Buffet: Enabling Multi-Tenant Network Functions

Heng Yu[†], Junxian Shen[†], Chen Sun[†], Zhilong Zheng[†], Jilong Wang[†]
[†]Institute for Network Sciences and Cyberspace, Tsinghua University
[†]Department of Computer Science, Tsinghua University
[†]Beijing National Research Center for Information Science and Technology (BNRist)
{h-yu17, sjx15, c-sun14, zhengzl15}@mails.tsinghua.edu.cn, wjl@cernet.edu.cn

***Abstract*—Many enterprises outsource traffic processing to third-party Network Function (NF) service providers to relieve management burden and reduce cost. NF providers have to process packets from multiple tenants simultaneously. However, most existing software based NFs are designed for one single tenant without internal state isolation mechanisms. These NFs cannot be securely shared across multiple tenants. Existing solutions that support multitenancy are either inefficient or ad-hoc for specific NFs. In this paper, we propose Buffet, a general and efficient framework that enables multitenancy for a wide range of NFs. First, Buffet introduces a general programming abstraction for various NFs to relieve NF developers from considering isolation details. Second, Buffet proposes dynamic tenant-level affinity to achieve high performance and resource efficiency. Finally, Buffet exploits SmartNIC offloading to eliminate host CPU overhead. We have implemented a prototype of Buffet. Evaluation results demonstrate that Buffet can effectively enable multitenancy for a wide range of NFs with high performance and resource efficiency.**

I. Introduction

Network Functions Virtualization (NFV) was recently introduced to address the limitations of traditional dedicated middleboxes. NFV implements NFs on commodity hardware for low cost, high flexibility, and simplified management [1]. Currently, many enterprises outsource traffic processing to third-party NF service providers to completely relieve management burden [2]. Since most of the time, tenants send far less traffic than their peak workloads [3], NF providers could share and multiplex resources across multiple tenants to leverage the economies of scale and reduce cost.

However, most existing software based NFs are designed for single tenant without considering internal state isolation, which severely challenges the capability of NF providers to serve multiple tenants simultaneously. For instance, *ips* is a typical Firewall that stores all ACL rules in a single table [4]. However, different tenants may have unique policies, while flows of different tenants may share the same IP addresses (e.g., internal IPs beginning with *192.168.**). Without isolation, packet processing of multiple tenants can introduce misbehavior and state confusion, such as policies conflict in Firewall and statistics error in Monitor.

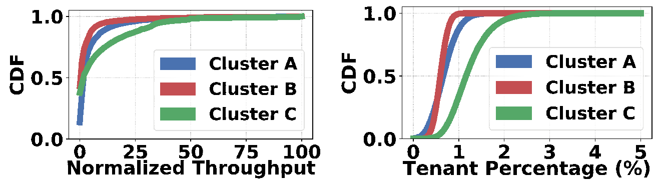
Some research efforts have been devoted to address this problem. A straightforward solution is to allocate separate NF instances running in virtual machines (VMs) or containers for different tenants to provide hardware- or system-level isolation [5], [6]. However, this approach comes with considerable compromises. First, dedicating resources to each tenant violates the resource multiplexing nature of service. Providers have to over-provision resource for every tenant to handle its peak traffic demand [7], which significantly reduces resource efficiency. Second, the usage of virtualization technologies such as VMs or containers introduces significant performance overhead, especially when frequent context switching is needed to multiplex resource across multiple NFs [8]. Some recent researches [7], [9] proposed lightweight *intra-process isolation* to support multitenancy efficiently. Unfortunately, existing works are specialized for specific NFs such as IPSec Gateway [7] and require system re-design to support other NFs. Developers need to repeatedly write low-level tenant isolation and state management logic, which could be time consuming and prone to errors.

Above discussion reveals three design goals for enabling multitenancy for NFs. (1) Providing a *general tenant isolation mechanism* for various NFs to relieve developers from considering isolation enforcement. (2) Achieving *high performance* while guaranteeing isolation. (3) Achieving *high resource efficiency* for time-varying workloads.

