UNiS: A **U**ser-space **N**on-**i**ntrusive Workflow-aware Virtual Network Function **S**cheduler

University of Waterloo
Anthony (20663343) and Tim Bai (20489611)

1. Background and Motivation

1.1 Existing NFV Platforms and Their Problems

Over the last few years, many Network Function Virtualization (NFV) platforms have been proposed. Some DPDK-based performant NFV platforms proposed in 2016, such as NetVM [7], OpenNetVM [8] and NetBricks [9], assume that each Network Function (NF) runs on a dedicated CPU core in a polling mode. As such, each NF occupies one core and continuously checks its input buffer for packets to process. When a system is fully loaded, this approach fully utilizes all the CPU cores and gives an excellent throughput. However, this approach wastes many CPU cycles if the ingress buffer is empty or if the ingress rate is low. It also limits the number of NFs that can be instantiated to the number of available cores. Flurries [10], another container-based NFV platform, attempts to fix this by proposing a different system architecture. An additional *Flow-Director* in Flurries is proposed to distribute the incoming packets to NF instances and wake them up using Linux semaphore. However, the NFs need to voluntarily yield the CPU after they have finished processing several batches of packets based on certain scheduling policies. This approach is considered "*intrusive*" as it requires NFs to cooperatively yield the resource. In other words, NF source code modification is required if existing NFs are not cooperative. Recently, μNF [11], a NFV platform proposed by Prof. Boutaba's research group at the University of Waterloo, is focusing on the decomposition of NF functionalities into smaller functions to avoid redundancies in a service chain. One of the challenges for μNF is to efficiently run many small NFs (microservices) on the available CPU cores. The core sharing (NF scheduling) mechanism proposed in μNF platform has the notion of *Weighted Fair Queuing,* in which each NF has to cooperate and yield the CPU after processing *k* batches of packets. The value of *k* represents the weight in using the CPU share; the larger *k,* the more cpu time.

1.3 Limitation of Linux Scheduler

Using μNF as the NFV platform, our early experiments show that Linux scheduler classes, *Completely Fair Scheduler (CFS) and Real Time Scheduler (SCHED_RR),* are not suitable for scheduling NF workloads on CPU cores. Since kernel version 2.6.23, CFS has been the default Linux scheduler. The basic principle of CFS is that processes sharing the same core will have a fair/balanced proportion of the CPU share. The fairness is aimed (if not achieved) by maintaining and monitoring the *virtual runtime* of each process which is used for determining which process should be preempted. Initially, CFS was designed for scheduling general processes running on Linux. These processes are either *I/O-bound (interactive)* or *processor-bound.* The other option is the *Real Time Scheduler (SCHED_RR)* that implements a first-in first-out (FIFO) scheduling algorithm with timeslices (a.k.a time quantum) and priorities. Under this scheduling algorithm, a running process is preempted when its allocated timeslice expires or when a higher priority process becomes ready. After a process is preempted, the next process with the same priority is selected in FIFO Round Robin (RR) order.

One problem with SCHED_RR is that the NFs workflow (i.e., the order of NFs in a service chain) does not translate into the execution order on the CPU. Although processes are selected in FIFO manner, it is insufficient to guarantee that the execution order is the same as the NFs order in the service chain. For instance, a simple service chain consisting of NF1→NF2→NF3 is scheduled on one core. Consider a case when NF3 is about to be instantiated, NF2 is running, and NF1 is in the wait queue. In this case, NF3 is being put into the end of the wait queue behind NF1 and after NF2 has used up all the timeslice, NF2 will be placed back at the end of the wait queue behind NF3. As a result, the execution order is NF1→NF3→NF2, which is not desirable. Also note that NF3 might be scheduled with its ingress buffer empty. In this case, both CPU cycles for context switches and the NF3 timeslice will be wasted. In addition, this out-of-order execution might also decrease cache warmness. The situation is more likely to occur when a larger number of processes are being scheduled. Another problem with SCHED_RR is that the timeslice is not per process based; it is *one-value-for-all* of the processes scheduled with this policy. When NFs are deployed as a chain, each type of NFs often has a different computation load.

Therefore, a scheduler should apportion the shares of the CPUs to NFs based on their computation load, instead of aiming for fairness among NFs.

1.4 The Proposed Solution

We realize that complete fairness and *one-value-for-all* are not desired properties in scheduling NFs on the same core. In this project, we propose ***UNiS**: A **U**ser-space **N**on-**I**ntrusive Workflow-aware VNF **S**cheduler*. The primary goal of UNiS is to devise a non-intrusive and workflow aware scheduling entity that schedules multiple NF processes on one core with minimal impact on the performance. *Non-intrusive* means no modification or cooperation from the NFs is required. *Workflow-aware* means that the scheduling of NF processes is following the pipeline order of the NFs in a chain. This way, we expect the current process to leverage the cache warmness resulting from the previously scheduled process.

Without modifying the NF code, UNiS is able to significantly improve the compute resource efficiency of those NFV platforms that are using a polling mode or dedicated cores by sharing cores whenever possible. The target NF processes that can be grouped together into one core are those NFs that are lightweight. For example, *Time To Live (TTL) decrementer* and *Mac Swapper* are considered lightweight NFs. Moreover, UNiS allows a consolidation of NF processes into the same core when the throughput requirement is relatively low. For instance, an intrusion detection system (IDS) can be considered a heavy NF because it accesses packet payload and often performs string or signature matching. However, if the incoming traffic is just 75% of the capacity the IDS can handle, it is likely that we can group the next lightweight NF into the IDS's CPU while maintaining the required throughput.

