UNiS: A User-space Non-intrusive Workflow-aware Virtual Network Function Scheduler

Anthony¹, Shihabur Rahman Chowdhury¹, Tim Bai¹, Raouf Boutaba¹, Jerome François²

University of Waterloo¹
INRIA Nancy Grand Est²
The Development of Network Function

Hardware middleboxes

Deep Packet Inspection

Firewall
The Development of Network Function

Hardware middleboxes

Virtual Network Function

Orchestrator

Commercial Off-the-shelf (COTS) Servers

Deep Packet Inspection

Firewall
The Development of Network Function

Hardware middleboxes

Virtual Network Function

Orchestrator

Commercial Off-the-shelf (COTS) Servers

Common Practices

Kernel bypass technologies (DPDK, Netmap, Solarflare, etc)
+
CPU core pinning
+
Poll mode

Deep Packet Inspection

Firewall

Performance
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, ...
The state-of-the-art

1. *Flurries*  
   Poll mode + interrupt

2. *NFV-Nice*  
   *Flurries* + back pressure


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.


UNiS

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

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
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 \texttt{sched\_priority} parameter
Interface Monitor

Goal
- Provide buffer occupancy monitoring data

Implementation
- DPDK \textit{rte\_ring} library
  - zero copy packet transfer
How UNiS works
Per-core Data structure

Core 0

Core 1

... 

Core n
Other Data Structures

- Configuration files
  
  ```
  vnf_id: A
  cput_id: 1
  ingress_port {
    ...
  }
  egress_port {
    ...
  }
  ```

- Interface Monitor
  
  ```
  num_pkts(buffer_id)
  refresh()
  ...
  ```

- Cycle Estimator
  
  ```
  vnf_type1 → time_slice1
  vnf_type2 → time_slice2
  vnf_type3 → time_slice3
  ...
  ```
Scheduling Algorithm
Initialization Phase

Core 0

:= NULL

:= true

A B C

Core 0

:= \text{true}

Core 0

:= \text{ts}_A

:= \text{false}

B C

Core 0

:= \text{false}

Core 0
Initialization Phase

Initialization:

- **Core 0**
  - $A := \text{NULL}$
  - $B := \text{true}$
  - $C := \text{false}$

Move to the next core, repeat the initialization.
Execution Phase

```java
Monitor.refresh()

B = ts_A

== true?

Running: A

core-0
```
Execution Phase

Monitor.refresh()

$B = ts_A$

$== true?$

Running: A

core-0

$\theta_{min}$: Low watermark

$\theta_{max}$: High watermark

$A.ingress < \theta_{min}$

OR

$A.egress > \theta_{max}$
Execution Phase

Monitor.refresh()

- $\text{ts}_A$ == true?
- OR $A.\text{ingress} < \theta_{\text{min}}$
- OR $A.\text{egress} > \theta_{\text{max}}$

$\theta_{\text{min}}$ : Low watermark
$\theta_{\text{max}}$ : High watermark
Execution Phase

- `Monitor.refresh()`

- `ts_A` is the time

- `A.ingress < \theta_{\text{min}}` or `A.egress > \theta_{\text{max}}` or `B.ingress > \theta_{\text{min}}` or `B.egress < \theta_{\text{max}}`

- \( \theta_{\text{min}} \) : Low watermark
- \( \theta_{\text{max}} \) : High watermark
Monitor.refresh()

