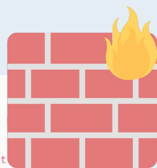
Exhaustive Symbolic Execution → **Constraints Generator** → **RSS Config Finder** → **Code Generator**





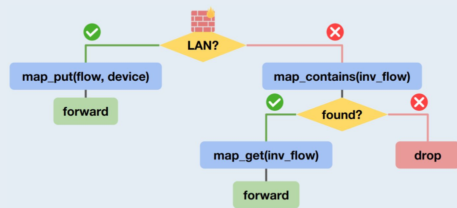
Extracting the NF model





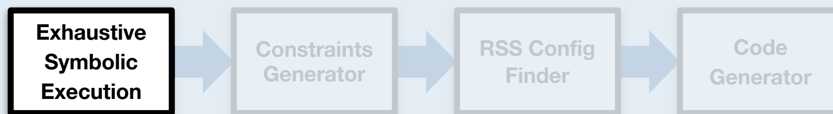
```
void process_packet(int device, pkt_t
  if (device == WAN) {
    struct Flow flow = {
      src_port: p.tcpudp_dst, dst_port
      src_ip:  p.ipv4_dst,    dst_ip:   p.ipv4_src
    };

    if (!map_contains(map, flow)) {
      drop(p);
      return;
    }

    int dst_device = map_get(map, flow);
    forward(p, dst_device);
  } else {
    struct Flow flow = {
      src_port: p.tcpudp_src, dst_port: p.tcpudp_dst,
      src_ip:  p.ipv4_src,    dst_ip:   p.ipv4_dst
    }

    map_put(map, flow, device);
    forward(p, WAN);
  }
}
```

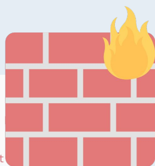
Exhaustive Symbolic Execution



Sound and complete model

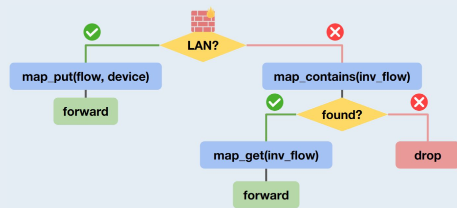
Extracting the NF model





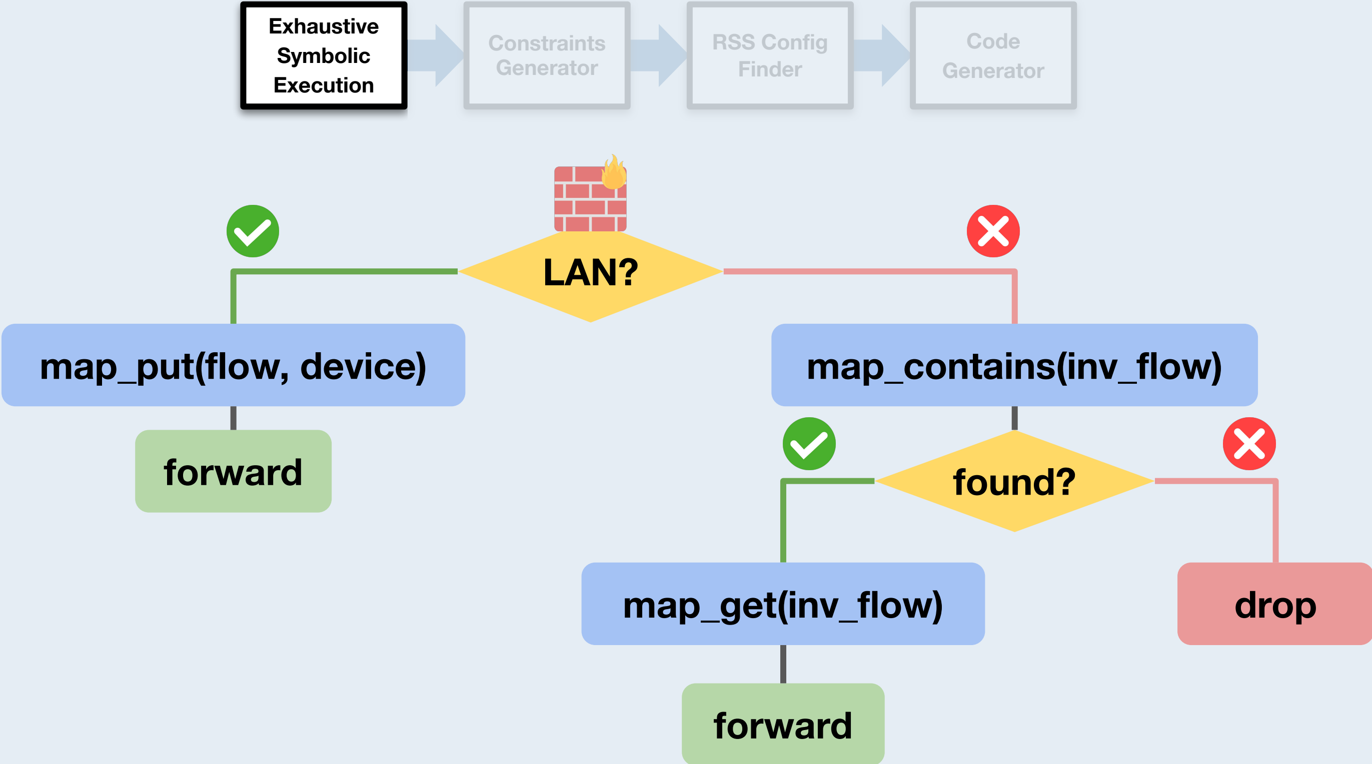
```
1 void process_packet(int device, pkt_t
2   if (device == WAN) {
3     struct Flow flow = {
4       src_port: p.tcpudp_dst, dst_port
5       src_ip:  p.ipv4_dst,    dst_ip:   p.ipv4_src
6     };
7
8     if (!map_contains(map, flow)) {
9       drop(p);
10      return;
11    }
12
13    int dst_device = map_get(map, flow);
14    forward(p, dst_device);
15  } else {
16    struct Flow flow = {
17      src_port: p.tcpudp_src, dst_port: p.tcpudp_dst,
18      src_ip:  p.ipv4_src,    dst_ip:   p.ipv4_dst
19    }
20
21    map_put(map, flow, device);
22    forward(p, WAN);
23  }
24 }
```

