Bubble: Lightweight Core Sharing in NFV

Haiping Wang[†], Zhilong Zheng[†], Chen Sun[†], Jun Bi[†]
[†]Institute for Network Sciences and Cyberspace, Tsinghua University
[†]Department of Computer Science, Tsinghua University
[†]Beijing National Research Center for Information Science and Technology (BNRist)

***Abstract*—Many researches have revealed the requirement of enabling multiple network functions (NFs) to share a CPU core in Network Function Virtualization (NFV) to support fine-grained NF models, efficient resource utilization, and chain consolidation. However, these works usually enable core sharing via kernel-level threads, which incurs significant performance degradation. In this paper, we present `Bubble` to enable lightweight core sharing in NFV. `Bubble` leverages user-level threads to eliminate the performance overhead introduced by kernel-level thread scheduling. `Bubble` is designed to satisfy unique requirements in NFV by providing accurate and low-overhead scheduling, in support of on-demand resource allocation and accurate NF load measurement. Evaluations over a `Bubble` prototype implementation demonstrate that `Bubble` can improve the performance by 1.6× to 6.2× for co-located NFs and by 3.7× to 68.8× for a consolidated Service Function Chain (SFC) in a core against two state-of-the-art solutions.**

I. Introduction

Network Function Virtualization (NFV) was introduced to address the limitations of traditional proprietary middleboxes. In NFV, Network Functions (NFs) run on commodity servers with software-based implementation, which brings easier operation, lower costs, and resource sharing between NFs [1]. A NF in NFV is usually deployed in Virtual Machines (VMs) [2], containers [3], or as native processes [4], with one or more *dedicated CPU cores*. Several NF instances are often deployed in the same server to share cores resources. Without CPU contention, this kind of sharing could fully exploit server processing power and achieve high performance for all NFs.

Some researches [5]–[8] reveal that enabling a NF to monopolize one or several CPU cores may incur low core utilization efficiency. As a result, these works enable *core sharing* by co-locating multiple NF instances in the same core. For example, modularized NFs in OpenBox [7] and flow-level NFs in Flurries [6] require fine-grained resource allocation within a single core. NFVNice [5] places multiple NFs in one core and performs intra-core scheduling to avoid wasting resources [5]. CoMB [8] even consolidates a chain of NFs in the same core to avoid cross-core overhead.

However, deploying multiple NFs in one core incurs significant performance degradation [3], [5], [6]. As we will show in §II-B, when using the default Linux kernel thread scheduler (i.e., CFS), the performance (throughput) of a service function chain (SFC) with three firewalls is almost two orders of magnitude lower than the non-sharing deployment [3]–[5]. Our key observation is that the fundamental causes of the performance drop are ill-suited scheduling of OS kernel as well as costly context switch and kernel call [3], [5]. A state-of-the-art NFV scheduler [5] tries to enhances default CFS by translating NFs demand into resource allocation decision. Nevertheless, it still bases designs on kernel threads scheduling, in which case frequent and costly context switches and kernel calls are inevitable. According to recent researches [9], it costs 5-20 μs per context switch or system call, which introduces severe performance overhead and cannot efficiently support core sharing in NFV.

Therefore, in this paper, we aim to enable *efficient core sharing* with *low additional performance overhead*. Our key idea is to use *user-level threading* instead of conventional kernel-level threading for core sharing. User-level threading has the potential advantage of negligible context switch overhead and complete avoidance of costly kernel calls.

Actually, user-level threading is not a new concept and has been proposed for many years, such as `Goroutines` in Go [10], `uThreads` in C++ [11], and some recent efforts [12], [13]. However, supporting user-level threading for NFs is non-trivial due to three requirements that must be satisfied in NFV: (1) The scheduling mechanism must be able to provide fine-grained CPU resource allocation with minimal scheduling overhead. (2) The requirement of Service Level Objects (SLOs) or Quality of Service (QoS) [1], [14], traffic variation, and chaining requirements for NFs incur unique challenges of allocating CPU cycles adaptive to diverse demand of NFs. (3) Elastic scaling for NFs imposes the requirement of real-time NF load detection, which is hard to be supported by current user-level threading systems.