2. UNiS Design and Implementation

An overall UNiS system architecture is shown in the yellow background in Figure 1. UNiS consists of a Cycle Estimator module, a Ring Monitoring module, a Timer subsystem, and a Process Controller module. Interactions between UNiS and μNF are represented by the dashed lines, which mean it is not on the fast path of

the µNF platform. Note that Cycle Estimator is run beforehand or separately from an operational platform. Details of each UNiS module are explained in the next subsections.

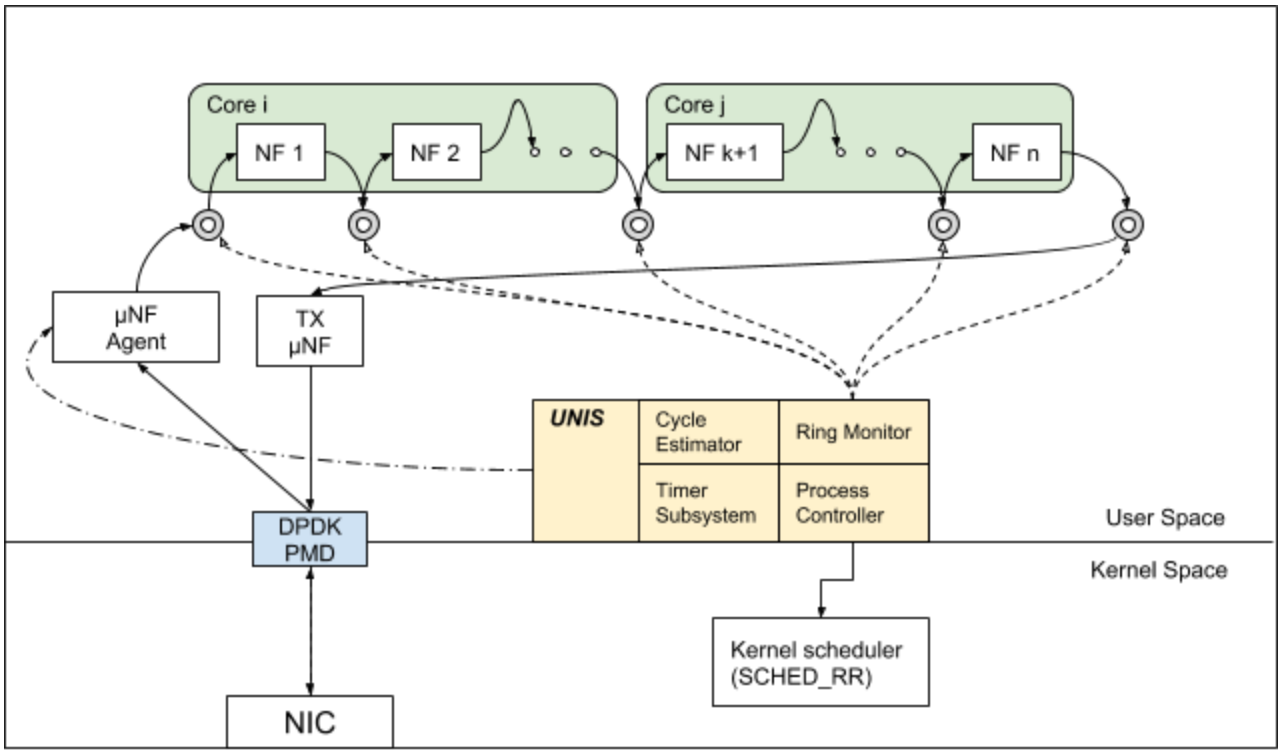

Figure 1. UNIS System Architecture

2.1 Cycle Estimator

Being able to estimate processing cost of a NF is important in designing UNIX because it gives us the profile of each NF. These profiles will later be used in determining the absolute and ratio values of the timeslices for different NFs.

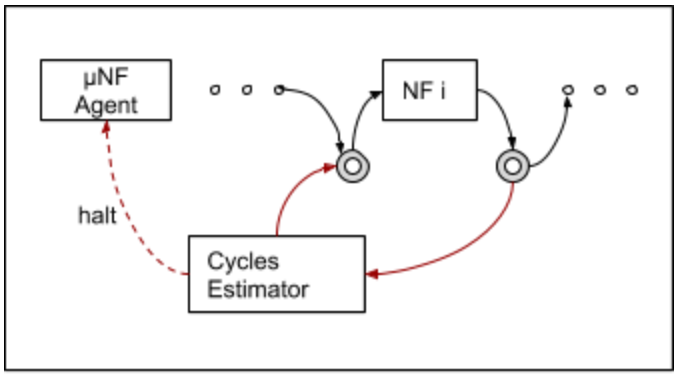

Figure 2. Cycle Estimator

As shown in Figure 2, our design of the Cycle Estimator does not require instrumental code to be inserted into the NF source code. To estimate the processing cost of a NF-i, the Cycle Estimator first asks the *µNF Agent* to halt reading packets from the Network Interface Controller (NIC) to the system. Afterwards, the Cycle Estimator flushes the remaining packets residing in the NF-i's ingress and egress ring. Then, the Cycle Estimator crafts a batch of 32 packets, takes a timestamp (*CLOCK_MONOTONIC*) and injects the batch into the ingress ring of the NF-i. As soon as it is injected, our estimator busy-waits on the egress ring of the NF-i to retrieve the injected batch. Once the batch is captured by the Cycle Estimator, a second timestamp is taken and the difference with the first timestamp is calculated. These steps are repeated thousands of times to obtain statistics. Other than the processing cost in nanoseconds, the UNiS Cycle Estimator also provides the result in terms of cpu cycles using *rdtscp* instructions.