Exhaustive Symbolic Execution

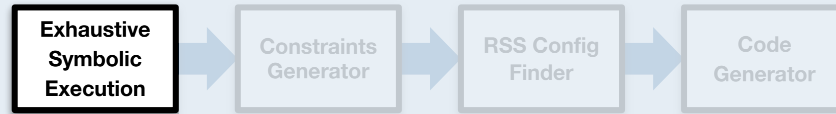


Sound and complete model

Extracting the NF model



Extracting the NF model



LAN?

map_put(flow, device)

forward

map_contains(inv_flow)

found?

map_get(inv_flow)

forward

drop

When is it safe to *concurrently* access a map?

Partitioning the map across cores

Partitioning the map across cores

- **R1** Key equality
- **R2** Subsumption
- **R3** Disjoint Dependencies
- **R4** Incompatible Dependencies
- **R5** Interchangeable Constraints

Partitioning the map across cores

R1 Key equality

R2 Subsumption

R3 Disjoint Dependencies

R4 Incompatible Dependencies

R5 Interchangeable Constraints

Dealing with hardware limitations

For more details check our paper

Partitioning the map across cores

R1 Key equality

R2 Subsumption

map_put(flow, v)

Same key

Same state

Partitioning the map across cores

R1 **Key equality**

R2 Subsumption

map_put(flow, v)

Same ~~key~~ flow

Same ~~state~~ core

Partitioning the map across cores

R1 **Key equality**

R2 Subsumption

map_put(flow, v)

Same ~~key~~ flow

Same ~~state~~ core

p_0 and p_1 are sent to the same core if

$p_0[flow] = p_1[flow]$

Partitioning the map across cores

R1 Key equality

R2 **Subsumption**

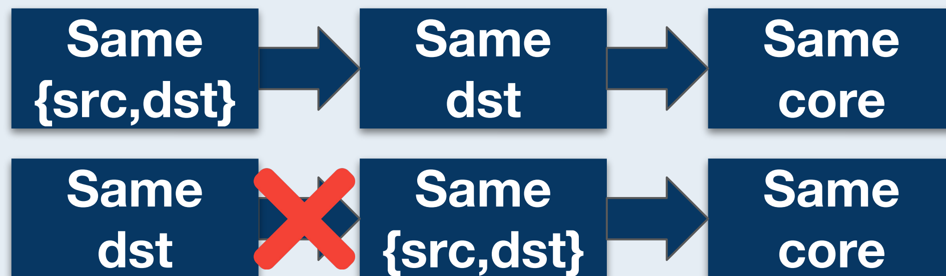
```
map_put({src_ip, dst_ip}, v)
```

```
map_put(dst_ip, v)
```

Partitioning the map across cores

R1 Key equality

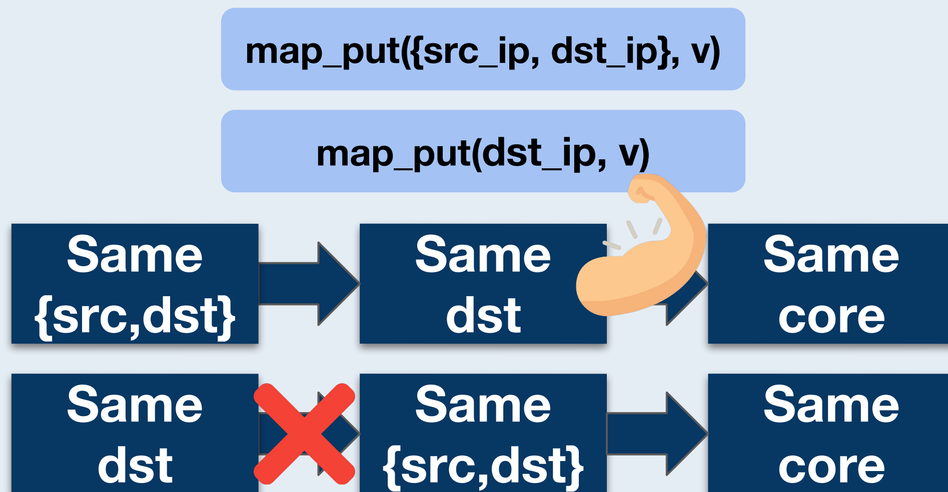
R2 **Subsumption**



Partitioning the map across cores

R1 Key equality

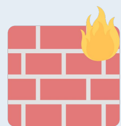
R2 **Subsumption**



p_0 and p_1 are sent to the same core if:

p_0[dst_ip] = p_1[dst_ip]

Finding the constraints for the firewall

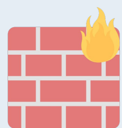


map_put(flow, device) LAN

map_contains(inv_flow) WAN

map_get(inv_flow) WAN

Finding the constraints for the firewall



map_put(flow, device) LAN

map_contains(inv_flow) WAN

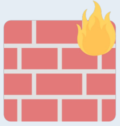
map_get(inv_flow) WAN

p_0 (LAN) and p_1 (LAN) are sent to the same core if

$p_0[flow] = p_1[flow]$

R1 Key equality

Finding the constraints for the firewall



LAN
map_put(flow, device)