We address above challenges in `Bubble`, which provides user-level threading for lightweight core sharing in NFV. For accurate and lightweight scheduling, `Bubble` adopts frequent CPU yielding and batch size controlling which extremely decreases scheduling overhead. To capture diverse demand of NFs, `Bubble` models the scheduling problems taking consideration of NFs characteristics to find the optimal placement and CPU resource allocation. Moreover, `Bubble` automatically embeds load measurement code to enable load detection without burdening the developer. In summary, we make the following major contributions in this paper:

- We reveal that kernel-level threading could incur significant NF performance degradation when sharing cores (§II-B), and present the design challenges of introducing user-level threading in NFV (§II-C).
- We present `Bubble` to provide lightweight core sharing for NFV with user-level threading and introduce three

key designs (§III) and the implementation (§IV).
- We perform evaluations on a prototype implementation of `Bubble` and demonstrate that `Bubble` can improve performance by 1.6× to 6.2× for co-located NFs and from 3.7× to 68.8× for consolidated SFCs against two existing solutions, and can effectively support performance isolation and accurate load detection for NFs (§V)

II. Motivations and Challenges

This section starts by introducing the requirements of sharing CPU cores when deploying NFs. Next, we illustrate the problems of existing solutions. Finally, we present the challenges in enabling efficient user-level threading in NFV.

A. Requirements of Core Sharing

Despite some NFV frameworks advocate deploying NFs on dedicated CPU cores, some research works [5], [6] reveal that sometimes it is inevitable to share a core between NFs. We conclude the requirements for core sharing as follow.

Fine-grained NF models: Regarding the granularity of implementation, NFs can be monolithic [3], modularized [7] or flow-level [6]. For fine-grained NF models, the number of active NFs may far exceed that of cores in a NFV system. For example, in Flurries [6], each lightweight NF only processes one flow and over 80,000 NFs will run on a single server per second. Thus, core sharing becomes inevitable.

Efficient resource utilization: NFs are deployed with dramatically diverse performance requirements [15]. Traffic volume of different NFs may also change during runtime. The variation in CPU demand enables network operators to densely deploy multiple NFs on a machine to efficiently use CPU. For example, NFVnice [5] places multiple NFs on sharable cores according to their demand for CPU cycles.

Service chain consolidation: In most networks, the delivery of end-to-end services often requires various NFs that form SFCs. Many systems prefer to deploy a SFC on one core [8]. This consolidation ensures that one packet is processed entirely on a single core, which avoids the performance overhead from inter-core communication.

B. Problems of Existing Solutions

Driven by the above requirements, a straightforward solution is using threading management system, such as `pthreads` and `std:thread`, to create many NF threads in one CPU core. For example, existing researches such as NFVNice [5] and Flurries [6] adopt `pthreads` to spawn a bunch of NF instances. However, these instances (threads) are created and scheduled by the OS kernel, which incurs severe performance problems due to two major reasons.

R1: The kernel scheduler cannot determine the right time to schedule a NF thread. To avoid wasting CPU cycles on packet waiting, the optimal scheduling is to wake a NF up when several packets are ready in its receiving buffer (e.g., `packet ring`), and sleep a NF when its buffer is empty. However, existing kernel thread scheduler such as Linux CFS [16] does not care about the above buffer status. In addition, its smallest scheduling timescale is 1 *ms*, which is at least one order of magnitude larger than the 10s *us* processing time of a batch of packets in a NF [3]–[5]. Such coarse-grained scheduling can often schedule a NF for execution at the wrong time (i.e., empty buffer), which could waste at least 10 × more CPU cycles than a right one. Finally, when scheduling NFs in a SFC, NFs should be executed in sequence to correctly process packets. However, existing scheduling mechanisms cannot schedule threads with a specific order.

