UN*i*S: A User-space Non-intrusive Workflow-aware Virtual Network Function Scheduler

Anthony[1], Shihabur Rahman Chowdhury[1], Tim Bai[1], Raouf Boutaba[1], Jerome François[2]

University of Waterloo[1]
INRIA Nancy Grand Est[2]

CNSM 2018, Rome, Italy

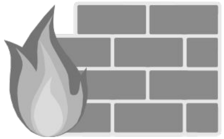
UNIVERSITY OF WATERLOO

inventors for the digital world

The Development of Network Function

Hardware middleboxes

Deep Packet
Inspection

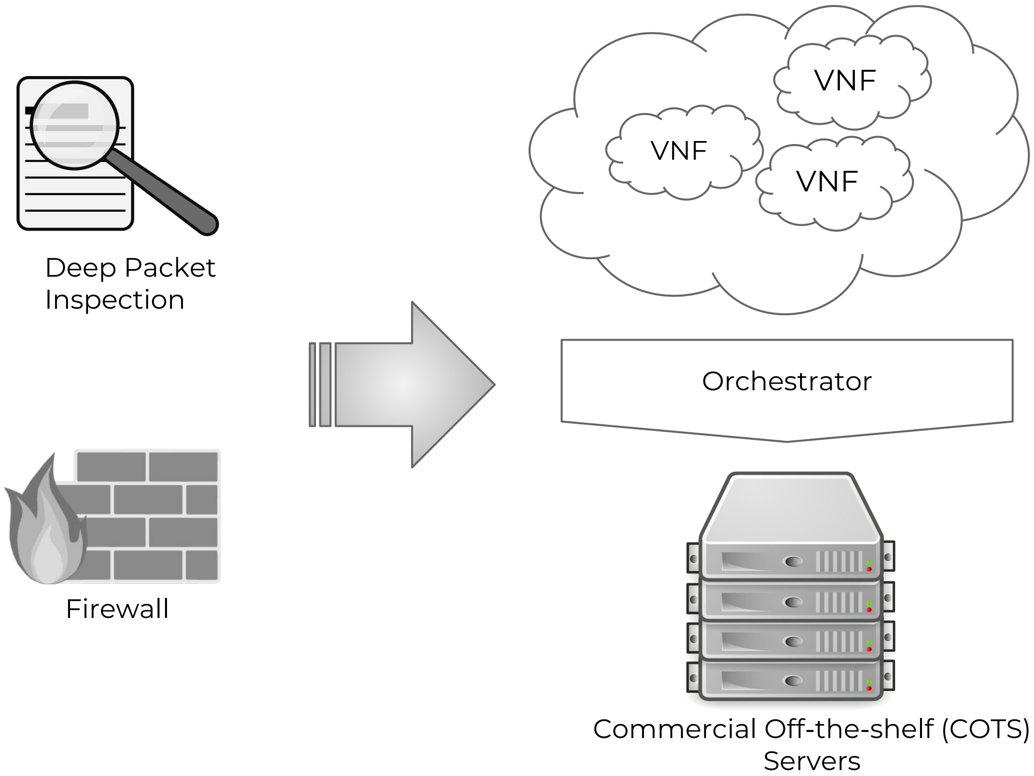


Firewall

The Development of Network Function

Hardware middleboxes

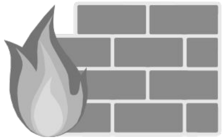
Virtual Network Function



The Development of Network Function

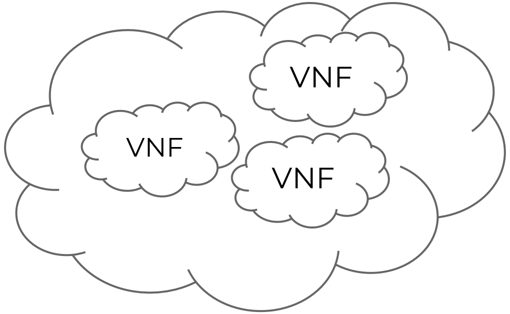
Hardware middleboxes

Deep Packet Inspection



Firewall

Virtual Network Function



Orchestrator

Commercial Off-the-shelf (COTS) Servers

Performance

Common Practices

Kernel bypass technologies (DPDK, Netmap, Solarflare, etc)

\+

CPU core pinning

\+

Poll mode

The Problems

1. Poll-mode
   → Inefficient resource utilization

The Problems

1. Poll-mode
   → Inefficient resource utilization

2. Core Pinning
   → Limited number of cores

The Problems

1. Poll-mode
   → Inefficient resource utilization

2. Core Pinning
   → Limited number of cores

Can we just put more VNFs on a single core?

The Problems

1. Poll-mode
   → Inefficient resource utilization

2. Core Pinning
   → Limited number of cores

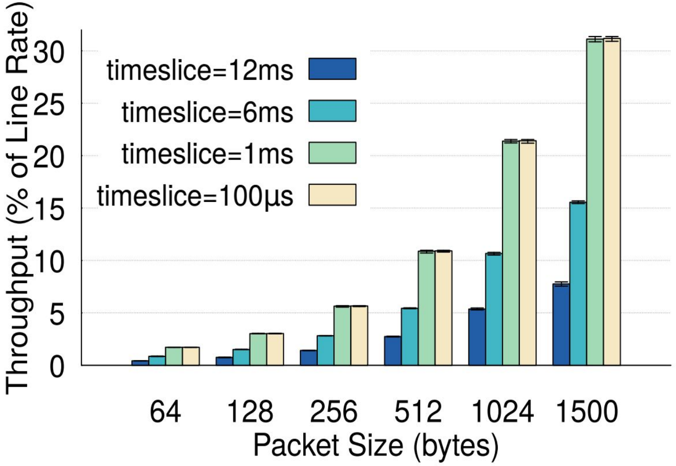
3. Inadequate Linux schedulers

The Problems

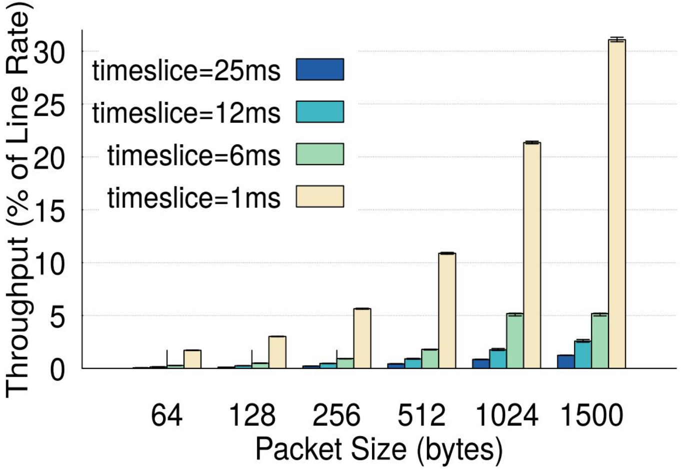
Default Linux schedulers

- Completely Fair Scheduler (CFS)
- Real Time scheduler (RT)

Setup: 2 lightweight VNFs, 10Gbps NIC, …



(a) CFS



(b) RT Scheduler

The state-of-the-art

1. *Flurries*
   Poll mode + interrupt

2. *NFV-Nice*
   Flurries + back pressure

1. W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, “Flurries: Countless fine-grained nfs for flexible per-flow customization,” in Proceedings of ACM CoNeXT. ACM, 2016, pp. 3–17.
2. S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, “NFVnice: Dynamic backpressure and scheduling for nfv service chains,” in Proceedings of ACM SIGCOMM. ACM, 2017, pp. 71–84.

The state-of-the-art

1. *Flurries*
   Poll mode + interrupt

2. *NFV-Nice*
   Flurries + back pressure

Another problem: *Intrusive*
Require VNF to use or be built with a certain library.