To achieve the above goals, in this paper, we propose Buffet, a *general framework* to enable multitenancy for a wide range of NFs. First, we adopt *intra-process isolation* to avoid extra performance overhead and propose a *general programming abstraction* to provide automatic tenant isolation. Second, we propose *dynamic tenant-level affinity* to enable parallel packet processing and active traffic scheduling for high performance and resource efficiency. Third, we leverage SmartNICs [10] to offload schedule processing to avoid wasting host CPUs and improve efficiency. Buffet makes the following major contributions:

- We present the motivation of resource multiplexing across tenants with real world traffic patterns and the requirement of a general framework that enable efficient multitenancy for various NFs (§II).
- We introduce the design of Buffet, including a general programming abstraction to enable automatic tenant isolation, dynamic tenant-level affinity for high performance and resource efficiency, and SmartNIC offloading to avoid host CPUs consumption. (§III).
- We build a prototype of Buffet. Evaluation results demonstrate that Buffet is able to effectively support a wide range of multi-tenant NFs with high performance and resource efficiency (§IV).

978-1-7281-0962-6/19/$31.00 ©2019 IEEE



(a) Throughput distribution (b) Traffic peak distribution

Fig. 1: Traffic patterns

II. Motivations and Related Work

In this section, we illustrate the need for resource multiplexing across tenants by analyzing real world traffic patterns. Next, we summarize the shortcomings of existing solutions and present the motivation of designing a general framework to enable multitenancy for NFs.

A. Why we need resource multiplexing across tenants?

We analyze the traffic pattern from real-world trace, which is publicly available from Facebook [11]. This trace samples from three production clusters and we use the destination IP to identify a tenant [2]. Fig. 1(a) shows the CDF of normalized throughput distribution across tenants. In more than 90% of the time, tenants send less than 50% of their peak workloads. Fig. 1(b) shows the CDF of tenants percentage that reach their maximum workloads at the same time. In all clusters, less than 3% of tenants reach their peak traffic simultaneously and the workloads from multiple tenants have staggered peaks. Furthermore, our analysis results are consistent with existing observations [3], [12].

Above analysis illustrates that to achieve high resource efficiency, NF providers should multiplex resources across tenants, which requires NFs to provide service to multiple tenants simultaneously.

B. Why are existing solutions insufficient?

Virtualization solutions are general but inefficient. Virtualization technologies such as VM and container can isolate states of different tenants [5], [6]. However, they introduce significant performance or resource overhead. If we exclusively allocate resource to VMs/containers and over-provision them for peak traffic demands, most VMs/containers handle far less traffic than their maximum capacity and suffer from significant waste of resource [7]. Over-subscribing physical machines with VMs or containers has been studied as a solution [13]. However, enabling multiple VMs or containers to share one CPU core introduces significant performance overhead due to frequent context switching [8].

Specialized isolation solutions are efficient but ad-hoc. Some researches proposed to exploit intra-process isolation to overcome the performance or resource overhead of virtualization technologies. To support multitenancy, existing efforts add dedicated state isolation logic for different NFs. For example, [7] designed a multi-tenant IPsec Gateway. [9] designed a different architecture for Load Balancer. However, above works are specialized for specific NFs, which cannot be easily extended to support other NFs. Developing a new architecture for every NF to achieve isolation and efficiency is highly repetitive and error-prone.