Scheduling Policy	R1?	R2?	Throughput (Mpps)
CFS	√	√	0.23
CFS & `sche_yield`	×	√	2.93
Bubble	×	×	13.92

TABLE I: Throughput (64B packets) when a three-firewall chain shares a CPU core.

As shown in Table I, we measure the performance of a SFC with three firewalls with CFS. Results show that CFS can only process 0.23M packets per second, which is significantly lower than the non-sharing methods. Some researches [5] proposed a centralized manager to signal NFs to voluntarily yield CPU, such as invoking the system call `sche_yield`. Table I shows that this approach (i.e., CFS & `sche_yield`) is able to improve throughput to 2.93 Mpps. One major reason for this improvement is that it tries to stop a NF from execution if no packet is ready. Nevertheless, the performance still falls far behind the ideal one.

R2: Kernel scheduling requires costly context switches and kernel calls. Scheduling user-level applications (e.g., NFs) via kernel scheduler will introduce context switches and kernel calls, which usually cost 5-20 μs per operation [9]. Therefore, as shown in Table I, even with an improved kernel-based scheduling policy (CFS & `sche_yield`), its performance is still 4.75× lower than `Bubble`, which is our scheduling mechanism without context switches in kernel.

`Bubble`. Motivated by the above problems, we design `Bubble` to support core sharing with a marginal performance overhead. The basic idea of `Bubble` is adopting *user-level threads scheduling*. `Bubble` could find the right time to schedule NFs and avoid costly context switches and kernel calls since the scheduling is highly controllable in user space.

C. Challenges of Designing User-level Threading for NFV

We face three major challenges when introducing a user-level thread management system into NFV.

Accurate and low-overhead scheduling: A NFV system needs to support Service Level Objects (SLOs) or Quality of Service(QoS) for a NF [1], [14], which is usually based on performance isolation. However, existing user-level thread systems fail to guarantee that because their design philosophy is to avoid complex scheduling operations (e.g. preemption and priority support) to keep threading as lightweight as possible [13]. Some compromise methods were proposed. For example [12] enables *timer-based scheduling* that is built on a high-performance timer (i.e., Time Stamp Counter (TSC)), to control the execution of a user-level thread. However,

frequent timer checking could induce significant performance overhead, which will overshadow the performance benefit from user-level threading. Arachne [13] groups cooperated user-level threads to various sets of CPU cores (with a different number of cores), to prioritize threads without adding scheduling overhead. However, it cannot guarantee the CPU cycle allocation in the same set among threads. Shortly none of them satisfies the scheduling goal of NFV systems. Therefore, we are challenged to design an accurate and low-overhead scheduling mechanism for NFs.

Varied Demands from NFV deployment: When deploying services in a NFV network, there could be several demands that should be satisfied, such as time-varying traffic, different SLO or QoS requirements, and dynamically changed chaining requirement of SFCs. This variety imposes the challenge that our scheduler must allocate CPU resource according to specific demand of each NF and schedule the right NF to execute at runtime.

Valid core utilization measurement for elastic scaling: Elastic scaling requires accurate core utilization measurement for timely NF scaling. Current user-level threading systems such as Arachne [13] accumulate the function execution time of short-lived tasks (microsecond timescale) as a load indicator. However, in NFV, there exists a lot of long-lived NFs, some of which may leverage *busy polling* for high performance. In this case, the accumulation approach could regard polling as counted cycles, which could cause that even no packets are processed, it always returns non-zero utilization (about 20% in our evaluation). Therefore, we are challenged to design core utilization measurement for NFs.

III. Bubble Design

In this section, we propose three major designs of our user-level threading system `Bubble` and present the formulations of the optimal scheduling problem.

A. Key designs

`Bubble` supports running multiple NFs on sharable cores. It is based on the following key designs.