2.2 Ring Monitoring

The key enabler in devising a non-intrusive core sharing scheduler is the data collected by our monitoring system. UNiS adopts three important scheduling policies to optimize packet processing performance. First, whenever a new NF process is scheduled, it should have enough batches in its ingress ring to be processed during its timeslice. Second, during its timeslice, a running process will be preempted as soon as its ingress ring is empty or its egress ring is full. These two policies allow UNiS to avoid wasted work and packet drop. Third, a process should be allotted a timeslice long enough to process most of the batches in its ingress ring. This policy aims to avoid high overhead from rapid context switches.

To implement the first two policies, UNiS frequently monitors the occupancy of each ring connecting the NFs. These rings are *rte_ring* type and are created by the μNF platform, which runs as a dpdk *primary process*. After μNF is started, UNiS is run as a dpdk *secondary process* and updates the ring occupancy information every millisecond. Due to a potential concurrency issue, extra caution has to be taken when interpreting the ring information connecting NFs running on different cores. The third policy is enforced by calculating the share of each NF based on its processing cost, the batch size, and the ring size.

2.3 Timer Subsystem

Similar to the kernel scheduler class, UNiS also requires a timer subsystem to keep track of the cpu time used by a process. The timer can also be used to periodically refresh or display the ring occupancies for debugging purposes. However, two main drawbacks of any user space timer are related to the precision and the resource utilization. Our experiment with *Libevent* shows a low cpu utilization, but also with a low timer precision. Thus, it does not fit our scheduling use case that requires microsecond precision. On the contrary, *rte_timer* provides adjustable and precise timer accuracy at the expense of full cpu utilization.

In the kernel space, a scheduler is supported by a hardware timer; so, the need for busy-looping is alleviated. However, kernel module development and debugging are known to be more complicated and time consuming. Due to the time constraint in this course project, we did not took this path. We opted for the *DPDK rte_timer* subsystem and set its precision to 1 microsecond. The detailed usage of the timer in our UNiS scheduling algorithm will be elaborated in Subsection 2.6.

2.4 Process Selection

As mentioned in Section 1.3, one of the issues with the Linux real-time scheduler class *SCHED_RR* is that NF workflow does not translate to execution order. In UNiS, the order of NFs is preserved by using a FIFO C++ *std::queue* that keeps the process identifications (pids) in the desired order. This wait queue is kept in a per-core data structure named *sched_core.* Other than the wait queue, *sched_core* also has the *rte_timer,* a boolean *expiration* flag, and a core id. There fields are useful in determining when and which a process should be

run or suspended. The head of the queue represents the pid of the currently running NF process. When it needs to be preempted, the first element in the wait queue is moved to the end of the queue and the second element moves forward and is executed. Since the information about the order of NFs is gathered and processed before UNiS starts, the execution order of the NFs can be guaranteed to be the same as their order in the chain / workflow graph.

2.5 Process Controller

As a scheduler, UNiS must have a mechanism to effectively control the execution state of a process from the user space. The mechanism should be able to suspend a process and assign another process to run in a timely manner. We have tried several mechanisms, such as: *cgroup, POSIX signal,* and *sched_priority* in *SCHED_RR*. This led us to develop a process controller library to allow multiple process controller mechanisms to co-exist with a common API. It allows programmers to easily select different mechanisms and add new process controller implementations in the future.

Control Groups (cgroups) is a Linux kernel feature that limits and isolates the resource (CPU, memory, disk I/O, and network.) usage of a collection of processes. One way to use *cgroup* is to access its virtual file system directly. In our initial experiment, however, we found that the pids of NF processes could not be added into the appropriate *cgroup* hierarchy.

The *POSIX signal* allows a user space program to send notifications to a process or a specific thread. In our case, we are interested in SIGSTOP that instructs the operating system to stop a process for later resumption, and SIGCONT that instructs the operating system to continue the execution of a process. Our experiment with the POSIX signal was successful when scheduling two NF processes, so we decided to test with more NF processes. We tried to schedule three processes in such a way that a SIGSTOP is sent to the two of them and a SIGCONT is sent to the third one; then our experimental program sleeps for 5 microseconds and it repeats the same way with the other combination of processes. Surprisingly, the system's throughput drops to zero a few seconds after this experiment is started. Our conjecture is that the POSIX signal can not handle the high frequency (a few microseconds interval) invocations of these signals.

Finally, our last option works by adjusting the *sched_priority* values of the processes scheduled under the *SCHED_RR*. In this mechanism, any running process will never yield or be preempted by lower priority processes. Hence, the lower priority processes will never have a chance to run even though they are ready. By increasing or decreasing the priority of one of the processes, UNiS is able to control which process to run and which processes should be put into a waiting state.

2.6 Scheduling Algorithm

In this section, we assume that information about the processing cost of each NF has been obtained by the Cycle Estimator module. This information determines the shares core-sharing NF processes are assigned. In UNiS, this share is expressed in microseconds. The general rule in calculating the absolute values of the shares is *processing_cost_of_a_batch * (ring_size / batch_size) / C* microseconds where *C* is a constant value > 0. This allows one NF to fill approximately 1/*C* of the ring_size within the timeslice after it is scheduled to run. Thus, reducing the number of context switches. However, there might be a better choice in picking the absolute share value depending on the complexity of the processing graph.

In the current state of UNiS, the algorithm can be split into two phases: a preparation phase and a scheduling phase. In the preparation phase, it reads a log file produced by the *uNF-Agent* to deduce the mapping between *pids* of the NFs and their corresponding configuration files. Alternatively, the mapping could also be gained by iterating the running processes in the operating system one by one. However, due to time constraints in finishing this project, we chose the former approach that requires the NFV platform to provide a log file about pids of running NFs and their configuration files. In this phase, UNiS also constructs a relation between a ring connecting NFs with the *pids* who push and pull from this ring. The final step in this phase is initialising the shares assigned for the core-sharing NFs.