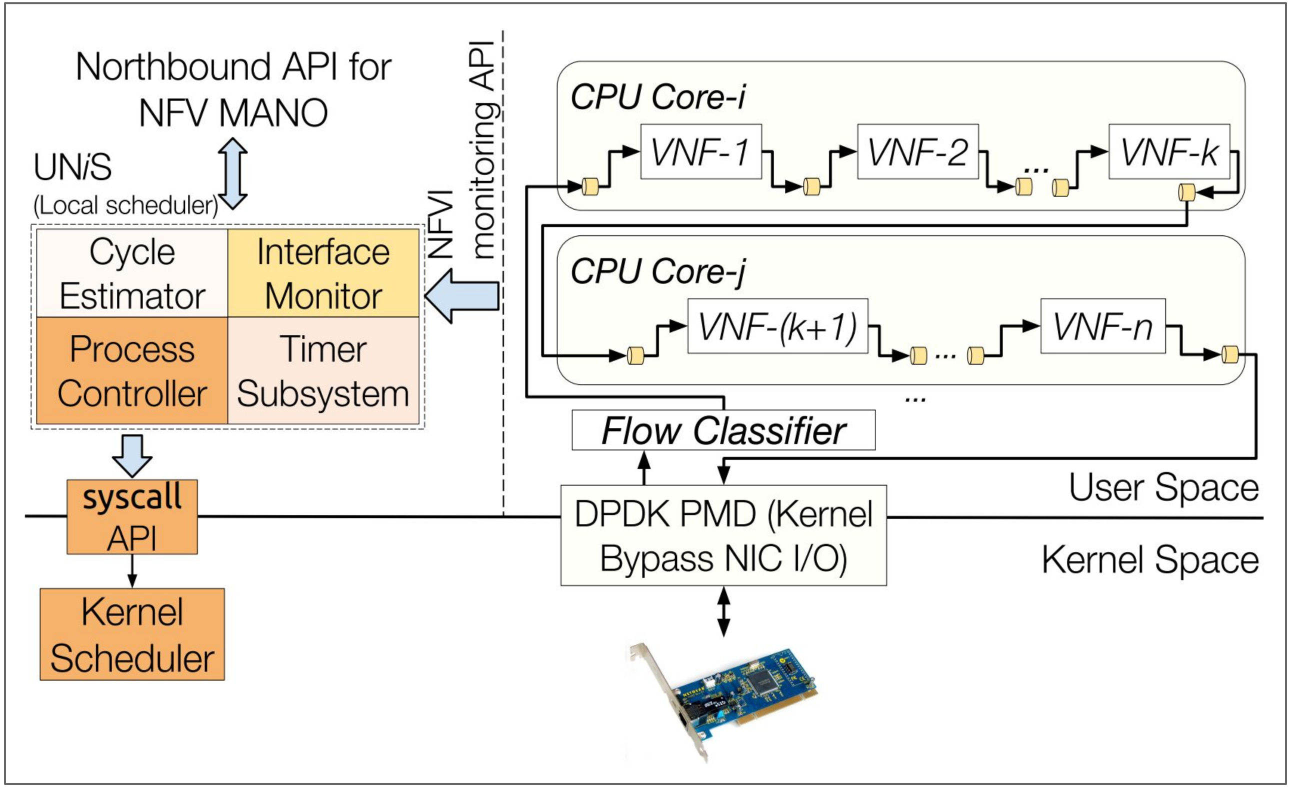
1. W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, “Flurries: Countless fine-grained nfs for flexible per-flow customization,” in Proceedings of ACM CoNeXT. ACM, 2016, pp. 3–17.
2. S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, “NFVnice: Dynamic backpressure and scheduling for nfv service chains,” in Proceedings of ACM SIGCOMM. ACM, 2017, pp. 71–84.

UN*i*S

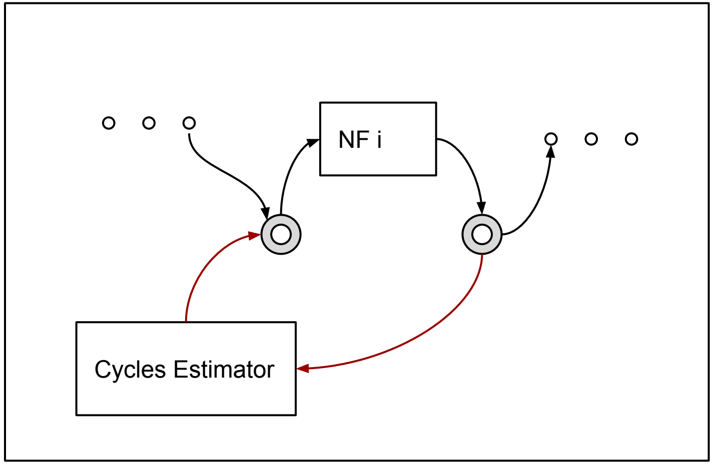
A User-space Non-intrusive Workflow-aware

Virtual Network Function Scheduler

System Architecture



Cycle Estimator



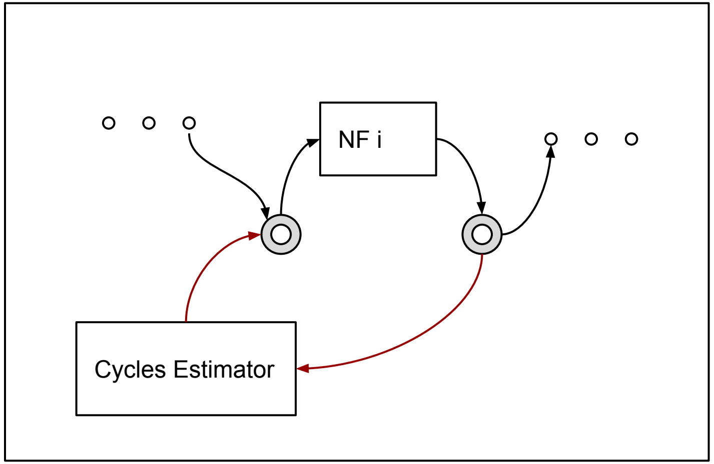
Goal

- Estimate the processing cost of a VNF

Implementation

- A static offline profiler
- Run NF-i in an isolated environment
- Inject a batch of packets
- Pull the batch and calculate the timestamp difference

Cycle Estimator



Goal

- Estimate the processing cost of a VNF

Implementation

- A static offline profiler
- Run NF-i in an isolated environment
- Inject a batch of packets
- Pull the batch and calculate the timestamp difference

UNiS introduces buffer occupancy based optimization to deal with variable cost VNF.

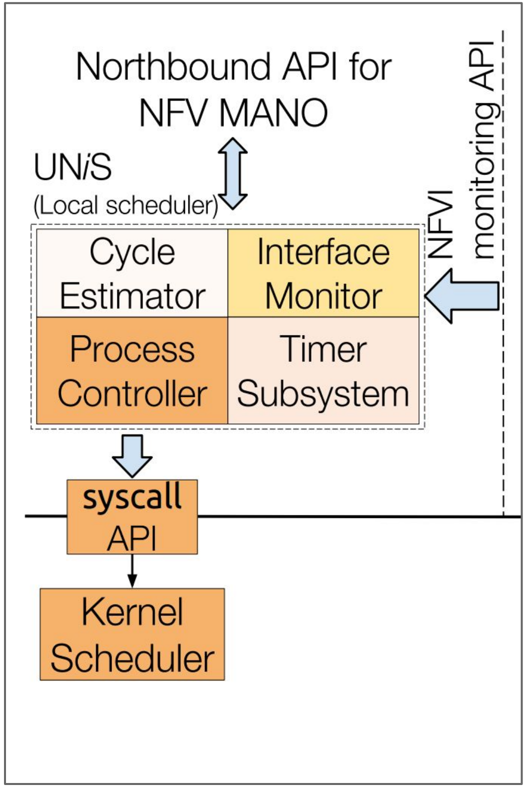
Timer Subsystem

Goal

- Keep track of time used by a VNF

Implementation

- DPDK *rte_timer* library
  - Async callbacks
  - Support high precision



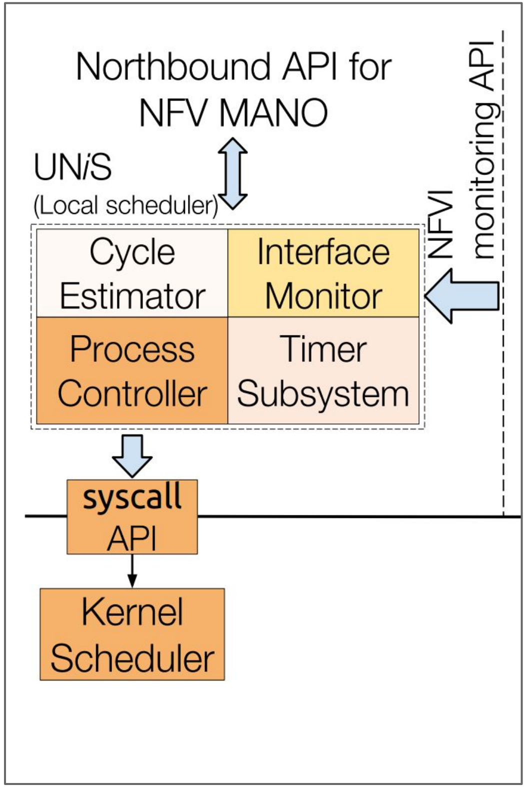
Process Controller

Goal

- Put a process to waiting/running state

Implementation

- Linux Real Time scheduler class (*SCHED_RR*)
- Adjust the *sched_priority* parameter