Accurate scheduling with frequent CPU yielding after processing a batch of packets. As analyzed in §II-C, scheduling should be accurate and lightweight. To achieve this, we leverage the *batching mechanism* that is usually adopted in packet processing to support high-capacity processing [17]. Specifically, a typical packet processing workflow in a NF could be summarized as: reading packets from NICs, batching several packets, processing this batch, and sending out these processed packets. Basing that, we propose *frequent CPU yielding*, which implies a NF to yield the CPU core after processing a batch of packets. Meanwhile, we control *batch size* to implement resource allocation. This indirect allocation can benefit us from, (1) avoiding monitoring and checking CPU cycles usage of all user-level threading NFs since their resources are allocated on-demand via batch size, and (2) more accurate scheduling because the execution time of each thread becomes easily predictable with batch size.

Scheduling decision varied with dynamic deployment requirements. As mentioned in §II-B, it is important for the scheduler to decide when a NF should get or relinquish CPU. `Bubble` makes co-located NFs use CPU in order, where chaining NFs are scheduled as with the sequence of service function chains. This relieves the job of scheduler to dynamically select next NF to run. Further, `Bubble` formulates the problem that how much CPU resource a NF can get, by carefully analyzing dynamic traffic, QoS requirement, NF processing complexity, and chaining requirements. Therefore, `Bubble` translates the scheduling decision into the batch size to control resource allocation.

Core utilization measurement with automatically embedded code. A straightforward approach to decide the load of a long-lived busy-polling NF is adding measurement code before and after the packet processing, and count the cycles spent on processing (denoted as pkt_cyc). A monitor should be used to obtain the cycle count and accumulate in a time period. In this period, the total cycles can be measured as $total\_cyc$ and cycles cost by a NF is accumulated to $\sum pkt\_cyc$. Thus, the core usage of a NF can be calculated as $\sum pkt\_cyc/total\_cyc$.

We follow the above approach in `Bubble`. However, it is stressful and error-prone for operators to manually add measurement codes. Moreover, hundreds of NFs run measurement codes simultaneously (e.g. by calling `time` system call) could heavily load the system. Instead, we leverage the fact that NFs in most NFV frameworks (e.g., Netbricks [4] and OpenNetVM [3]) are linked to specified libraries for packets processing and forwarding. We could modify the libraries to automatically support measurement. Moreover, we reuse the scheduler to monitor the load, which is enabled only when measurement is required. Finally, we use CPU's TSC [18], which only costs a single CPU instruction from user space.

B. Formulation of the Optimal Scheduling Problem

To provide accurate scheduling, `Bubble` should make right decision about when and how long a NF will be scheduled to run. We start our formulation by analyzing the time of a batch of packets delayed in the NFV system. The time of packets staying in NFs can be split as waiting for former co-located NFs and being processed to send out.

We use a linear model to calculate the processing time, as most batch-processing systems [19]:

$$T_{pi} = k_i \cdot batchsize_i + c_i \ (+cost) \tag{1}$$

Parameter k_i and c_i are determined by the processing complexity of each NF. Since NFs in SFCs would suffer cache invalidation from shared state accessing in cross-core communication [8], we use the term $cost$ to capture the overhead of cross-core placement if any. Then the waiting time can be calculated by summing up the processing time of former co-located NFs. For simplicity we ignore the scheduling overhead, since it only contains less than three operations (as shown in Algorithm 1 later).

Then we formulate the optimal scheduling problem as deploying as much NFs as possible without violating their SLO requirements. Given a set of NFs F, where NFs are numbered from 1 to n and associated with a traffic rate r_i and a priority weight w_i (to enable support for SLOs or QoS). Suppose that the set of cores available on the system is denoted as C, where cores are numbered from 1 to N.

We introduce a binary variable $x_{iJ} \in \{0, 1\}$ to indicate whether NF $i \in F$ is deployed on core $J \in C$, and an integer variable b_i to indicate the number of packets NF i scheduled to run with on each round. Hence, we set the objective as maximizing the total weighted throughput of all NFs in a system, which is formulated as,

$$max \ \sum_i (\sum_J x_{iJ}) \cdot w_i \cdot \frac{b_i}{T_i} \tag{2}$$

s.t.