WAN
map_contains(inv_flow)

WAN
map_get(inv_flow)

p_0 (LAN) and p_1 (LAN) are sent to the same core if

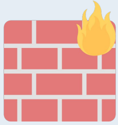
$p_0[flow] = p_1[flow]$

p_0 (WAN) and p_1 (WAN) are sent to the same core if

$p_0[inv_flow] = p_1[inv_flow]$

R1 Key equality

Finding the constraints for the firewall



map_put(flow, device) LAN

map_contains(inv_flow) WAN

map_get(inv_flow) WAN

R1 Key equality

p_0 (LAN) and p_1 (LAN) are sent to the same core if

p_0[flow] = p_1[flow]

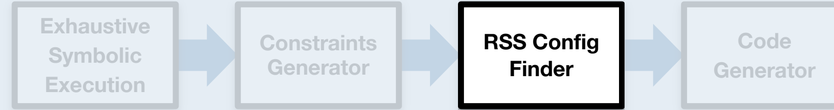
p_0 (WAN) and p_1 (WAN) are sent to the same core if

p_0[inv_flow] = p_1[inv_flow]

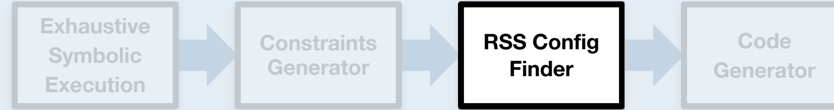
p_0 (LAN) and p_1 (WAN) are sent to the same core if

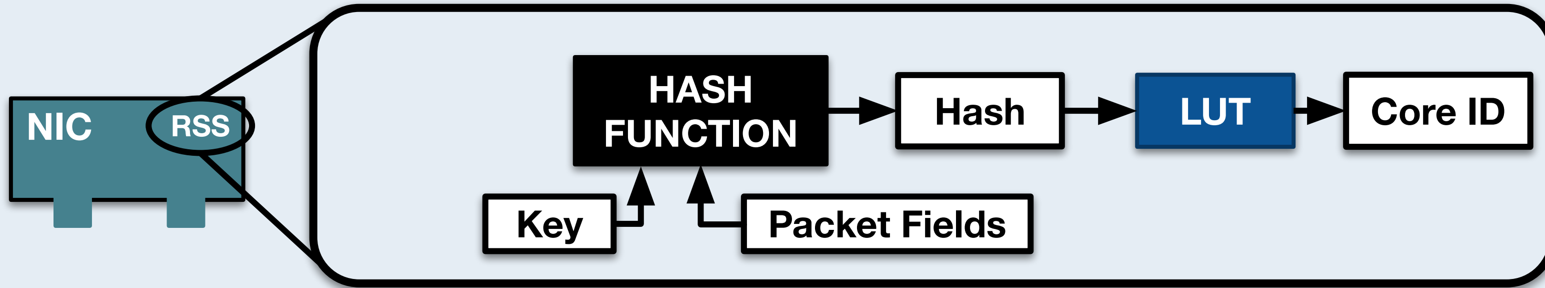
p_0[flow] = p_1[inv_flow]