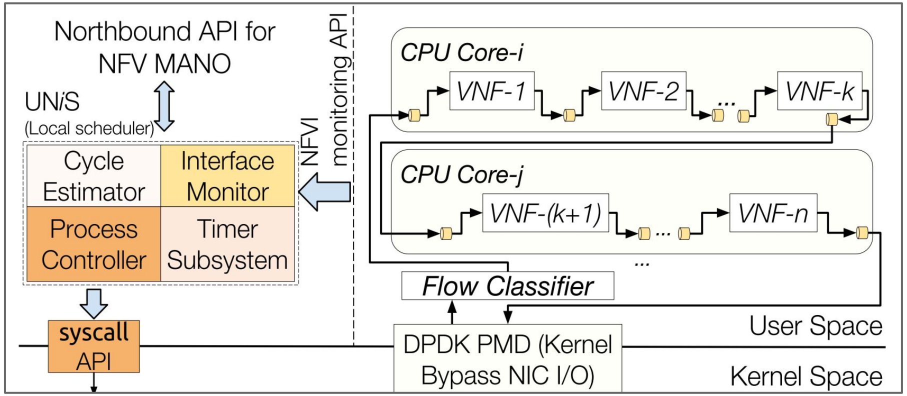
Interface Monitor

Goal

- Provide buffer occupancy monitoring data

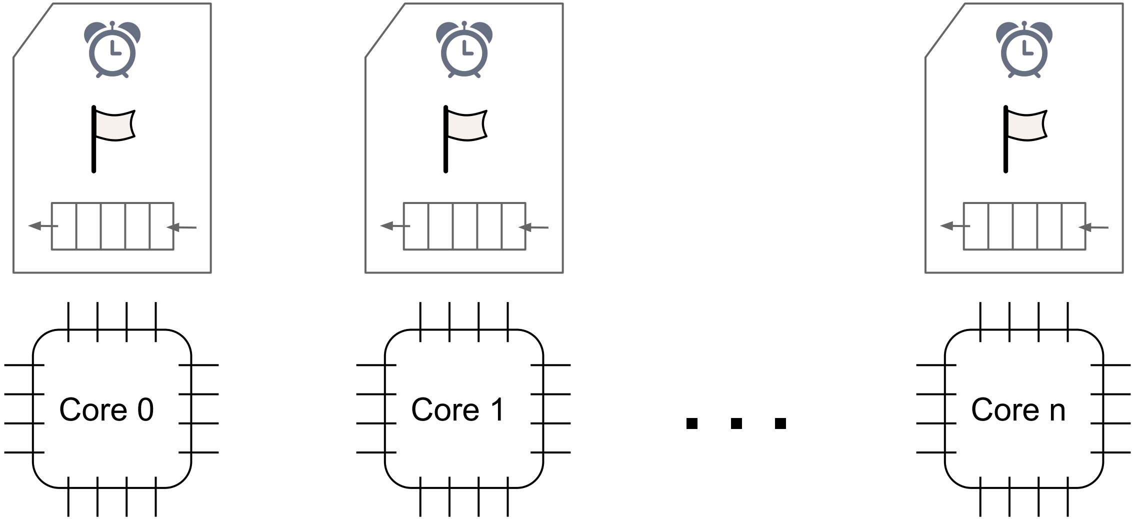
Implementation

- DPDK *rte_ring* library
  - zero copy packet transfer

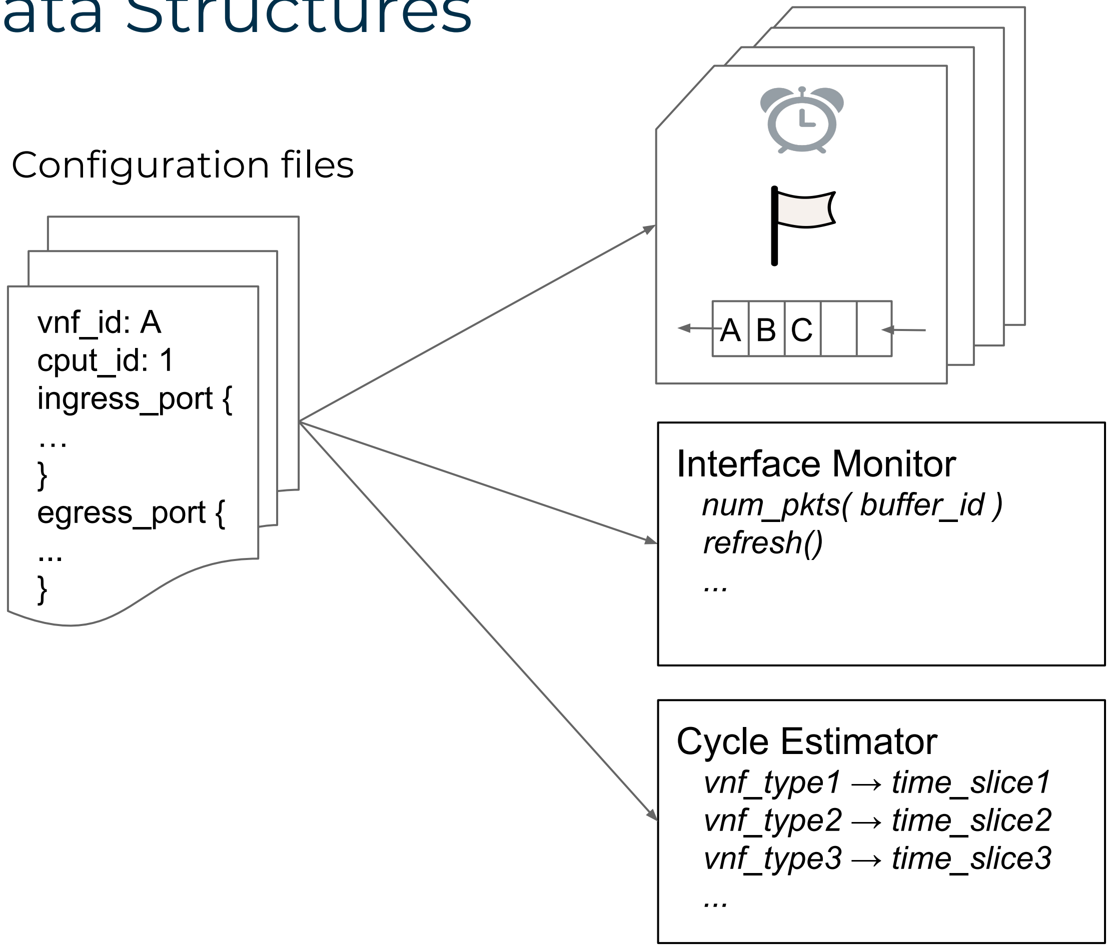


How UNiS works

Per-core Data structure