(1) $T_i = T_{pi} + \sum_{j \neq i} T_{pj}, \ \forall j \in F, x_{iJ} \cdot x_{jJ} = 1$

(2) $b_i \leq r_i \cdot \sum_{x_{iJ} \cdot x_{jJ} = 1} T_j, \ \forall i, j \in F, \ \forall J \in C$

(3) $\sum_J x_{iJ} \leq 1, \forall i \in F, \ \forall J \in C$

(4) $\sum_J x_{iJ} \cdot \sum_K x_{jK} = 1, \ \forall i, j \in F, chain(i) = chain(j), \ \forall J, K \in C$

More specifically, Constraint (1) specifies the total time of a batch of packets spent in the NFV system. Constraint (2) limits the scheduler to set batch size less than the number of packets already buffered in NFs' queue. This enables NFs not to wait for packets and waste CPU cycles. Constraint (3) prevents any NF from being deployed redundantly. Constraint (4) specifies the integrity of a SFC, i.e., NFs within a chain should be either accepted or rejected altogether.

To find solutions on $\vec{x}$ and $\vec{b}$ for the optimization objective (Eqn. 2), we need to provide parameters k_i, c_i and $cost$. Since these parameters are strongly related to NFs' characteristics, we adopt *profiling method* to learn that for every type of NFs. Firstly, we deploy NFs in a cross-NUMA way to capture parameter $cost$. To learn the other two parameters, we run each NF with a dedicated core, during which we vary its processing batch size and record the throughput. Then we can adopt linear regression model to learn k_i and r_i from data examples. Finally we can use the mathematical tool MATLAB [20] to find solutions for Eqn. 2.

However as mentioned in § II-C, resource allocation is required to consider traffic variation. Running the optimization solver for every scheduling decision is too slow to high-speed NFs. Therefore, we propose a two-step lightweight but accurate scheduling algorithm as shown in Algorithm 1. The optimal placement decision is made by the optimization solver in the first step, and dynamically adapted to traffic variation by adjusting processing batch size in the second step. Algorithm 1 takes deployment requirements and parameters profiling results as inputs. In the first step (line 1),

Algorithm 1: Scheduling algorithm

```
input  : F, C, {SFC}, k⃗, c⃗, cost, K⃗ - NFs set, cores set,
         chaining requirements, and profiling results.
output : (F⃗_J, b⃗) - placement result of cores, and batch size of NFs
1  Initialization: solve Eqn. (2) and denote the placement and batch
     size in the optimal solution as F⃗_J and b⃗_0, with input of
     r⃗ = [r_0], F⃗_J = [φ], 0 ≤ b_i ≤ 1024, ∀i ∈ F
2  while at runtime do
3      foreach local core scheduler J ∈ C at time slot t do
4          if utilization > μ then
5              inform the manager to scale co-located NFs
6              break;
7          foreach i ∈ F_J do
8              Δ = r_i^t − r_i^{t−1} // Calculate rate
                 variation
9              if Δ ≥ ε then
10                 b_i^t = b_i^{t−1} · α // Add fast
11             else if Δ ≤ α then
12                 b_i^t = b_i^{t−1} − β // Decrease slowly
```

the optimization solver is run once to decide which CPU core each NF to deploy on. More specifically, the optimal solutions of b_i may be very large or not fit well in practice due to the unbounded value range of b in our formulation. To make the calculated batch size more practical, we empirically limit batch size with a lower bound and an upper bound and constrain the batch size to be integer multiple of 16 (setting them as 0 and 1024 in our current implementation). The second step is carried out at runtime. The main policy of adjustment is to add processing batch size when traffic rate increases, and reduce it when the rate decreases. Specifically, we adopt a quick adding and slow reducing adjustment strategy, which makes the scheduler more sensitive to increase CPU cycles for NFs to guarantee performance. The factors of adding rate α and the factor of reducing rate β (line 7-12) can be set based on the sensitivity requirement on the scheduler. Periodically the scheduler will detect the CPU load variation and informs the manager to scale as soon as it reaches the threshold μ (line 4-5).