Finding the RSS configuration

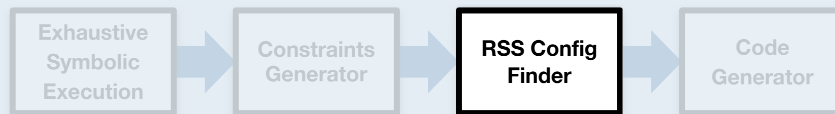


Finding the RSS configuration



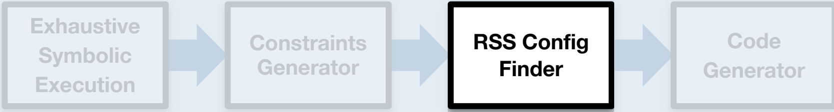


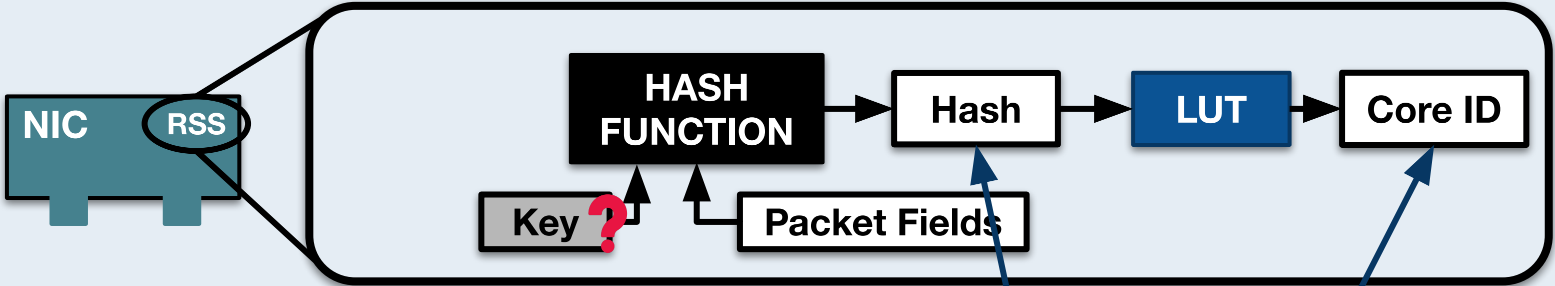
Finding the RSS configuration

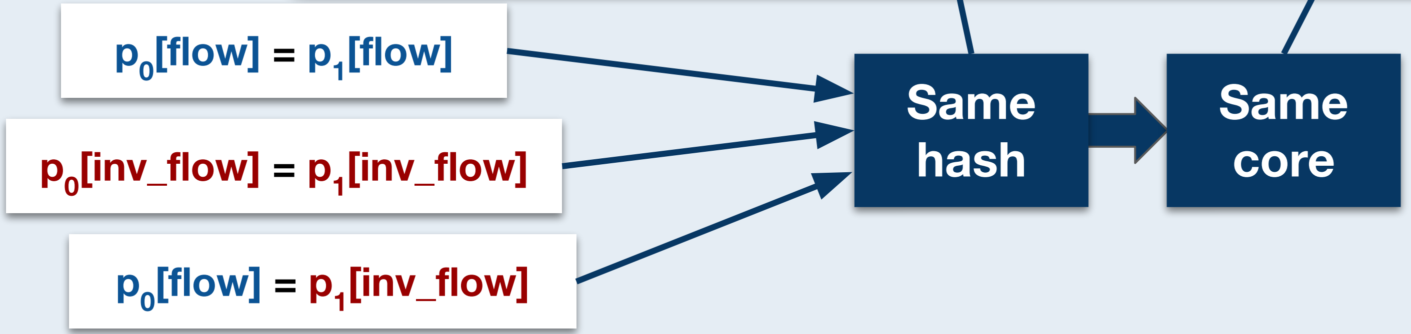


NIC
RSS
HASH FUNCTION
Hash
LUT
Core ID
Key
Packet Fields
Same hash
Same core

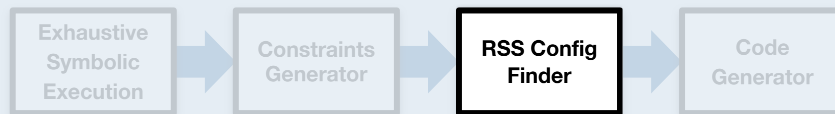
Finding the RSS configuration





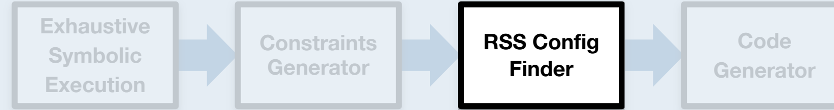


Finding the RSS configuration

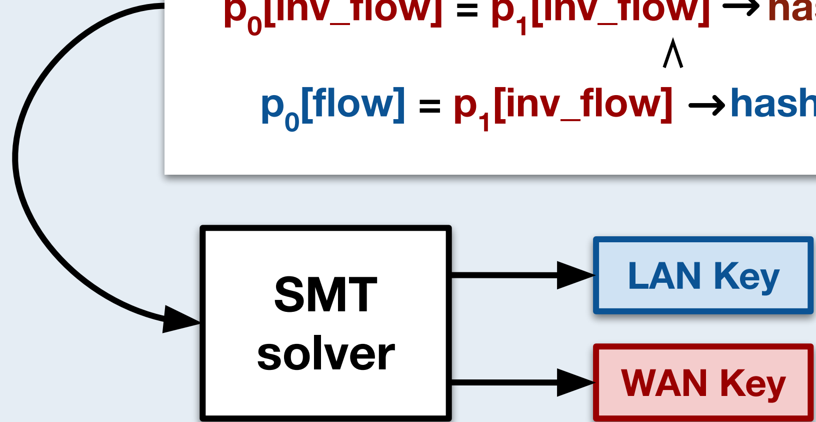


$$p_0[flow] = p_1[flow] \rightarrow hash(p_0) = hash(p_1)$$
$$\wedge$$
$$p_0[inv_flow] = p_1[inv_flow] \rightarrow hash(p_0) = hash(p_1)$$
$$\wedge$$
$$p_0[flow] = p_1[inv_flow] \rightarrow hash(p_0) = hash(p_1)$$

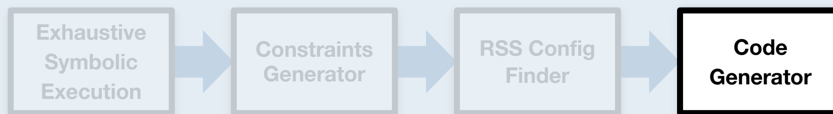
Finding the RSS configuration

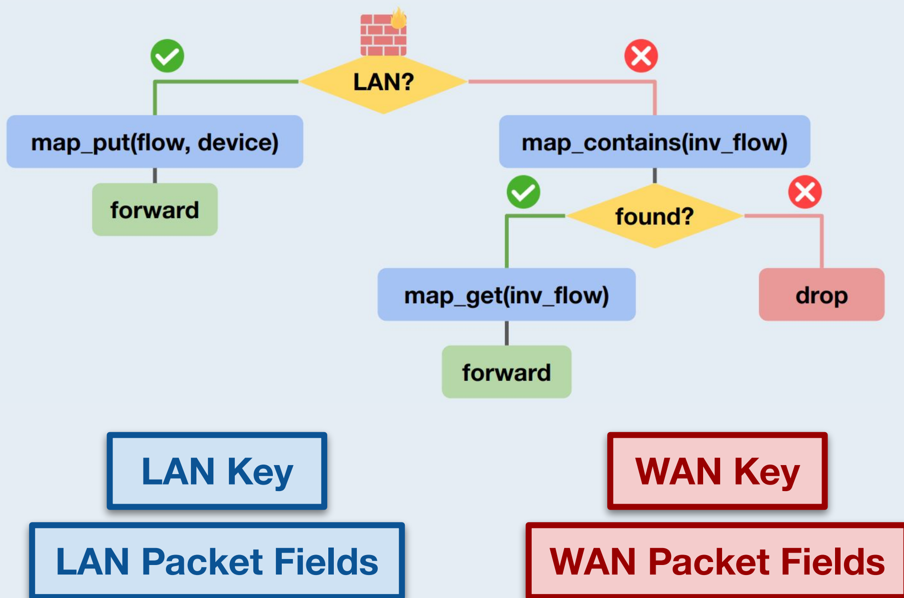


$$p_0[flow] = p_1[flow] \rightarrow hash(p_0) = hash(p_1)$$
$$\wedge$$
$$p_0[inv_flow] = p_1[inv_flow] \rightarrow hash(p_0) = hash(p_1)$$
$$\wedge$$
$$p_0[flow] = p_1[inv_flow] \rightarrow hash(p_0) = hash(p_1)$$