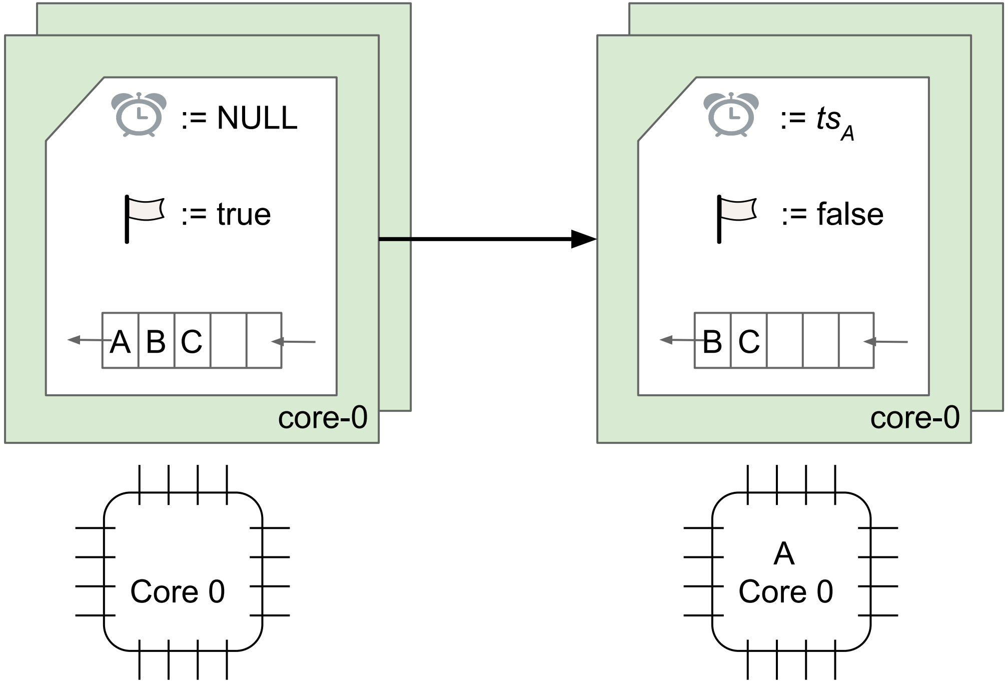


Other Data Structures

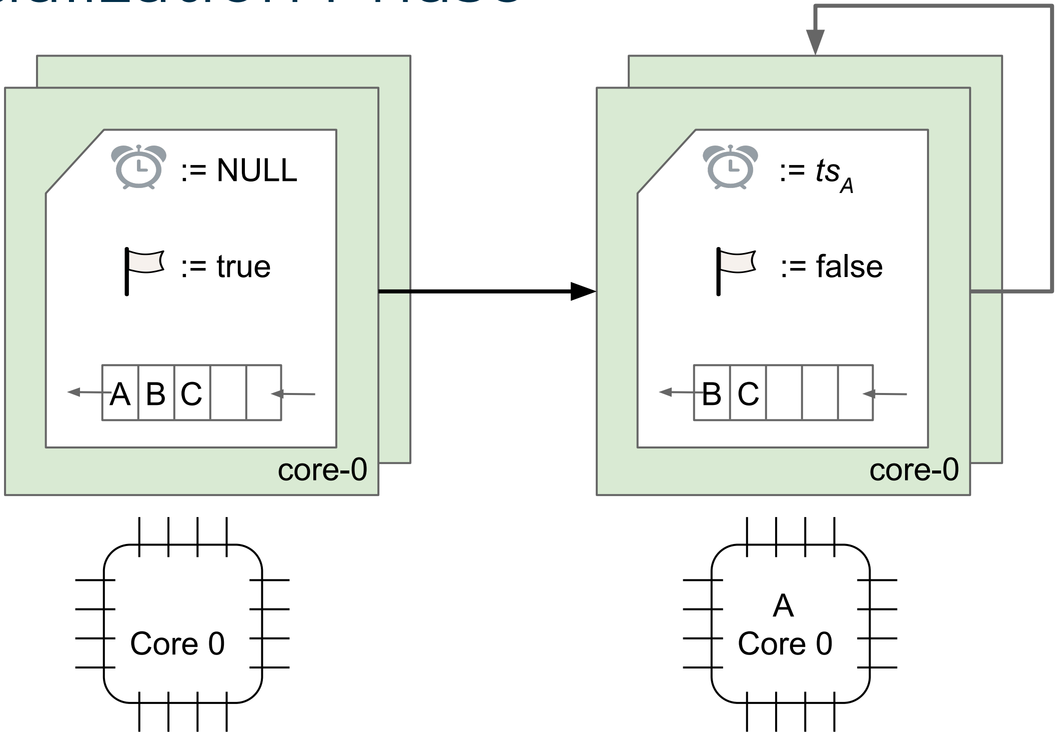


Scheduling Algorithm

Initialization Phase



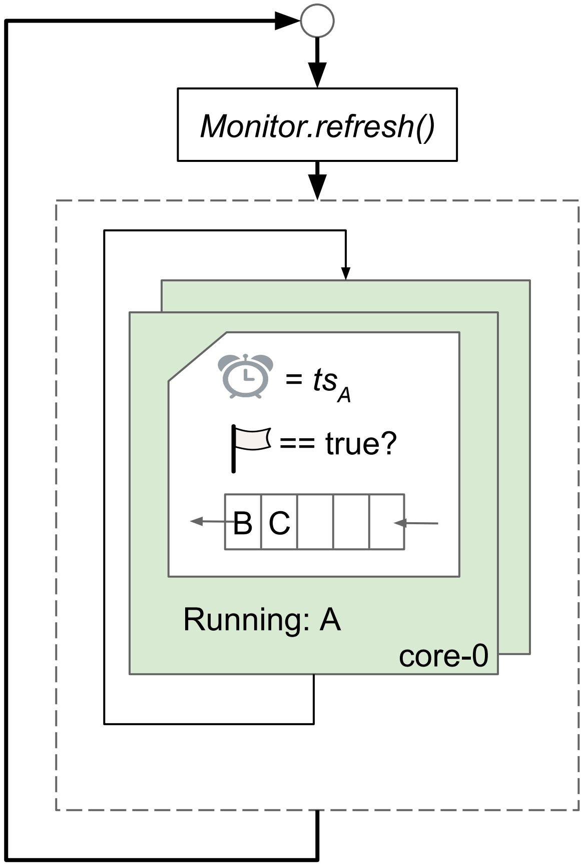
Initialization Phase



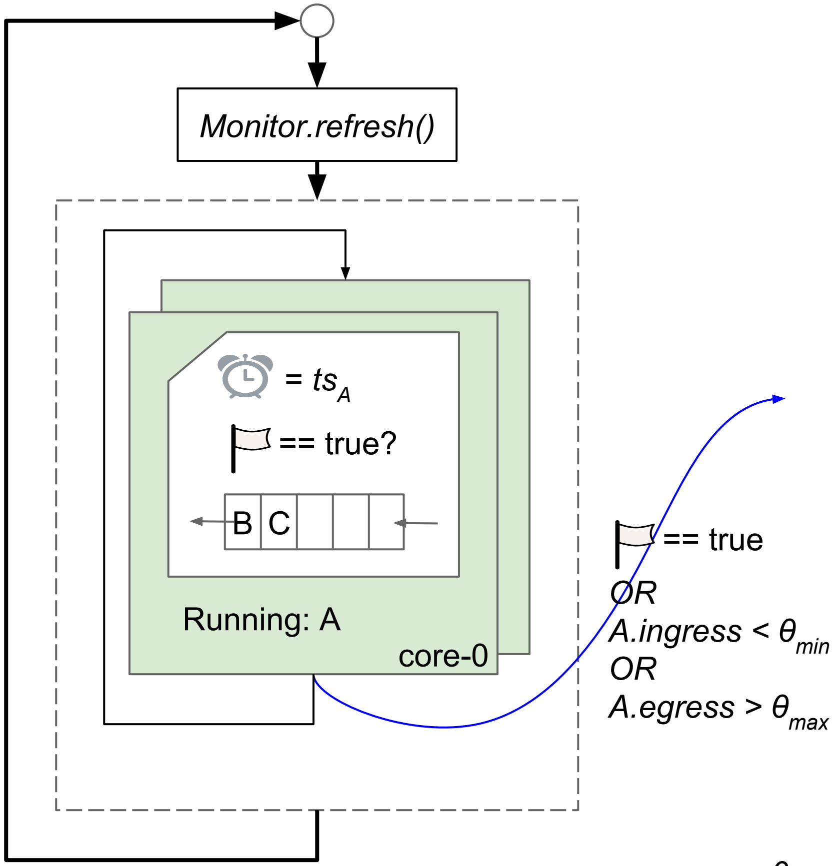
Move to the next core,