IV. Implementation

Fig. 1 shows the implementation details of `Bubble`. Guided by the design approaches introduced in §III, three modules work together to provide lightweight user-level core sharing for NFV. The *Manager* is responsible for making NFs placement decisions and scaling NFs when cores overloaded. There is a *Local Scheduler* on each core responsible for user-level threads scheduling and scaling detection. The `Bubble` *nflib* is a static library linked to each NF, which mainly provides two interfaces to enable automatic CPU yielding of a NF as well as core utilization measurement.

A. Manager

The Manager runs as a process in user space. It reads NFs configuration and produces the optimal placement solution using MATLAB, as introduced in III-B.

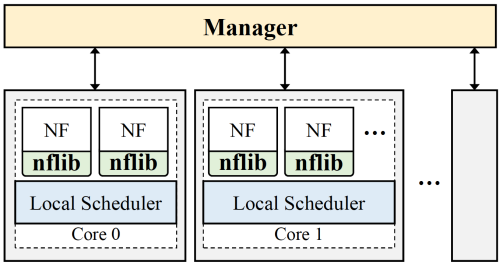


Fig. 1: Architecture of Bubble's implementation.

The placement result are transferred to local schedulers by sockets, following the communication mechanism of Arachne [13]. At begins, the manager launches multiple kernel threads as local schedulers, as much as the number of available cores. We let these schedulers make requests of placement result using sockets, which make them sleep on sockets until the manager replies them. On receiving manager's replies, local schedulers are wakened up to schedule NFs.

During running, the manager periodically checks whether there are scaling requests from schedulers. If any, the manager migrates some of co-located NFs to under-utilized cores.

B. Local Scheduler

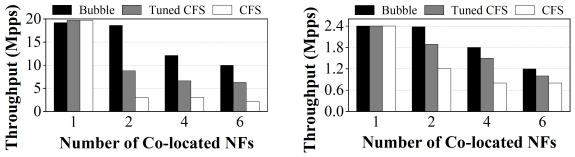
There are multiple local schedulers, each of which is pinned on one exclusive core. The local scheduler is responsible for managing the lifetime of NFs which are implemented as user-level threads. In the beginning, the local scheduler launches assigned NFs according to the scheduling decision of the Manager. Then it serializes the execution of NFs in an order table, where NFs in the same SFC are scheduled with chaining sequence. During running, the scheduler dispatches CPU to NFs according to the order table, and detect traffic variation to adjust the batch size of NFs based on Algorithm 1.

Moreover, the scheduler periodically measures the load of each NF (1 second in our implementation) to detect core overload. At each measurement, the cycle counting code in NFs will be enabled in a short time (30 ms in the implementation). The scheduler accumulates the cycle value of NFs, and sums up all usages in a core as the core utilization. When the core utilization exceeds a threshold value, it informs the manager to scale. NFs migration is lightweight in Bubble. The scheduler only needs to move NFs id from the scheduling table of the origin to that of the destination.

C. Programming Model

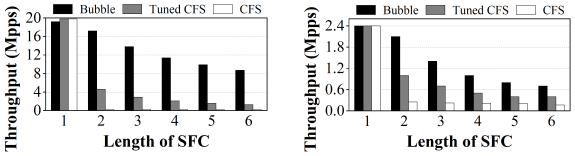
The goal of Bubble nflib is enabling automatic CPU yielding and core utilization measurement for a NF. Like most NF development frameworks (e.g., NetBricks [4], and OpenNetVM [3]) which support their own interfaces to receive and send packets, our nflib also provide them. Moreover, we utilize them to embed yielding and core utilization measurement code, which is transparent and does not burden the developer. Bubble nflib exposes the following two methods:

- `recv_pkts(packets *)` can fetch a batch of packets from a NIC or other NFs (i.e., SFC). The `packets` $\star$ reference to these packets with library related structure. For example, it is `rte_mbuf` in our implementation based on DPDK [21].
- `send_pkts(packets *)` can send a batch of packet to a NIC or other NFs. After completing sending, `sche_yield()` can be automatically invoked to support frequent yielding. Also, if core utilization measurement is enabled, a TSC is obtained to subtract the starting value in `recv_pkt` as the cost processing cycles of this batch of packets.