Figure 3 is the UNiS code snippet for the scheduling phase. Initially, *KickScheduler* in line 8 runs one process on each core and sets a timer. Then, the program enters a continuous loop to maintain the timer and do the NF scheduling work. Note that lines 16 to 59 are only executed every adjustable precision time interval. Line 29 decides whether a running pid should be preempted or not; it is preempted if the timer set for that pid is expired,

or if the input ring of that process is almost empty, or if the output ring is almost full. “Almost” here is defined as less than 1 batch (32 packets), but it is adjustable. If the preemption conditions are met, then a new process is selected from the wait queue for that core. The second UNiS policy mentioned in the Section 2.2 is implemented on line 39 - 40 that checks if the selected process has meaningful work to do, i.e., its input ring is not empty and its output ring is not full, before running it. Afterwards, a context switch happens on line 54 and a timer for the new process is setup according to its share (Line 55 - 59).

```
1static int __attribute__((noreturn))
2unis_mainloop(__attribute__((unused)) void *arg)
3{
4   uint64_t prev_tsc = 0, cur_tsc, diff_tsc;
5   uint64_t timeout = 0;
6
7   // run process on each core for first time.
8   KickScheduler();
9
10  while ( 1 ) {
11     // Manage timer subsystem
12     cur_tsc = rte_rdtsc();
13     diff_tsc = cur_tsc - prev_tsc;
14     // Resolution of the timer
15     if ( diff_tsc > TIMER_RESOLUTION_CYCLES ) {
16        rte_timer_manage();
17        prev_tsc = cur_tsc;
18        // Update ring info in EACH timer resolution time unit
19        RefreshRingInfo( rings_info );
20
21        // Check and manage expiry for each core
22        for ( int i = 0; i < core_sched.size() - 1; i++ ) {
23           unsigned pid = core_sched[ i ]->wait_queue.front();
24           unsigned coreid = core_sched[ i ]->core_id;
25           unsigned npid;
26
27           // Reschedule a core if a running process is expired, or
28           // the input ring is almost empty, or the output ring almost full.
29           if ( core_sched[ i ]->expired ||          \
30                prev_ring_map[ pid ]->occupancy < 32 || \
31                next_ring_map[ pid ]->occupancy > 2016 ) {
32
33              core_sched[ i ]->wait_queue.pop();
34              core_sched[ i ]->wait_queue.push( pid );
35              npid =  core_sched[ i ]->wait_queue.front();
36
37              // Make sure the next one has meaningful work to do before being scheduled.
38              // If not, check the next one in the queue repeatedly until all are checked.
39              while( npid != pid && ( prev_ring_map[ npid ]->occupancy < 32 || \
40                                      next_ring_map[ npid ]->occupancy > 2016 ) ) {
41                 core_sched[ i ]->wait_queue.pop();
42                 core_sched[ i ]->wait_queue.push( npid );
43                 npid =  core_sched[ i ]->wait_queue.front();
44              }
45
46              // New pid ready to be run.
47              if ( npid != pid ) {
48                 // Instruct the timer to expire immediately.
49                 // This prevent race condition for the 'expired' variable.
50                 rte_timer_stop_sync( &core_sched[ i ]->timer );
51                 core_sched[ i ]->expired = false;
52
53                 // Put pid to waiting state and npid to runing state
54                 Switch( pid, npid );
55                 timeout = ( uint64_t ) ( hz *  (float) \
56                                          share_map[ coreid ][ pid_to_idx[ npid ] ] / 1000000 );
57                 // Timer responsible for the new running npid is started.
58                 rte_timer_reset( &core_sched[ i ]->timer, timeout, SINGLE, lcore_id, \
59                                  ExpiredCallback, &core_sched[ i ]->expired );
60              }
61           }
62        }
63     }
64  }
65}
```

Figure 3. UNiS code snippet for the scheduling phase.

3. Evaluation

3.1 Environment

3.11 Hardware Configuration

Our testbed consists of two machines connected *back-to-back* using SFP+ cables. This ensures an isolation of our experiment machines from the rest of the cluster and avoids network traffic interference. In addition, by not using a switch (*back-to-back* connection), we eliminate one additional factor that may affect our experiments. From the two machines, one hosts a traffic generator (*DPDK Pktgen*), while the other hosts µNF and UNiS. The hardware specification for each machine is as follows, an Intel(R) Xeon(R) E3-1230 v3 @ 3.30GHz CPU chipset with *hyper-threading* enabled, a 16GB memory, and a DPDK compatible Intel 10G Ethernet adapter (Intel X710-DA2). The size of L1, L2 and L3 caches are 32KB, 256KB and 8MB, respectively.

3.12 Software Environment

We use DPDK version 17.05.2, released in Sept 2017, on Ubuntu 14.04.5 LTS with Linux kernel version 3.16.0-77-generic. Additionally, the Address Space Layout Randomization (ASLR) kernel feature is disabled to ensure a consistent hugepage mapping between multiple runs of a DPDK *primary process* and *secondary process*. On both hosts, we allocated a total of 4GB hugepages; each hugepage is 2MB. The VNF framework used in our experiments is from our previous work, *µNF*, which is currently under review in SIGCOMM 2018. The NF in this framework is rather lightweight and processes packets in batches with a batch size of 32. *µNF* also comes with a prefetch feature that uses DPDK *rte_prefetch_non_temporal*. Because this feature brings a performance boost in *µNF,* we enabled it and set the prefetch size to 8. This means 8 packets from the next batch are fetched before processing the batch, and the 9th packet is fetched while looping and processing the first packet and so on. The ring size connecting NFs is 2048 *mbuf* (one mbuf, one packet). The size of the *mbuf headroom* is 128 bytes and the size of *mbuf dataroom* is 2176 bytes. The packet data is stored in the *dataroom,* whereas the *headroom* is for storing some application related metadata.