Code generator

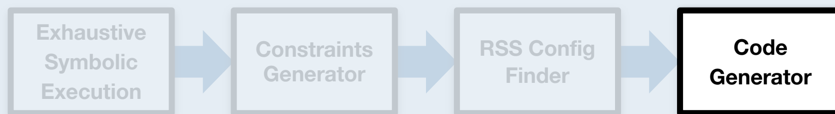


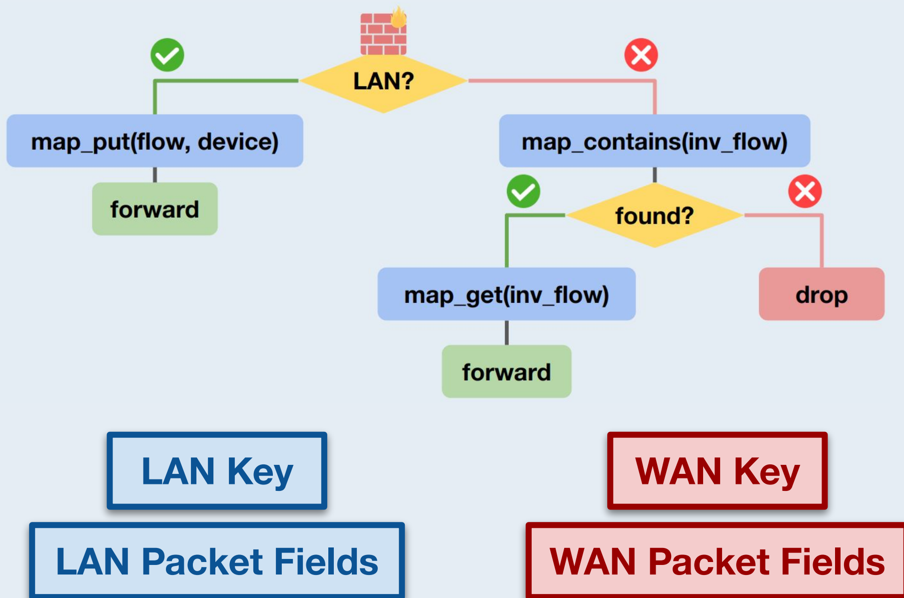


O(minutes)

```c
1 // One map for each thread.
2 struct Map** flows;
3
4 #define LAN 0
5 #define WAN 1
6
7 uint8_t RSS_HASH_PORT_0[52] = {
8   0xa1, 0x24, 0x00, 0x15, 0x00, 0x14, 0xa1, 0x24,
9   0xa1, 0x24, 0x00, 0x14, 0xa1, 0x24, 0x00, 0x15,
10  0xa7, 0xfa, 0x11, 0x22, 0x6f, 0xd3, 0xf0, 0x42,
11  0x1b, 0x6c, 0xeb, 0x14, 0x62, 0x02, 0xa3, 0x44,
12  0x24, 0x90, 0xf8, 0x1c, 0x43, 0x99, 0xe7, 0xaf,
13  0x80, 0x73, 0x15, 0xfe, 0x29, 0x5a, 0x73, 0xd0,
14  0x55, 0x85, 0xf2, 0xc4
15 };
16
17 //[...]
18
19 // Run by each worker thread.
20 int init() {
21   unsigned core_id = rte_lcore_id();
22   if (core_id == rte_get_main_lcore()) {
23     rss_configure(
24       LAN, RSS_HASH_PORT_0, IP_TCP | IP_UDP);
25     rss_configure(
26       WAN, RSS_HASH_PORT_1, IP_TCP | IP_UDP);
27   }
28   //[...]
29 }
30
31 // Run by each packet on a specific worker thread.
32 void process(int port, pkt_t pkt) {
33   unsigned core_id = rte_lcore_id();
34
35   //[...]
36 }
37
```

Code generator





O(minutes)

Read-write lock-based solution if shared-nothing is deemed infeasible.

```
21   unsigned core_id = rte_lcore_id();
22   if (core_id == rte_get_main_lcore()) {
23     rss_configure(
24
36 }
37
```

For more details check our paper

Evaluation

- **How does performance scale with the number of cores**
  - Shared nothing vs Lock-based vs HTM
  - Varying traffic patterns
  - Packet size
  - Churn
- **How does it fare against other parallel frameworks?**
  - Vector Packet Processing (VPP)

Evaluation

- **How does performance scale with the number of cores**
  - Shared nothing vs Lock-based vs HTM
  - Varying traffic patterns
  - Packet size
  - Churn
- **How does it fare against other parallel frameworks?**
  - Vector Packet Processing (VPP)

For more details check our paper

Scalability

Shared-Nothing

Locks

HTM

Scalability

Shared-Nothing

Locks

HTM

Shared-nothing Lock-based TM

Throughput (Millions of packets/second)