(a) Performance of firewalls (b) Performance of ipsecs

Fig. 2: Performance of co-located NFs on one core



(a) Chained with firewalls (b) Chained with ipsecs

Fig. 3: Performance of SFCs on one core

V. Evaluation

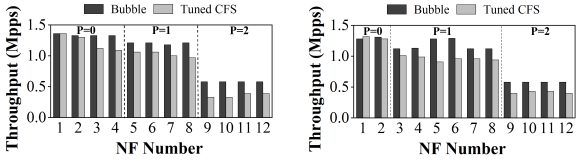
We have implemented a prototype of Bubble based on DPDK lthread [12], which uses DPDK [21] for networking I/O. We have also implemented a low-complexity NF *Firewall* and a high-complexity NF *IPSec*.

Experimental setup: We use two servers which are connected by a 40 GbE switch for evaluation. Each server is equipped with two Intel Xeon E5-2650 v4 CPUs (2.20 GHz, 12 physical cores), 128GB total memory, and a single-port 40G NICs (Intel XL710). Both servers run Ubuntu 14.04 (with kernel 4.4.0-137) and DPDK version 18.02. We use DPDK Pktgen [21] as the traffic generator. The pktgen is configured to generate line-rate traffic with varied-size packets and flows as needed.

Evaluation goals: We demonstrate Bubble's effectiveness with compared to the native Linux kernel scheduler, i.e. CFS, as well as a NFV-tuned kernel scheduler, which we called tuned-CFS [5]. We evaluate Bubble with the following goals: (1) the performance improvement compared with current core sharing solutions; (2) the effectiveness of performance isolation for NFs; and (3) the accuracy of core utilization measurement.

A. Performance Improvement

Co-located NFs share a core. First, we co-locate a varied number of identical NFs (same processing complexity, traffic rate, and QoS requirements) on one core to demonstrate performance improvement. Fig. 2 shows the total throughput of co-located firewalls and co-located ipsecs respectively. As



(a) Priority configuration #1 (b) Priority configuration #2

Fig. 4: NF performance with different priority settings

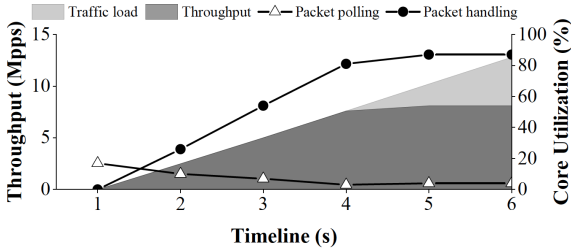


Fig. 5: Measured core utilization with time-varying traffic

we can see, for firewalls, Bubble outperforms performance tuned CFS by 1.6× to 2.1× and CFS by 4.5× to 6.2×. The major reason for this improvement is that Bubble minimizes the overhead of context switches and scheduling.

A SFC shares a core. Then, we consolidate NFs as a chain on one core. Fig. 3 shows that Bubble always outperform other two thread schedulers no matter what NFs are chained. For example, for the firewalls chain, Bubble improves the performance with the scheduling of tuned CFS from 3.7× to 6.7×, and from 51.1× to 68.8× with default CFS. The major reason for this significant improvement is that Bubble could make the right schedules according to SFCs.