repeat the initialization

Execution Phase

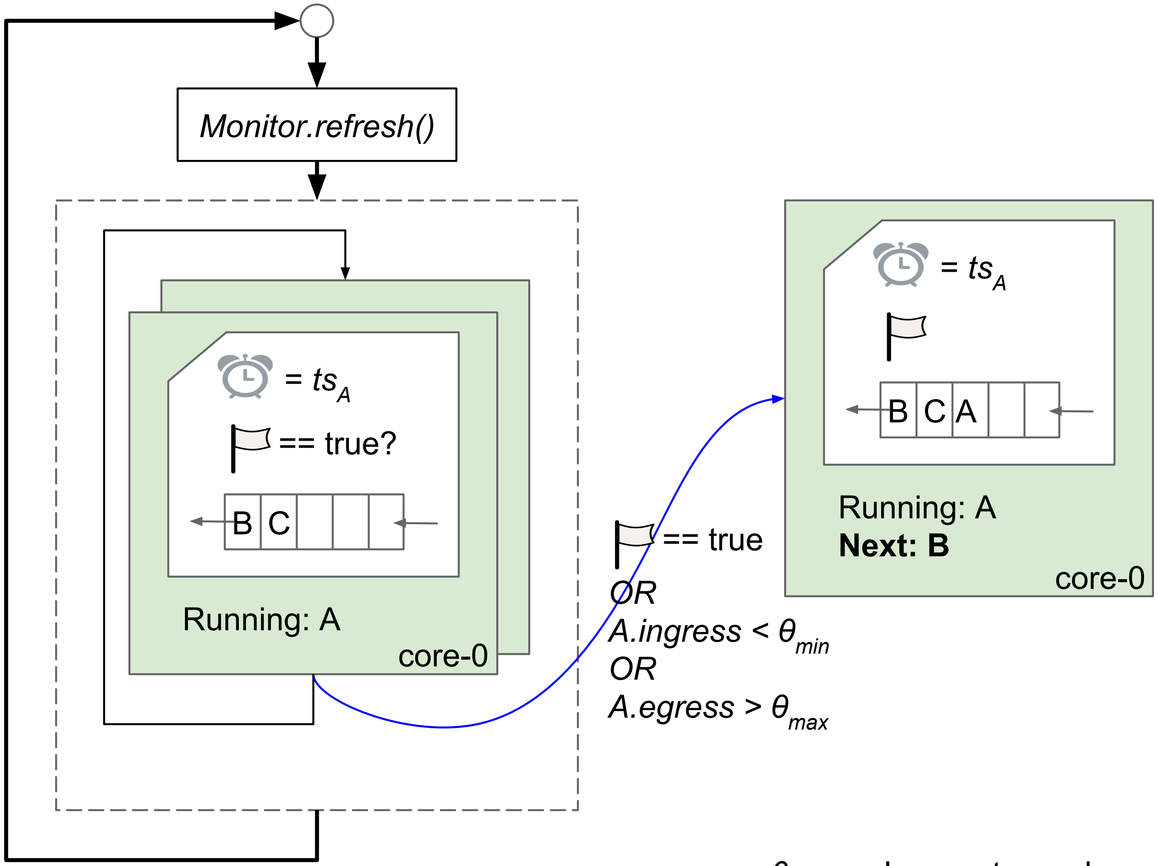


Execution Phase



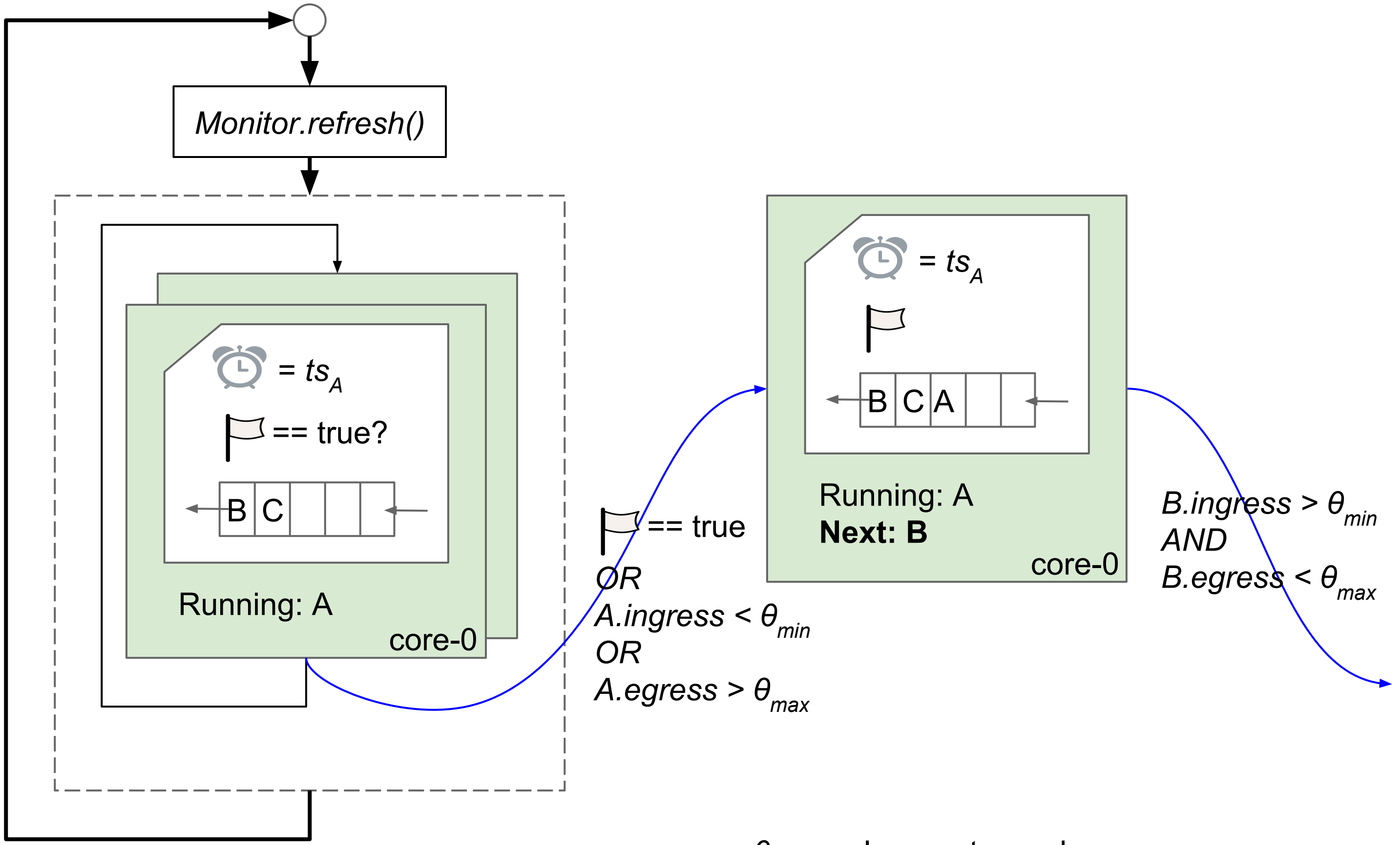
θ_{min} : Low watermark
θ_{max} : High watermark

Execution Phase



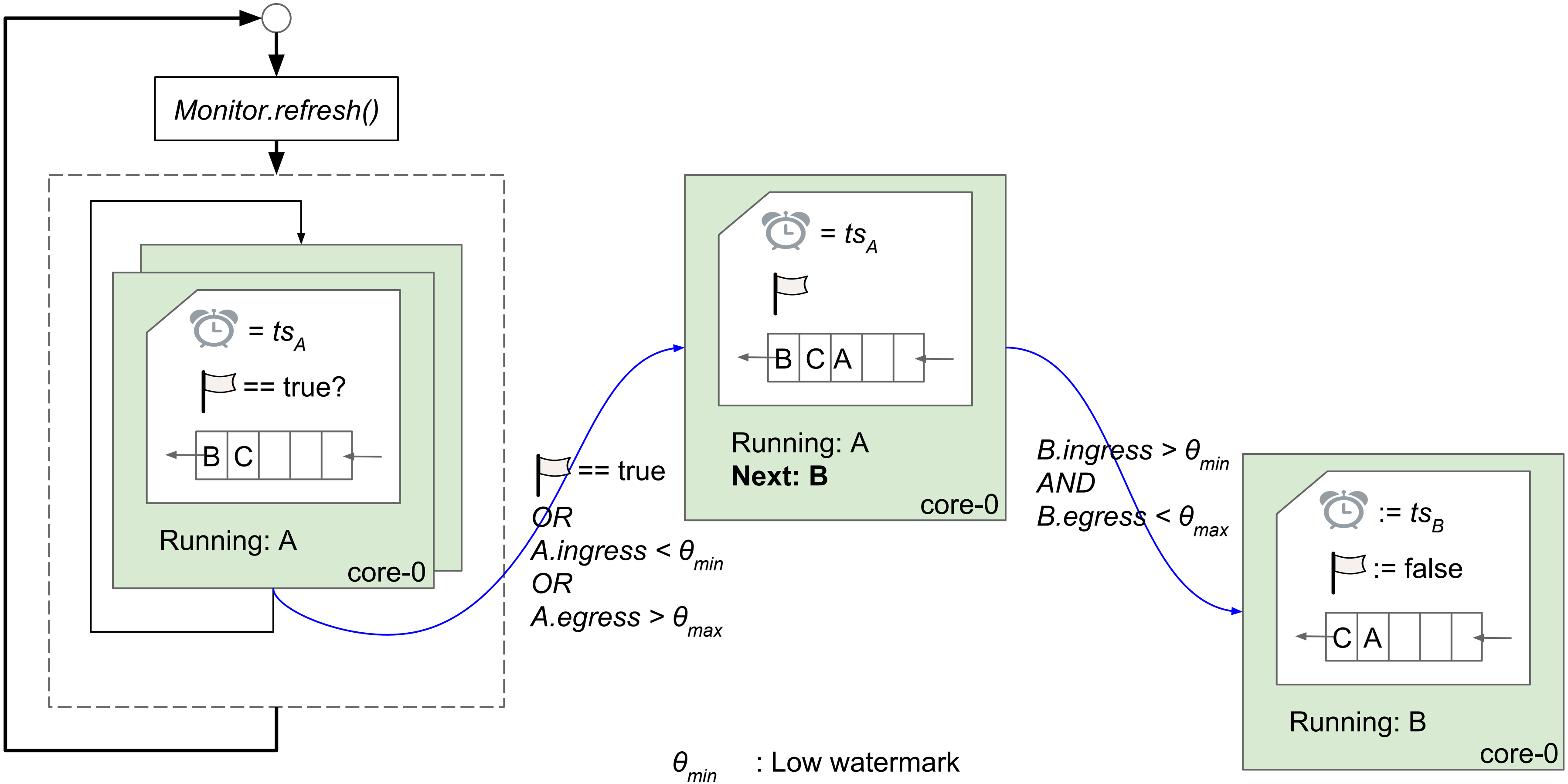
θ_{min} : Low watermark
θ_{max} : High watermark