Running: A

Next: B

Running: B

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

```
Monitor.refresh()

Running: A
Next: B

B.ingress > θ_{\text{min}} \quad \text{AND} \quad B.egress < θ_{\text{max}}

else

Running: B

θ_{\text{min}}: \text{Low watermark}
θ_{\text{max}}: \text{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\(^1\)

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 **non-uniform** cost VNFs

Interleaving *Medium* and *Heavy* flavor VNFs.
2. SFC with fixed and non-uniform cost VNFs

Interleaving Medium and Heavy flavor VNFs.
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

![Diagram of VNFs on a single core](image)

![Bar chart showing VNF density vs target throughput](image)
4. VNF density on a single core

Fixed and uniform cost VNFs in an SFC

UNiS can pack almost 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

<table>
<thead>
<tr>
<th># VNFs in SFC</th>
<th># VNFs on Core-1</th>
<th># VNFs on Core-2</th>
<th>Int. Thput. (Mpps)</th>
<th>UNiS Thput. (Mpps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) S1 = 3</td>
<td>S1 = 3</td>
<td>–</td>
<td>S1 = 5.31</td>
<td>S1 = 5.30</td>
</tr>
<tr>
<td>S2 = 1</td>
<td>S2 = 1</td>
<td></td>
<td>S2 = 5.31</td>
<td>S1 = 5.21</td>
</tr>
<tr>
<td>(b) S1 = 4</td>
<td>S1 = 3</td>
<td>S1 = 1</td>
<td>S1 = 5.24</td>
<td>S1 = 5.10</td>
</tr>
<tr>
<td>S2 = 4</td>
<td>S2 = 1</td>
<td>S2 = 3</td>
<td>S2 = 5.24</td>
<td>S2 = 5.14</td>
</tr>
<tr>
<td>(c) S1 = 8</td>
<td>S1 = 4</td>
<td>S1 = 4</td>
<td>S1 = 5.41</td>
<td>S1 = 5.34</td>
</tr>
</tbody>
</table>
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
Initialization Phase

- Parse the SFC configurations
- Create per-core data structures
  - wait queue, timer, \textit{expiry\_flag}
- Initialize each queue according to the VNFs order in the SFC
- Assign \textit{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: UNiS Scheduling Loop

Input: cores = Set of CPU cores; \( T \) = monitoring interval;
timer_subsystem, process_controller, monitor = Handler
to UNiS system components

1 function ScheduleVNFs()
2    timer_subsystem.monitoring_timer.start(\( T \))
3        /* The system is initialized by running
4           the first VNF in every core's wait
5           queue and creating corresponding per
6           core timers. */
7        while true do
8            /* Take scheduling decision after
9            every \( T \) \( \mu \)s */
10           if timer_subsystem.monitoring_timer.is_expired() ==
11              false then continue
12           /* Iterate over each core and check if
13              a new VNF can be scheduled */
14           foreach core \( \in \) cores do
15              \( C \leftarrow \) core.cur_vnf
16              if core.timer.is_expired() or
17                  monitor.num_pkts(C.ingress) \( \leq \) \( \theta_{min} \)
18                  or monitor.num_pkts(C.egress) \( \geq \) \( \theta_{max} \) then
19                  /* Iterate over the wait queue
20                     (\( W/Q \)) and find a VNF that
21                     has sufficient work to do */
22                  core.WQ.push(C)
23                  \( N \leftarrow \) core.WQ.pop()
24                  while (\( C \neq N \)) and
25                      (monitor.num_pkts(N.ingress) \( \leq \) \( \theta_{min} \)
26                      or monitor.num_pkts(N.egress) \( \geq \) \( \theta_{max} \)) do
27                        core.WQ.push(N)
28                        \( N \leftarrow \) core.WQ.pop()
29                    end
30              end
31          end
32        end
33        /* If a candidate VNF is found,
34           allocate it a time_slice */
35        if \( C \neq N \) then
36            core.timer.stop()
37            time_slice \( \leftarrow \) cost_estimator.get_cost(\( N \)) \( \ast \) \( \gamma \)
38            * monitor.pkt_cap(\( N.egress \))
39            process_controller.deactivate(\( C \))
40            process_controller.activate(\( N \))
41            core.cur_vnf \( \leftarrow \) \( N \)
42            core.timer.reset(time_slice)
43        end
44    end
45    timer_subsystem.monitoring_timer.reset(\( T \))