3.2 Experiments

Firstly, we conducted some basic experiments to demonstrate the poor performance of Linux scheduler classes, followed by a series of experiments comparing the default Linux Scheduler, an existing intrusive core sharing mechanism in μNF, and UNiS scheduler.

Figure 4 shows the processing cost of the Macswapper NF used in our experiment. To simulate a real-life scenario, an *artificial load* can be added. One unit *load* in the x-axis of the graph translates to 1000 * *load* times non-compiler-optimised computations of two 32-bit integers multiplication. The processing cost is measured with two different methods. The first method obtains a timestamp using the *time.h* C library. The second approach uses Intel's RDTSC instruction sets to read the CPU cycle counter from its register directly. Overall, both approaches give similar and consistent results. Although we expect that the increase in load is proportional to the increase in processing cost, it seems that other factors, e.g., cpu-optimization, prefetching, reading and writing of packets from rings, introduce some influences.

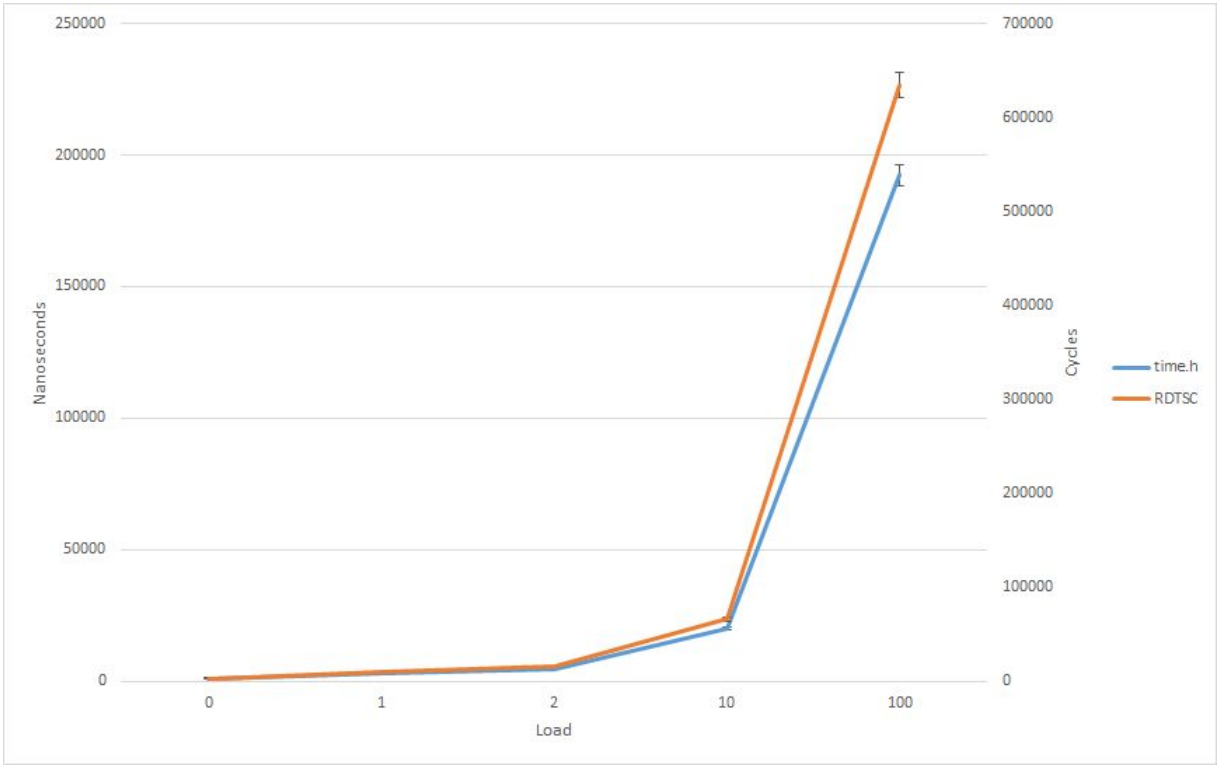

Figure 4. Processing cost of a MacSwapper with an extra load

3.2.1 Baseline

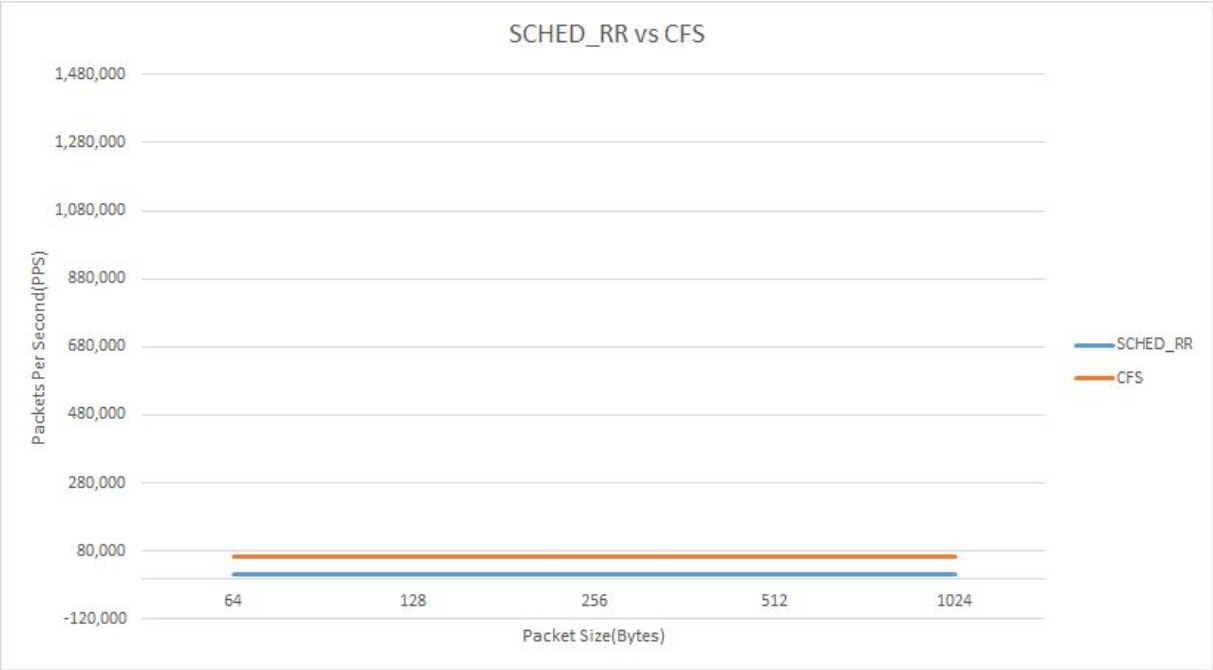

Figure 5. Performance of default linux scheduler with core-sharing NF workload

In this experiment, we demonstrate the performance of an NF chain with some default Linux schedulers. The NF chain used here consists of three *MacSwappers*. *MacSwapper* is a simple NF that swaps the source and destination MAC addresses inside a packet. In this experiment setup, two MacSwappers are pinned to the same core and the third one has a dedicated core. The third MacSwapper is mainly responsible for transmitting packets to the NIC and it requires more CPU resources. We place it on a separate logical core to prevent it from being the bottleneck in the chain.