B. Effectiveness of Performance Isolation

To demonstrate the effectiveness of performance isolation, we compare Bubble only with tuned CFS, since CFS cannot provide performance isolation. Tuned CFS uses cgroup to control CPU share of each NF based on $share_i = \frac{w_i * \lambda_i * s_i}{total}$ (w:weight, λ: traffic rate, s: service time, $total$: the total load of a core) [5]. We set 12 firewalls with different priorities and allocate 7 cores to schedulers in total. We run experiments on two priority configuration to observe NFs throughput: *#1* which evenly assigns 4 firewalls to priority 0, priority 1 and priority 2; and *#2* which assigns 2, 6, and 4 firewalls to priority 0, priority 1 and priority 2 respectively. As shown in Fig. 4, Bubble and Tuned CFS are able to regulate the performance of NFs according to QoS priorities. But Bubble has higher performance than Tuned CFS.

C. Accuracy of Core Utilization Measurement

To demonstrate the accuracy of core utilization measurement, we use a chain with three firewalls and record the core utilization as an indicator. We vary the traffic rate every second. Fig. 5 shows the results where the line with the black point is the core utilization we measures in Bubble. We can see that the measured utilization in Bubble is almost consistent with the throughput variation. Moreover, the figure shows that if counting start before polling (in *busy polling* mode), the measured utilization is about 20% even no traffic sending to the NFs. Lastly, the measured utilization cannot reach 100% (the maximal value is 91% in this evaluation). The remaining CPU is the slight overhead of Bubble's schedulers, which is acceptable for NFs.

VI. Conclusion

We propose Bubble to enable lightweight core sharing in NFV. Bubble provides user-level threads for NFs, which essentially avoids the costly context switch between user space and kernel space and kernel call. Moreover, it addresses the challenges with accurate and low-overhead scheduling, supporting for various NFV deployment requirements, and core utilization measurement for elastic scaling, to make user-level threading work better in a NFV network environment.

References

[1] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "Resq: Enabling slos in network function virtualization," in *NSDI*. USENI XAssociation, 2018.

[2] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *NSDI*. USENIX Association, 2014, pp. 459–473.

[3] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "Opennetvm: A platform for high performance network service chains," in *HotMiddlebox*. ACM, 2016.

[4] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of nfv," in *OSDI*, 2016.

[5] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "Nfvnice: Dynamic backpressure and scheduling for nfv service chains," in *SIGCOMM*. ACM, 2017.

[6] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained nfs for flexible per-flow customization," in *CoNEXT*. ACM, 2016.

[7] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: A software-defined framework for developing, deploying, and managing network functions," in *SIGCOMM*. ACM, 2016, pp. 511–524.

[8] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *NSDI*. USENIX Association, 2012.

[9] A. Sriraman and T. F. Wenisch, "µtune: Auto-tuned threading for OLDI microservices," in *OSDI*. USENIX Association, 2018.

[10] "The go programming language," https://golang.org/, 2009.

[11] S. Barghi, "uthreads: Concurrent user threads in c++ (and c)," https://github.com/samanbarghi/uThreads, 2015.

[12] "Dpdk l-thread subsystem," https://doc.dpdk.org/guides-16.04/sample_app_ug/performance_thread.html, 2014.

[13] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: Core-aware thread management," in *OSDI*, 2018.

[14] Z. Zheng, J. Bi, H. Wang, C. Sun, H. Yu, H. Hu, K. Gao, and J. Wu, "Grus: Enabling latency slos for gpu-accelerated nfv systems," in *ICNP*, 2018.

[15] ETSI-GS-NFV-002. (2013) Network functions virtualization (nfv): Architectural framework. [Online]. Available: http://www.etsi.org

[16] I. Molnar, "Linux kernel documentation: Cfs scheduler," https://www.kernel.org , 2017.

[17] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing." in *NSDI*, 2017, pp. 97–112.

[18] "Time stamp counter (tsc)," https://en.wikipedia.org/wiki/Time_Stamp_Co unter , 2017.

[19] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-net: Effective gpu sharing in nfv systems," in *NSDI*, 2018.

[20] "Nonlinear optimization," https://www.mathworks.com/help/optim/nonlinear-programming.html , 2019.

[21] Intel. (2012) Data plane development kit (dpdk). [Online]. Available: http://dpdk.org