NOP
SBridge
DBridge
Policer
FW
NAT
CL
PSD
LB

0 20 40 60 80

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

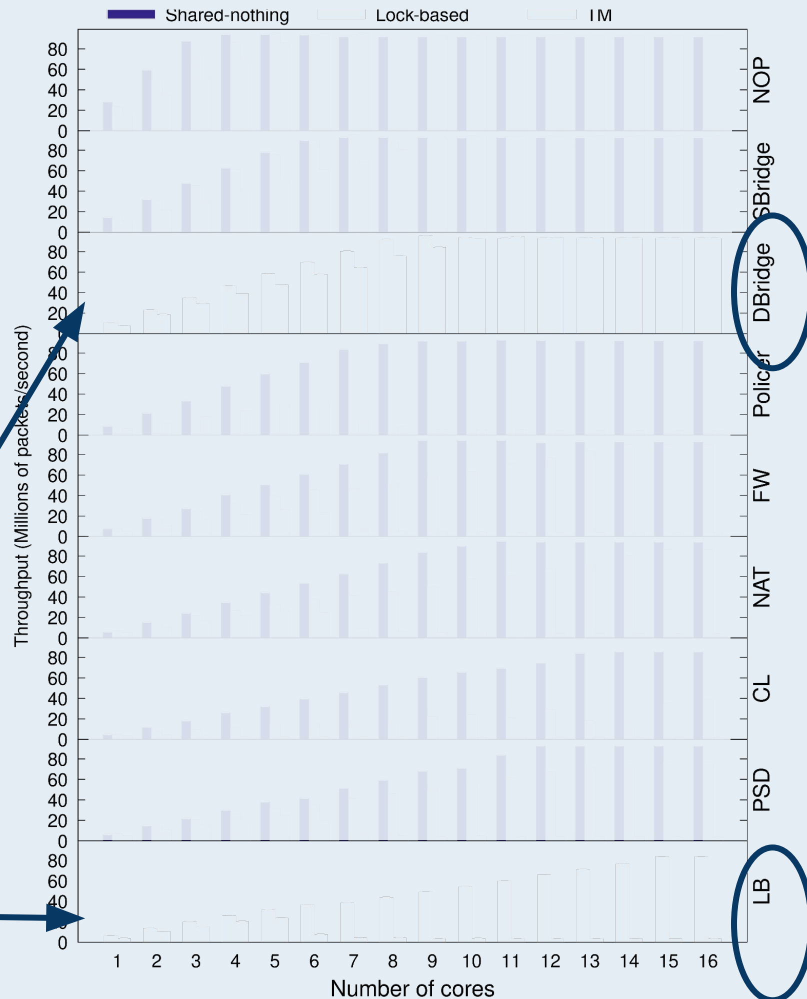
Number of cores

Scalability

Shared-Nothing

Locks

HTM



No shared-nothing solution

Scalability

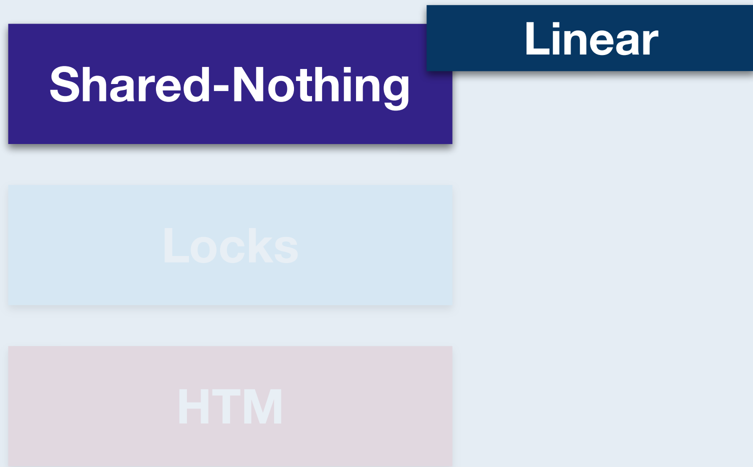
Shared-Nothing

Locks

HTM

Bottlenecked by the PCIe

Scalability



Shared-nothing Lock-based TM

NOP
SBridge
DBridge
Policer
FW
NAT
CL
PSD
LB

Throughput (Millions of packets/second)

0 20 40 60 80

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Number of cores

Scalability

Shared-Nothing

Linear

Locks

HTM

Shared-nothing Lock-based TM

NOP
SBridge
DBridge
Policer
FW
NAT
CL
PSD
LB

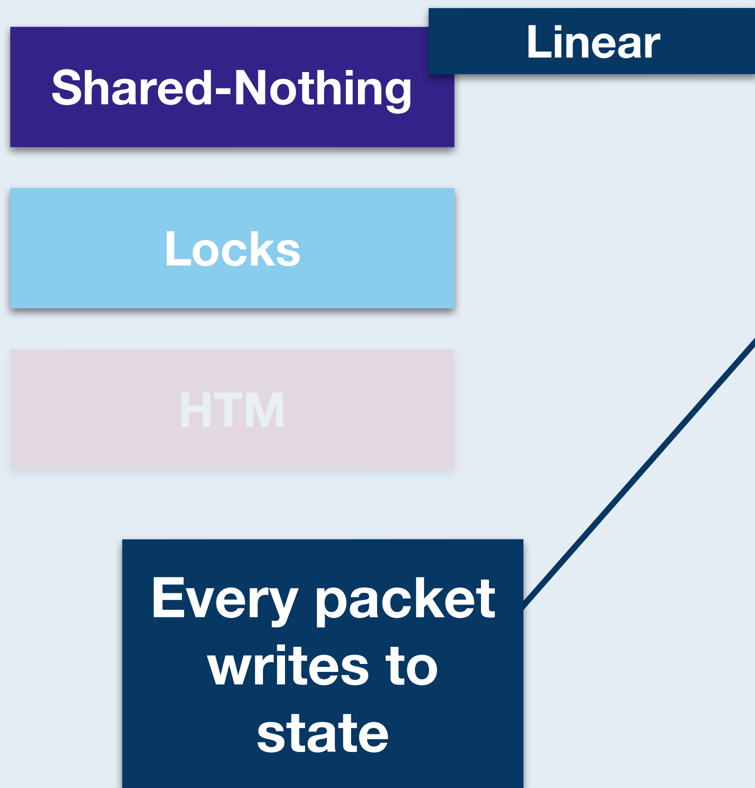
Throughput (Millions of packets/second)