Figure 5 clearly indicates that the throughput of the system for both SCHED_RR and CFS are terribly low and even far from the 1 Gbps line rate (1.488 mpps). CFS performs poorly because the run time for each process (which is proportional to its weight divided by the total weight of all processes) is still too long; this causes significant packet drop due to an overflow in the ring connecting the NFs. For the SCHED_RR case, the poor throughput is also due to the large default timeslice. After the egress ring is full or the ingress ring is empty, the NF does not have any meaningful work to do, but is still busy-looping. Therefore, a large portion of CPU cycles are wasted and the performance is poor.

3.2.2 Performance Comparison

For the comparison experiments, we fix the packet size to 64 bytes and vary the NF chain length. All of the NFs used in this set of experiments are MacSwappers with different artificial loads. The first experiment, shown in Figure 6, compares UNiS with the original μNF yielding mechanism without adding any artificial load to the NFs. Both schedulers are able to reach a 10Gbps line rate when there are fewer than six MacSwappers sharing one core. However, UNiS has slightly lower performance as more MacSwappers are instantiated to share the same core. This is a rather unexpected result and the root cause has not yet been confirmed. One hypothesis is that solely doing MAC swapping is too light of an operation; there might be other factors that influence the system.

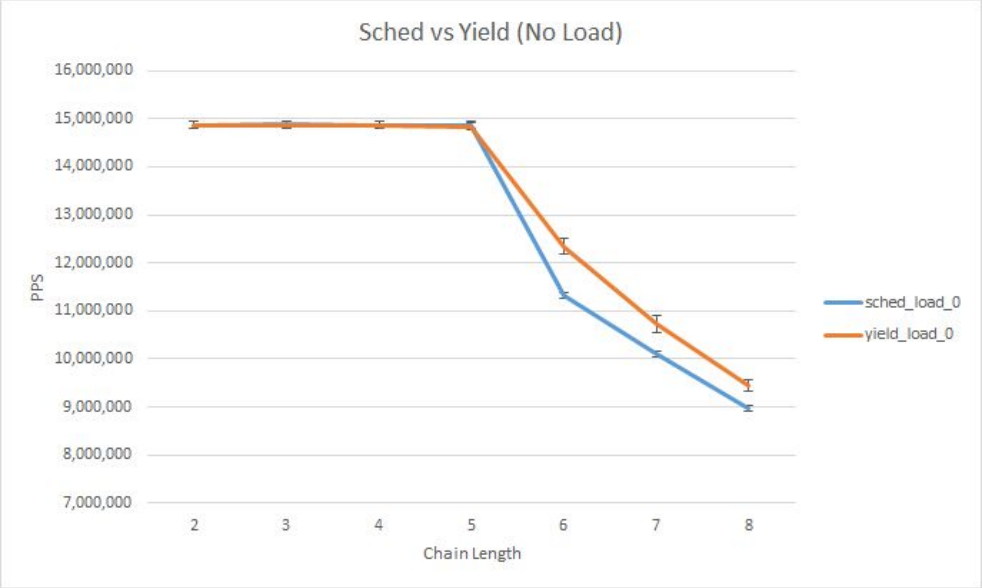

Figure 6. Intrusive μNF and UNiS performance without additional load

To simulate a real world NF scenario, we add one unit load to each NF. The performance of UNiS, as shown in Figure 7, is similar to the intrusive μNF scheduling mechanism. However, one might expect more performance boost since UNiS provides an execution order guarantee. We will investigate further whether the existing share core mechanism in μNF has guaranteed the order, or the L1 L2 caches are just too small.