Execution Phase



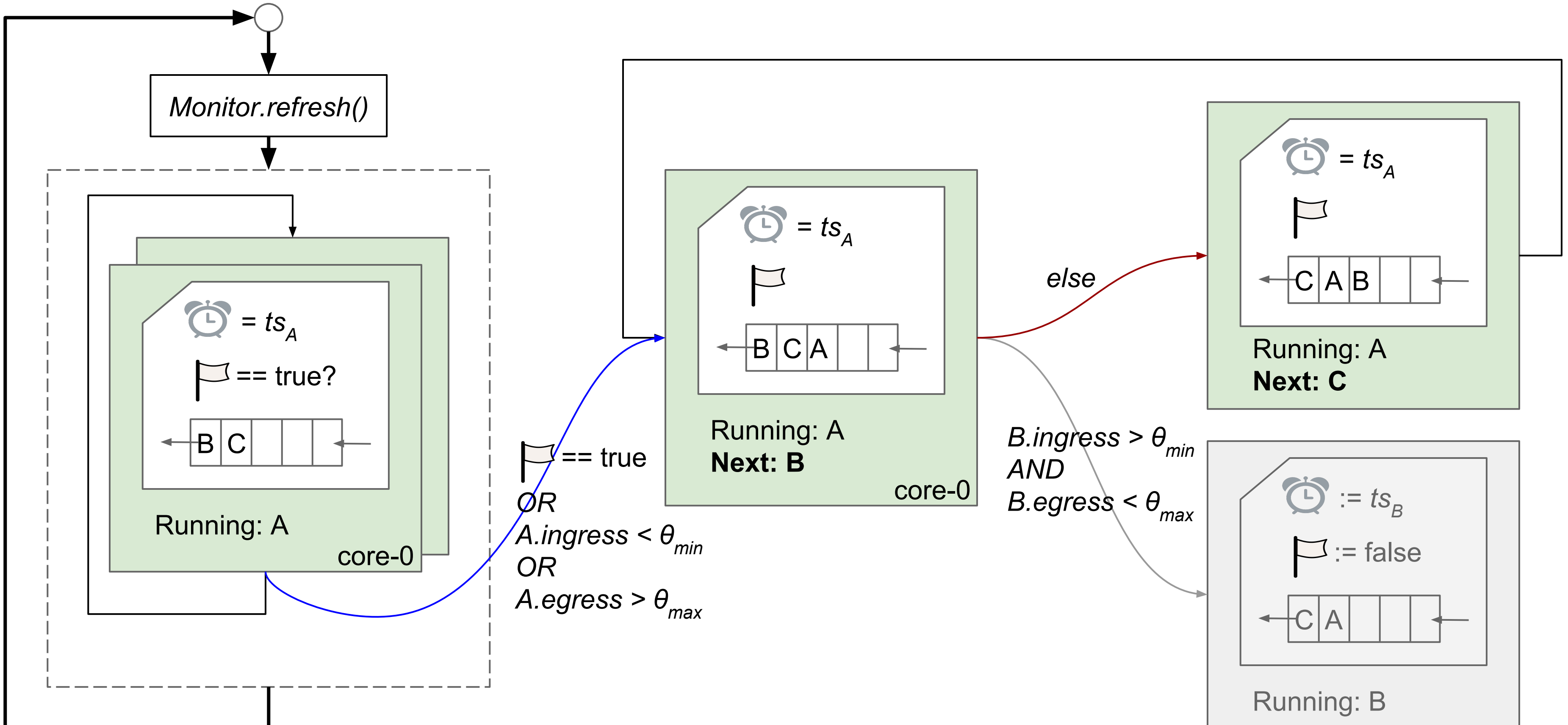
θ_{min} : Low watermark
θ_{max} : High watermark

Execution Phase



θ_{min} : Low watermark
θ_{max} : High watermark

Execution Phase



θ_{min} : Low watermark
θ_{max} : High watermark

Experiment Setup

Testbed

- Two back-to-back connected machines
- Intel X710-DA 10Gbps NIC
- Intel Xeon E3-1230v3 3.3Ghz 4-core CPU
- 16GB memory

Testbed

- Two back-to-back connected machines
- Intel X710-DA 10Gbps NIC
- Intel Xeon E3-1230v3 3.3Ghz 4-core CPU
- 16GB memory

VNF Types

- Fixed cost
  - *Light* : 50 cycles/packet
  - *Medium* : 150 cycles/packet
  - *Heavy* : 250 cycles/packet
- Variable cost
  - Step function proportional to packet size.

Workload

- Synthetic traffic
  - *DPDK-pktgen*
  - *Moongen*
- Real data-center traffic
  - UNI1 traces[1]

1. T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in Proceedings of ACM IMC. ACM, 2010, pp. 267–280.

Evaluation

Compared approach

Cooperative scheduling approach

- VNF built with a scheduling logic
  - Yield CPU after processing certain batches of packets
  - Minimal overhead

Compared approach

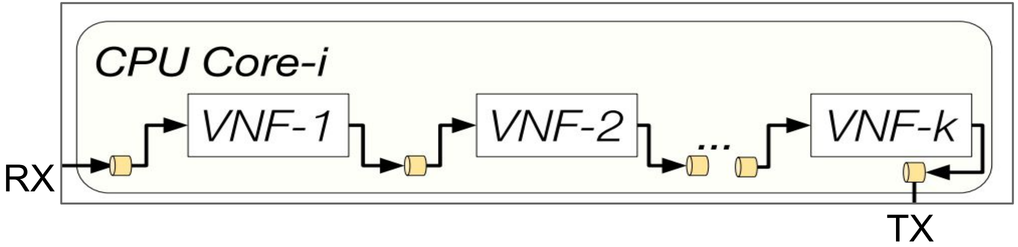
Cooperative scheduling approach

- VNF built with a scheduling logic
  - Yield CPU after processing certain batches of packets
  - Minimal overhead

Why not *Flurries* or *NFVNice*?

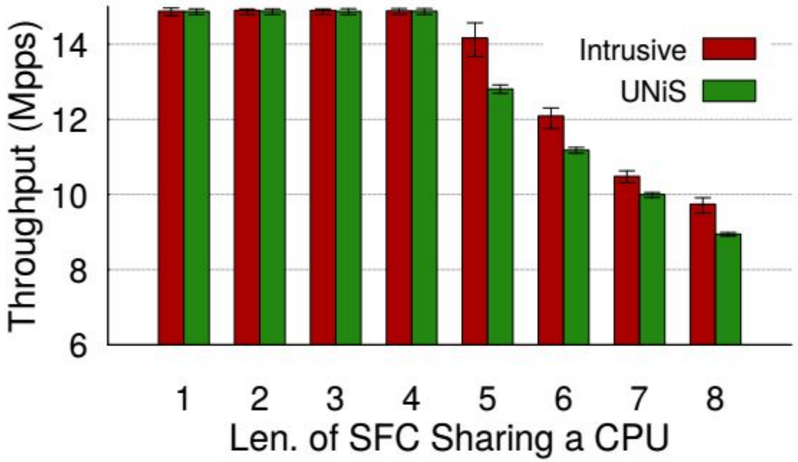
Evaluation Scenario 1
SFC with fixed and uniform cost VNFs

All VNFs in the SFC has the same fixed processing cost.