0 20 40 60 80

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Number of cores

Scalability



Shared-nothing Lock-based TM

NOP SBridge DBridge Policer FW NAT CL PSD LB

Throughput (Millions of packets/second)

0 20 40 60 80

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Number of cores

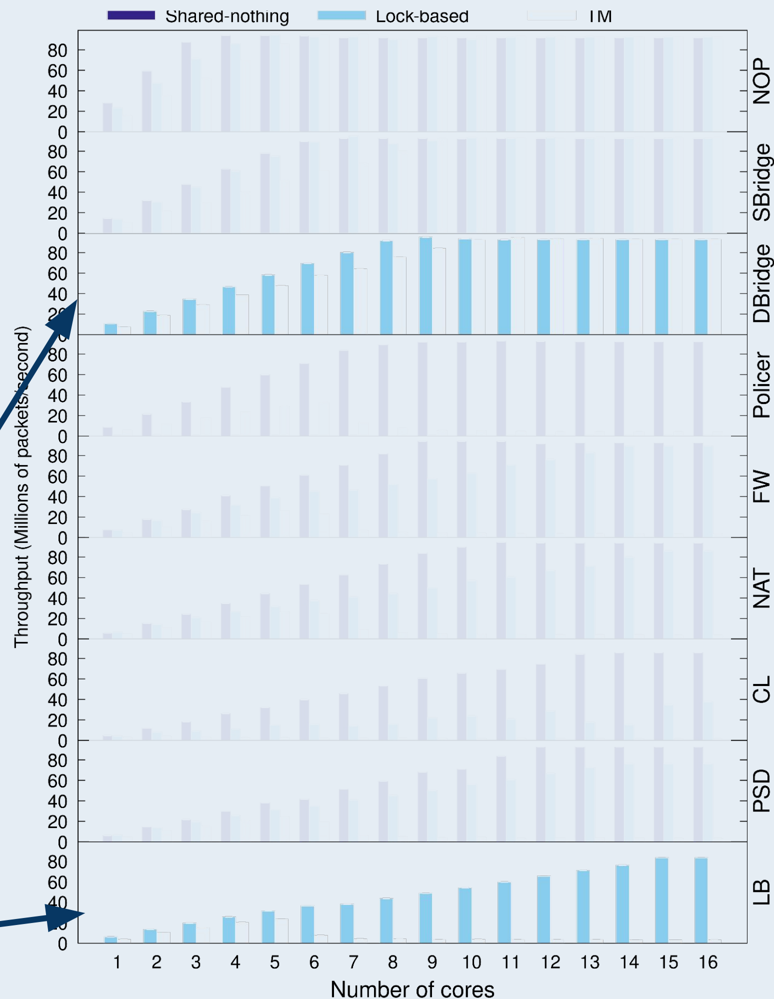
Scalability

Shared-Nothing

Linear

Locks

HTM



An alternative to shared-nothing

Scalability

Shared-Nothing — Linear

Locks — Linear / Less scalable

HTM

Shared-nothing | Lock-based | TM

Throughput (Millions of packets/second)

NOP
SBridge
DBridge
Policer
FW
NAT
CL
PSD
LB

0 20 40 60 80

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Number of cores

Scalability

Shared-Nothing — Linear

Locks — Linear / Less scalable

HTM

Shared-nothing Lock-based TM

Throughput (Millions of packets/second)