	Virtualization (Exclusive)	*Virtualization (Shared)*	*Specialization*	*buffet*
Performance	●	○	●	●
Resource Efficiency	○	●	●	●
Generality	●	●	○	●

TABLE I: Comparison of different multitenancy solutions

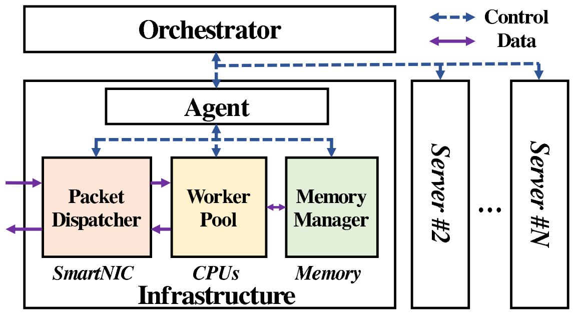


Fig. 2: Buffet framework overview

Buffet. We summarize the benefits and drawbacks of different solutions in TABLE I. Virtualization technologies are general enough but poor at performance or resource efficiency, while specialized solutions are highly efficient but lack generality. Different from all existing solutions, we aim to design Buffet, a general framework that simultaneously achieves isolation, high performance, and resource efficiency for a wide range of multi-tenant NFs.

III. Buffet Design

In this section, we present the design overview of Buffet and details to achieve general isolation, high performance, and high resource efficiency.

A. Overview

Similar to Maglev [9] and Protego [7], Buffet is designed for massive tenants running on a cluster of commodity servers. Each server carries the same NF with multiple CPU cores. Fig 2 shows the framework overview of Buffet, which consists of three major components. First, Buffet *Orchestrator* is responsible for management, such as new tenant admission, NF scaling and failure recovery. Second, a Buffet *Agent* locates inside each server to manage NF and communicate with the Orchestrator. Third, the Buffet *Infrastructure* is specially designed to enable tenant isolation, high performance and resource efficiency.

When a new tenant arrives, the Orchestrator selects a proper server for this tenant. The Agent in the selected server initializes resources for this tenant, and periodically reports server status (e.g., load status) to the Orchestrator. Buffet refers to prior wisdom and adopt techniques such as Split/Merge [14] and OpenNF [15] for NF scaling, and techniques such as FTMB [16] for quick NF failure recovery. In this paper, we focus on the design of Buffet Infrastructure.

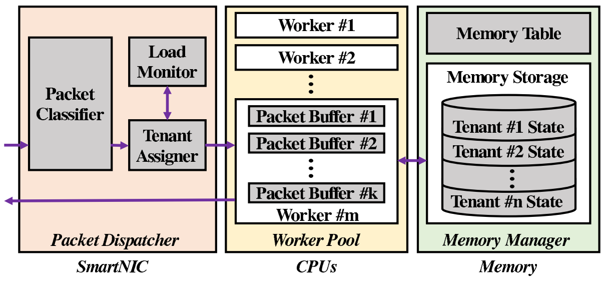


Fig. 3: Buffet design overview

B. General Tenant Isolation

To achieve general isolation with marginal performance overhead, our key design choice is adopting *intra-process isolation*. Based on it, we propose a NF *programming abstraction* for general isolation. Developers only need to provide three functions that are related to NF logic. To support multitenancy, inspired by [17], we decouple NF state and logic and manage the potentially conflicting states with a *Memory Manager*. The stateless logic is executed in a *Worker Pool* and shared by multiple tenants. Based on them, we design a uniform *workflow* for various NFs to enforce lightweight isolation.

Programming Abstraction. Our programming abstraction are designed to abstract details of tenant state isolation away from NF developers. The three functions needed to be specified by a developer are as follows:

- *Memory registration:* Different NFs define their own data structures for states, such as arrays, tables and so on. To store these states, developers need to request a memory region in advance. In Buffet, it is implemented by providing a memory registration function, i.e. `buffet_mem_register`, to declare the required memory size of states.
- *State initialization:* Different tenants have unique policies, such as security rules in Firewall [4]. Therefore, developers are required to provide an initialization function, i.e. `buffet_state_init`, to analyze these policies and initialize states for tenants. This function has two parameters (tenant policy and memory region for this tenant) and is invoked once for every new tenant.
- *Packet handling:* To define the logic of a specific NF, developers are required to provide a packet handling function, i.e. `buffet_pkt_handler`, to process packets and update states. This function also has two parameters (a packet of a tenant and corresponding states of this tenant) and is invoked once for each new packet.