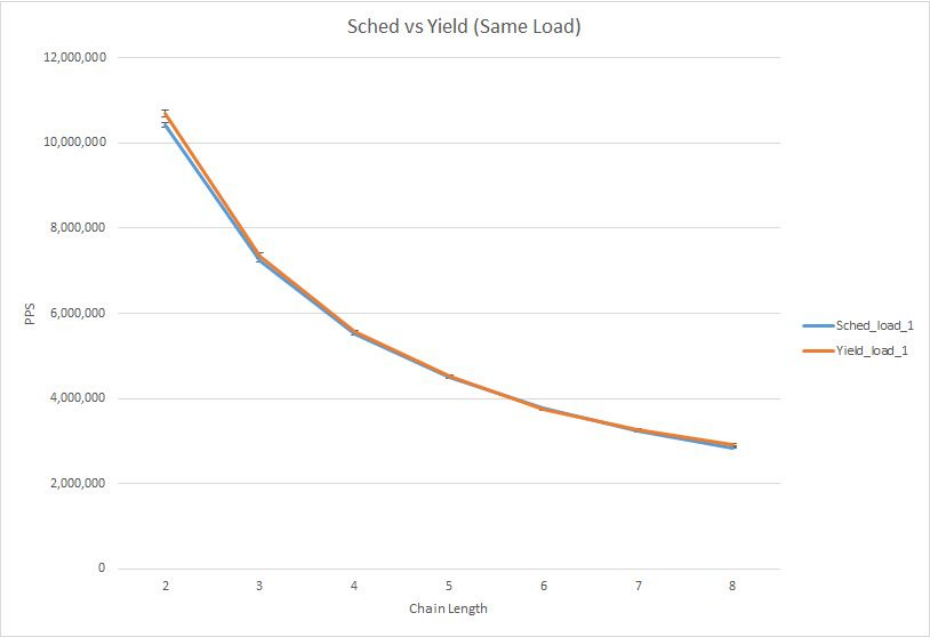

Figure 7. Intrusive µNF and UNiS performance with uniform load

In the third experiment, we adjust the workload of each MacSwapper to simulate an unbalanced workload scenario. Specifically, each newly spawned MacSwapper is assigned one unit or two units of loads repeatedly. The throughput is plotted in Figure 8. Similar to Figure 7, the UNiS performance and native µNF scheduling are very close; the two lines are overlapping. Although the current UNiS does not outperform the native one, some optimizations or additional features discussed in the Future Work section can be added to improve UNiS.

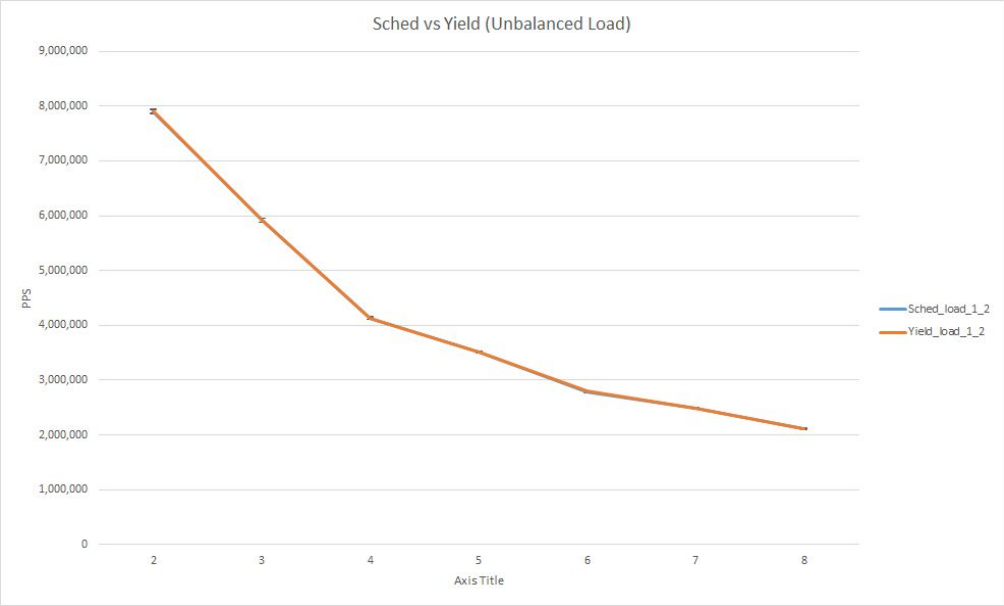

Figure 8. Intrusive µNF and UNiS performance with unbalanced load

4. Limitations and Future Work

From the efficiency perspective, the current implementation of UNiS is still not very resource efficient. UNiS occupies one core and its *rte_timer* subsystem fully uses the core for the time accounting work. This high-overhead could be alleviated by implementing UNiS as a kernel module. However, it requires much more effort to develop and test a kernel module.

Moreover, we foresee the next version of UNiS will be capable of handling dynamic load changes. It should be able to dynamically adjust the share for each process depending on its real time workload. To achieve this, UNiS Cycle Estimator should be able to measure NF processing time on the fly and detect the load change immediately. Alternatively, a reactive algorithm could be devised to adjust NF timeslices dynamically as the load varies. Both of the approaches require a formula and a mechanism to calculate the new timeslices to be assigned to the NFs. Furthermore, since there are usually more NF processes than the available cores, packing these processes into fewer cores considering each NF processing load and other hardware specification is one of our future research projects.

During our experiment, we also observe that our *sched_priority* process controller stops working when the timeslice is relatively long (e.g. larger than 32 microseconds). The cause of this is inconclusive and requires further investigation. Lastly, a platform specific challenge is related to the prefetch-and-process model in μNF platform. This model causes the processing cost estimated by Cycle Estimator to not be precisely equal to the actual processing cost when a chain of NFs processes a continuous stream of packets. A further action to improve the Cycle Estimator module in UNiS is required. One possibility is to quantify and correlate the processing cost offset with the prefetch size.