NOP
SBridge
DBridge
Policer
FW
NAT
CL
PSD
LB

Number of cores

Scalability

Shared-Nothing

Linear

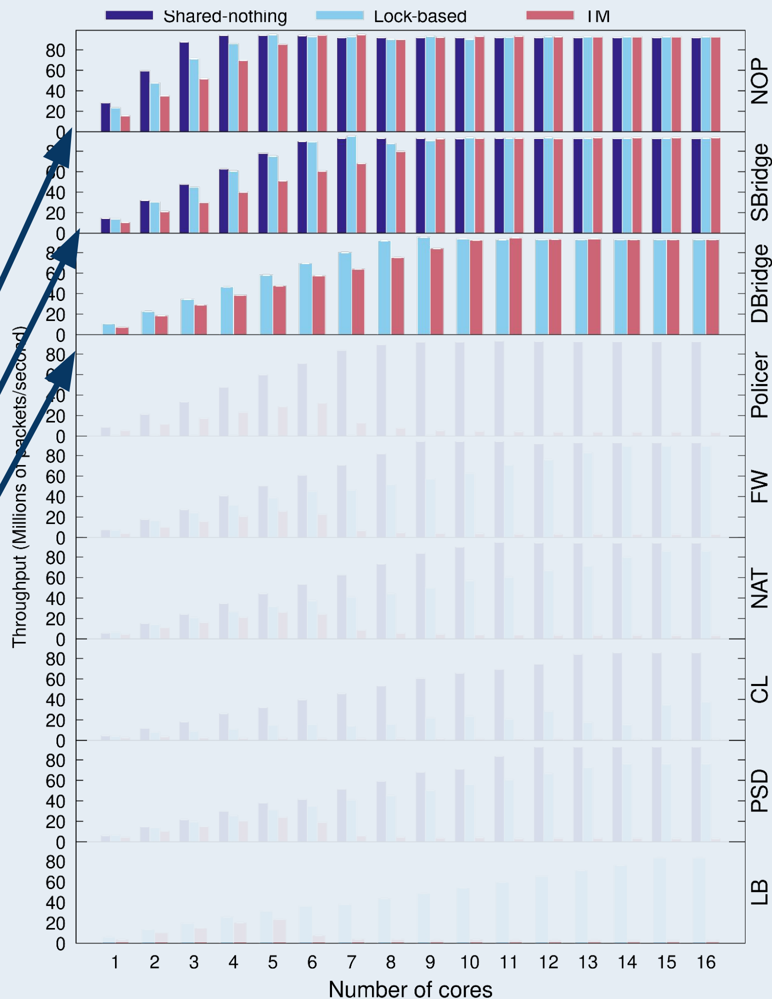
Locks

Linear

Less scalable

HTM

Best-case is comparable to locks



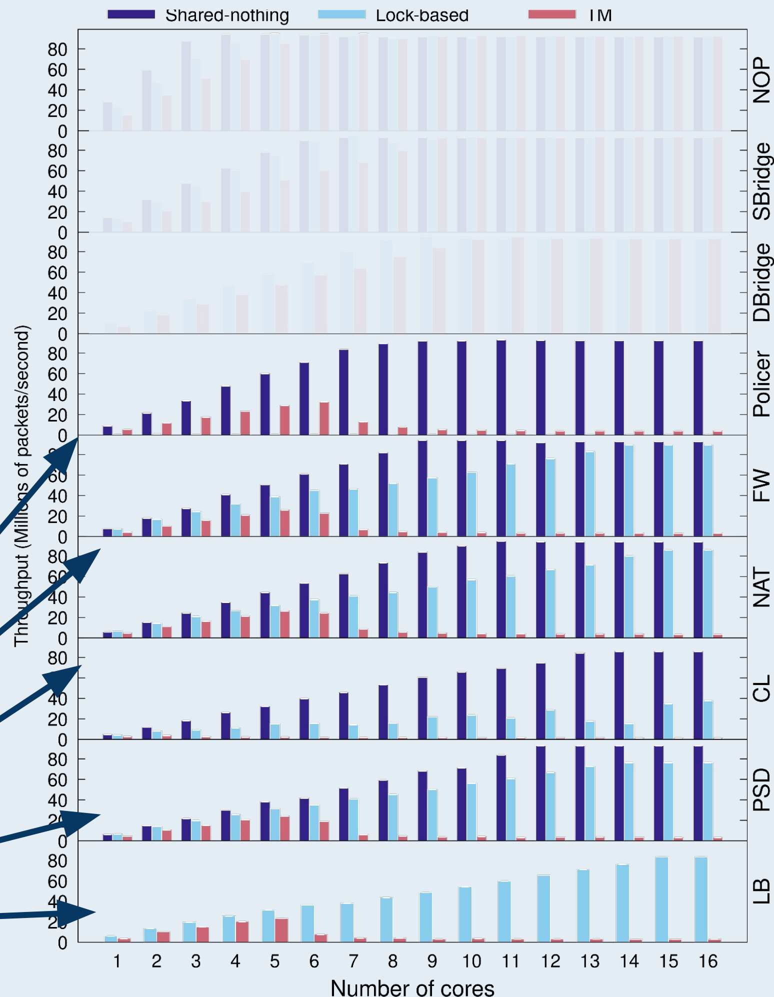
Scalability

Shared-Nothing — Linear

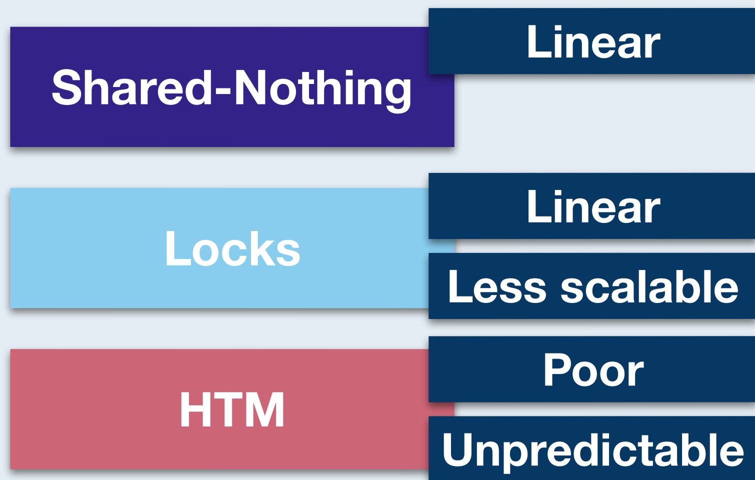
Locks — Linear / Less scalable

HTM

Otherwise is unpredictable or worse than locks



Scalability



Shared-nothing Lock-based TM

NOP SBridge DBridge Policer FW NAT CL PSD LB

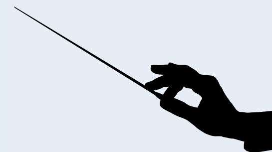
Throughput (Millions of packets/second)

0 20 40 60 80

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Number of cores

Conclusion



Maestro is a push-to-parallelize system that automatically parallelizes **software NFs.**

Generates **shared-nothing** parallel solutions whenever possible, and **lock-based** solutions otherwise.

Maestro's shared-nothing NFs scale **linearly** with cores.

Contact: francisco.chamica.pereira@tecnico.ulisboa.pt

Web: maestro.inesc-id.pt