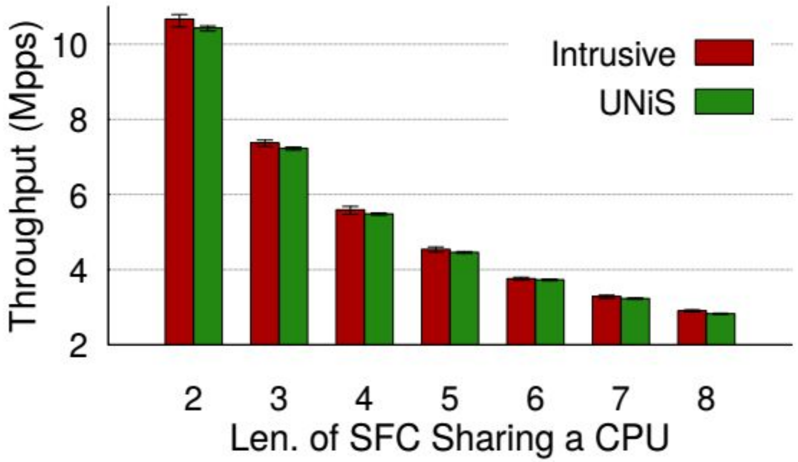


Evaluation Scenario 1
SFC with fixed and uniform cost VNFs

Workload: synthetic traffic 64B packet size at 10Gbps



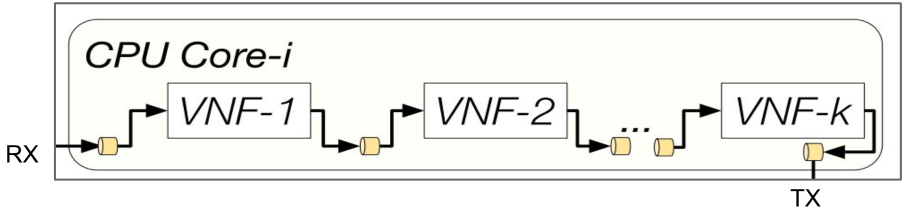
(a) Throughput with Light VNFs

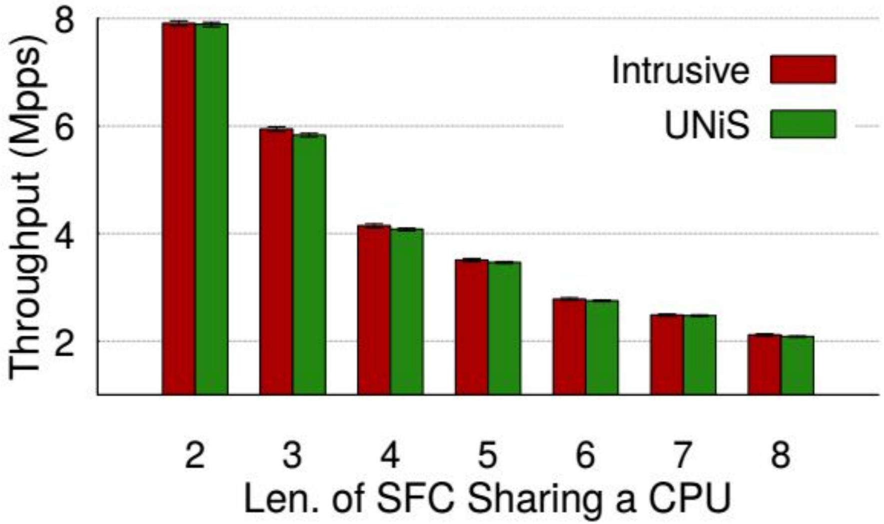


(b) Throughput with Medium VNFs

2. SFC with fixed and <u>non-uniform</u> cost VNFs

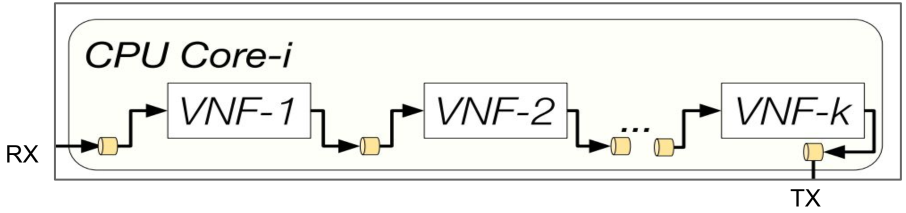
Interleaving *Medium* and *Heavy* flavor VNFs.



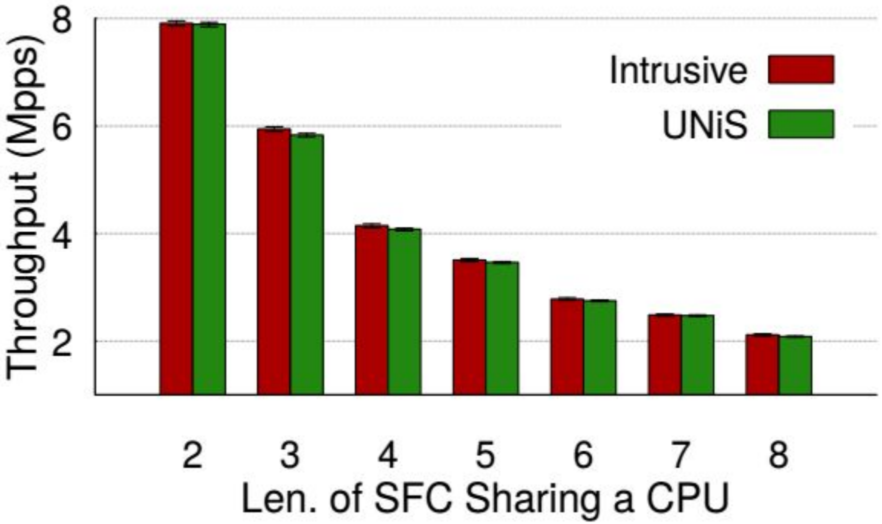


2. SFC with fixed and non-uniform cost VNFs

Interleaving *Medium* and *Heavy* flavor VNFs.



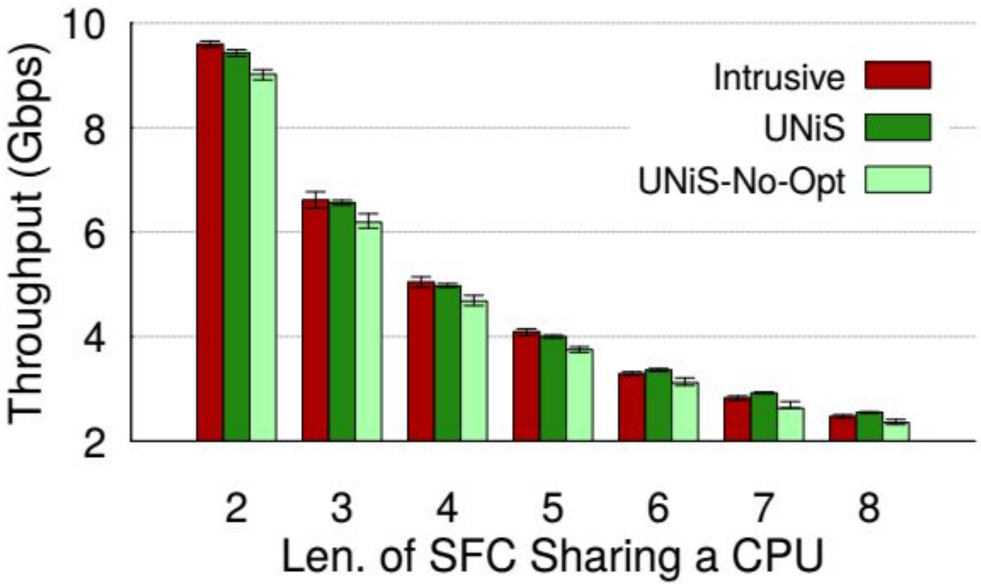
< 2% diff



3. SFC with variable cost VNFs

VNF processing costs vary proportionally to the packet sizes.

Workload: with real data center traffic capture.

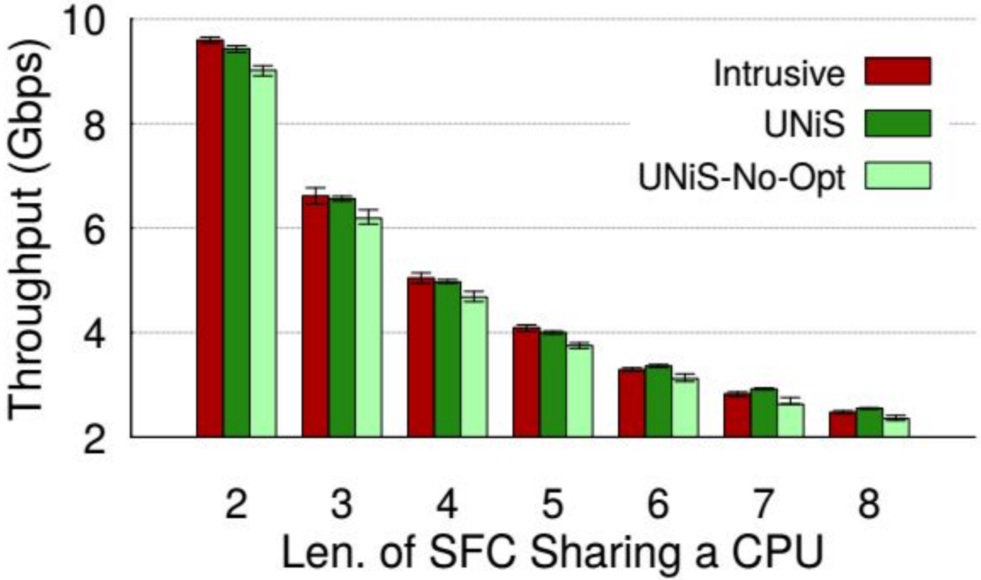


3. SFC with variable cost VNFs

VNF processing costs vary proportionally to the packet sizes.

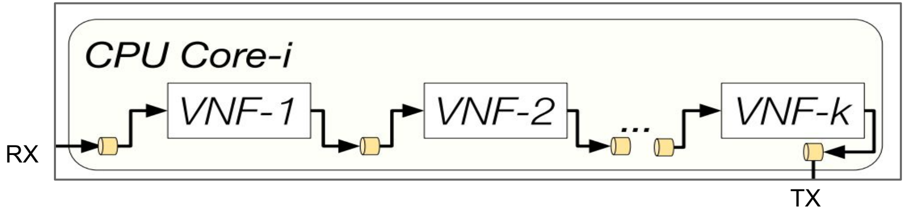
Workload: with real data center traffic capture.