5. Related Work

First, we look at literature dated back to 1999 [5] that delves into the topic of workconserving and non-work conserving packet scheduling. The scheduling techniques (*Earliest Deadline First,* and its variants) mentioned in the paper are targeted towards specialized network applications that often have a *rate controller* and

are often more concerned about buffer requirements, delay jitter and average delays. Comparing with UNiS, this paper focuses on the theoretical aspects, instead of the implementation aspects. The proposed scheduling technique is argued mathematically and is only evaluated using a simulation. However, the notion of *earliness* and *delay jitter* in the paper provides some helpful insights on improving UNiS once it reaches a stable state or if a stricter QoS is required.

Since NF scheduling is performance sensitive, we try to avoid some overheads already mentioned in the paper published in CAN'17 [6] while designing UNiS. The paper shows various overheads on several different situations, such as crossing NUMA boundaries and cache misses. Core utilization and different levels of cache hit ratio are the metrics evaluated in the paper. As they briefly suggest in the paper, these metrics can be used to optimize NF placement. This is closely related to one of our next steps to solve the problem of placing/packing NF processes into fewer cores.

Next, *A Processor-Sharing Scheduling Strategy for NFV Nodes* [2] is a study published in the 2016 Journal of Electrical and Computer Engineering. It proposed *Network-Aware Round Robin (NARR),* a scheduling algorithm aiming to reduce delays in traversing SDN/NFV by assigning more cpu slices to the VNFs whose NIC queues are less loaded. In addition, some work in the area of inter-VM scheduling [3] and green SDN/NFV [1] are also somewhat related to UNiS. The former assumes that NFs runs as VMs and changes are made to the hypervisor. The latter concerns most about energy consumption instead of performance.

NFVNice [4] is a state-of-the-art paper in VNF scheduler area which proposes a management framework to provide an efficient and dynamic resource scheduling for NF processes on NFV platforms. This framework constantly monitors all VNF load information via *libnf* and adjusts CPU share of each VNF accordingly. Moreover, it has a back-pressure mechanism that gives more share to NFs that have more packets in the input queue. However, NFVNice requires a rewrite of NF source code to use *libnf* to do some basic operations, for example, reading and transmitting packets. Therefore, this is considered as an intrusive scheduling mechanism, similar to μNF.

6. Conclusion

To the best of our knowledge, UNiS is the first non-intrusive NF scheduler. UNiS does not require NF code to be modified/written with a specific library. Therefore, it allows different NFs from different developers to be scheduled together. Moreover, UNiS segregates the scheduling logic out from the packet processing logic in NF development. The current state of UNiS is already able to sustain the same or marginally lower throughput compared to the intrusive mechanism in μNF. Two unexpected, yet interesting, observations about the slight performance difference in *no load* case, and also that UNiS does not perform better (even after guaranteeing the execution order) are mentioned in Section 3. We will investigate this. We will also continue working on UNiS to include support for dynamic timeslice adjustment since it is common that a NF complexity changes based on the type of traffic at that time. UNiS is implemented in C++ and the source code is on our github private repository https://github.com/anthonyaje/micro-nf-datapath/tree/cs798. A demo video is available at https://youtu.be/y-MIWlH8dss.

7. References

1. Faraci, G., & Schembra, G. (2015, 09). An Analytical Model to Design and Manage a Green SDN/NFV CPE Node. IEEE Transactions on Network and Service Management, 12(3), 435-450. doi:10.1109/tnsm.2015.2454293
2. Faraci, G., Lombardo, A., & Schembra, G. (2016). A Processor-Sharing Scheduling Strategy for NFV Nodes. Journal of Electrical and Computer Engineering, 2016, 1-10. doi:10.1155/2016/3583962
3. Guan, B., Wu, Y., Ding, L., & Wang, Y. (2013, 05). CIVSched: Communication-aware Inter-VM Scheduling in Virtual Machine Monitor Based on the Process. 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. doi:10.1109/ccgrid.2013.105
4. Kulkarni, S. G., Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K. K., Wood, T., . . . Fu, X. (2017). NFVnice. Proceedings of the Conference of the ACM Special Interest Group on Data Communication -

SIGCOMM '17. doi:10.1145/3098822.3098828

5. Liebeherr, J., & Yilmaz, E. (n.d.). Workconserving vs. non-workconserving packet scheduling: An issue revisited. 1999 Seventh International Workshop on Quality of Service. IWQoS'99. (Cat. No.98EX354). doi:10.1109/iwqos.1999.766500
6. Sieber, C., Durner, R., Ehm, M., Kellerer, W., & Sharma, P. (2017). Towards optimal adaptation of NFV packet processing to modern CPU memory architectures. *Proceedings of the 2nd Workshop on Cloud-Assisted Networking - CAN '17*. doi:10.1145/3155921.3158429
7. Hwang, J., Ramakrishnan, K. K., & Wood, T. (2015, 03). NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. IEEE Transactions on Network and Service Management, 12(1), 34-47. doi:10.1109/tnsm.2015.2401568
8. Zhang, W., Liu, G., Zhang, W., Shah, N., Lopreiato, P., Todeschi, G., . . . Wood, T. (2016). OpenNetVM. Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization - HotMIddlebox '16. doi:10.1145/2940147.2940155
9. Panda, A., Han, S., Jang, K., Walls, M., Ratnasamy, S., & Shenker, S. (2016, November). NetBricks: Taking the V out of NFV. In OSDI (pp. 203-216).
10. Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K. K., & Wood, T. (2016, December). Flurries: Countless fine-grained nfs for flexible per-flow customization. In Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies (pp. 3-17). ACM.
11. https://github.com/srcvirus/micro-nf (private repository)