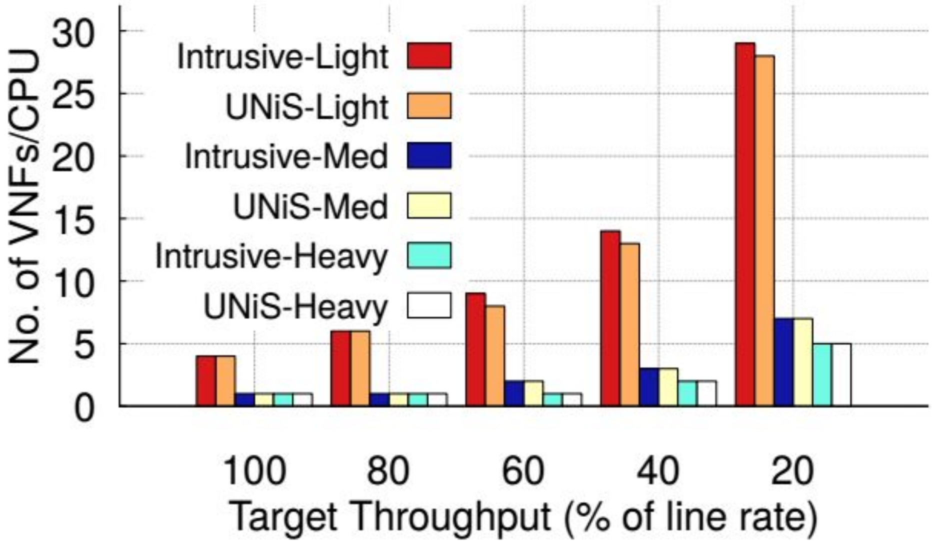
Intrusive vs UNiS : +2%
UNiS vs UNiS-No-Opt : +10%



4. VNF density on a single core

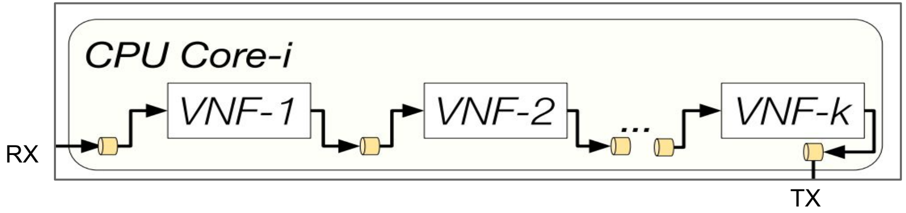
Fixed and uniform cost VNFs in an SFC



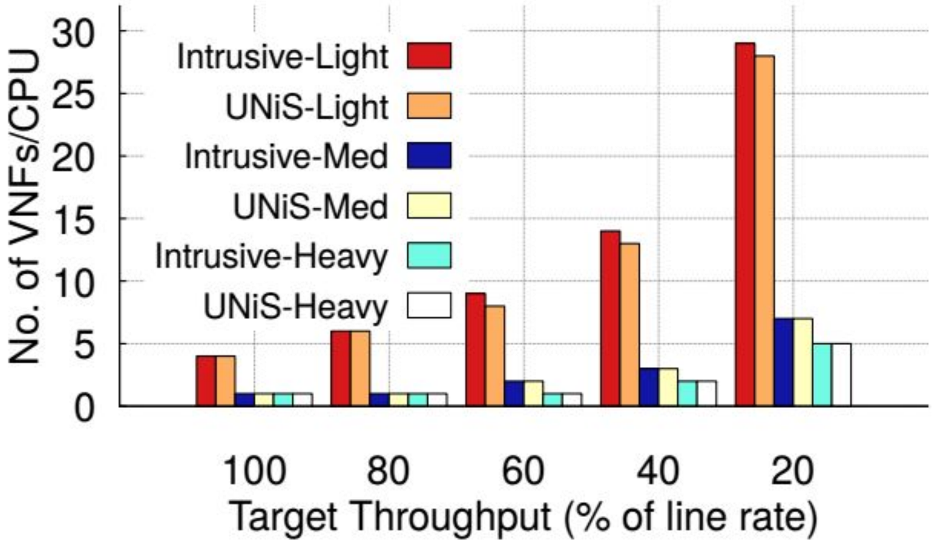


4. VNF density on a single core

Fixed and uniform cost VNFs in an SFC



UNiS can pack <u>almost</u> the same number of VNFs



Conclusion

- Default Linux schedulers (CFS, RT) are inadequate for VNF workload
- State-of-the art solutions are *intrusive*
- UNiS achieved its goals
    - a novel non-intrusive scheduling approach
    - does not require kernel modification
    - consider the VNFs order in SFC
- Experimental results show UNiS performance is promising
- UNiS saves CPU resource by packing multiple VNFs to same cores

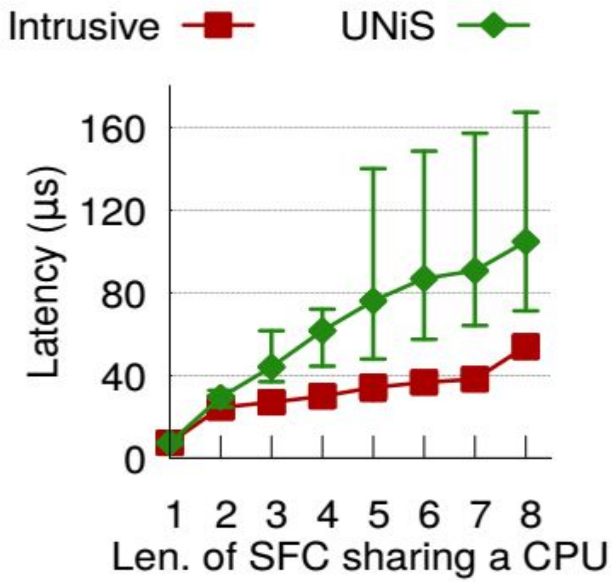
Thank you

Extra Slides

Latency

Scenario : SFC with fixed and uniform cost VNFs

Workload : Synthetic traffic 128B packet size at 80% sustainable throughput



(c) Latency with Medium VNFs

5. Multiple SFCs across multiple cores

# VNFs in SFC	# VNFs on Core-1	# VNFs on Core-2	Int. Thput. (Mpps)	UN*i*S Thput. (Mpps)
(a) S1 = 3 S2 = 1	S1 = 3 S2 = 1	–	S1 = 5.31 S2 = 5.31	S1 = 5.30 S1 = 5.21
(b) S1 = 4 S2 = 4	S1 = 3 S2 = 1	S1 = 1 S2 = 3	S1 = 5.24 S2 = 5.24	S1 = 5.10 S2 = 5.14
(c) S1 = 8	S1 = 4	S1 = 4	S1 =5.41	S1 = 5.34

UNiS Key Ideas

1. Estimate VNF processing cost
2. Allocate time_slice for each VNF
3. Leverage buffer occupancy information to optimize/adapt
4. Consider VNFs ordering in scheduling
5. Control the execution from userspace
6. Blackbox approach.

Initialization Phase

- Parse the SFC configurations
- Create per-core data structures
  - wait queue, timer, *expiry_flag*
- Initialize each queue according to the VNFs order in the SFC
- Assign *time_slice* for each VNF according to the Cycle Estimator results.

Execution Phase

- Traverse each of the per-core DS
- Pick the pid at the queue head, run the pid, set the timer for it.
- Periodically check
    - IF *expiry_flag* for a core is set

      OR ingress buffer is empty OR egress buffer is almost full
        - Pick the next process
        - Check if its ingress buffer is not empty
        - Switch the running process
        - Reset the timer

Execution Phase

- Traverse each of the wait queues
- Pick the pid at the head, run the pid, set the timer for it.
- Periodically check
  - IF *expiry_flag* for a core is set

    OR ingress buffer is empty OR egress buffer is almost full
    - Pick the next process
    - Check if its ingress buffer is not empty
    - Switch the running process
    - Reset the timer

Buffer Occupancy based Optimization

Algorithm 1: UN*i*S Scheduling Loop

Input: *cores* = Set of CPU cores; $\mathcal{T}$ = monitoring interval; *timer_subsystem, process_controller, monitor* = Handler to UN*i*S system components

```
function ScheduleVNFs()
    timer_subsystem.monitoring_timer.start(T)
    /* The system is initialized by running
       the first VNF in every core's wait
       queue and creating corresponding per
       core timers.                          */
    while true do
        /* Take scheduling decision after
           every T μs                        */
        if timer_subsystem.monitoring_timer.is_expired() ==
         false then  continue
        /* Iterate over each core and check if
           a new VNF can be scheduled        */
        foreach core ∈ cores do
            C ← core.cur_vnf
            if core.timer.is_expired() or
             monitor.num_pkts(C.ingress) ≤ θ_min or
             monitor.num_pkts(C.egress) ≥ θ_max then
                /* Iterate over the wait queue
                   (WQ) and find a VNF that
                   has sufficient work to do  */
                core.WQ.push(C)
                N ← core.WQ.pop()
                while (C ≠ N) and
                 (monitor.num_pkts(N.ingress) ≤ θ_min or
                 monitor.num_pkts(N.egress) ≥ θ_max) do
                    core.WQ.push(N)
                    N ← core.WQ.pop()
                end
            end
                /* If a candidate VNF is found,
                   allocate it a time_slice       */
                if C ≠ N then
                    core.timer.stop()
                    time_slice ← cost_estimator.get_cost(N) * γ
                     * monitor.pkt_cap(N.egress)
                    process_controller.deactivate(C)
                    process_controller.activate(N)
                    core.cur_vnf ← N
                    core.timer.reset(time_slice)
                end
            end
        end
        timer_subsystem.monitoring_timer.reset(T)
end
```