Memory Manager. Buffet Memory Manager manages the memory at the granularity of tenant and ensures that tenants in the same NF only access their own states. As shown in Fig. 3, we design two modules in Buffet Memory Manager including (1) a *Tenant Table* to record the mapping of tenant id (a unique identifier of each tenant) and its corresponding memory region; and (2) a *Memory Storage* to store states of tenants separately in isolated memory regions.

Worker Pool. As shown in Fig. 3, the Worker Pool consists of multiple *workers*. Each worker monopolizes a CPU core and can be dynamically launched or terminated. Once launched, the worker fetches packets from its pre-allocated port queue and processes them according to NF logic.

Workflow. According to [18], most NFs can be logically divided into three basic steps, including *NF initialization*, *packet receiving loop*, and *packet processing*. To enforce tenant isolation based on programming abstraction, we design the Buffet workflow that similar to the above three steps.

During the initialization of an NF in Buffet, it needs to reserve exclusively accessed memory for different tenants. Therefore, the Memory Manager calls the memory registration function to get required memory size of each tenant and automatically reserves separate memory regions. When a new tenant arrives, the Memory Manager needs to allocate memory and initialize state for this tenant. First, it looks up the Memory Table for an unused memory region. Then, it allocates the unused memory region to the tenant and records this mapping in the Memory Table. Finally, it calls the state initialization function to analyze the policies of this tenant and initialize the allocated memory region. After the NF starts up, the workers are burdened with packet I/O from port queues. When a worker receives a packet, it identifies the tenant id of this packet by matching on specified packet fields (e.g., five tuple), looks up the Memory Table to obtain the state of this tenant. Then it calls the packet handling function to process this packet and update state.

C. Packet Dispatching

To achieve high performance in a commercial server, NF service providers often launch multiple workers and run them in parallel [7]. The RSS [19] feature of commodity NICs is often used to balance the load across workers. RSS hashes on specific fields of packets and could guarantee *flow-level affinity* by always assigning packets from a flow to the same worker. However, a tenant is usually defined as a group of users of the same organizational entity [20]. The traffic from/to a tenant could include multiple flows, and there might exist cross-flow states such as tenant-level accounting [21], which can only be updated with *synchronization* such as mutexes. However, synchronization could incur serious performance degradation due to lock contention and frequent cache thrashing. To address the performance challenge, we propose *tenant-level affinity* for workers to enable parallel packet processing by always dispatching packets of a tenant to the same worker.

However, with time-varying workloads of tenants [3], [12], a fixed tenant-level affinity for workers can lead to significant resource overhead. We have to over-provision workers to satisfy their peak workloads, while the actual traffic is less than peak workloads most of the time [7]. Besides, since few tenants reach their peak workloads at the same time, the server could be capable of these tenants but overload on some workers because of load imbalance. To achieve resource efficiency, we design a traffic scheduling mechanism to actively schedule traffic across multiple workers. When some workers become overloaded, we should migrate some tenants to others to fully utilize available resource in a server.

To enforce *dynamic* tenant-level affinity, as shown in Fig. 3, we design a *Packet Dispatcher*, which consists of a *Packet Classifier*, a *Load Monitor*, and a *Tenant Assigner*. The Packet Classifier is responsible for identifying the tenant id of packets, which supports matching on arbitrary fields (Similar to OpenFlow protocol [22]). The Load Monitor is responsible for recording load information of tenants and workers. The Tenant Assigner records the mapping between tenants and workers and enables consistent packet dispatching between them. Besides, it dynamically updates the tenant-worker mappings according to load information to prevent potential overload.

To accommodate as many tenants as possible, the optimal method is evenly re-dispathcing tenants among workers once overload is detected. The equal sum subsets problem is a known NP-complete problem [23] and its time complexity is $O(K^N)$ (K is the total number of workers and N is the total number of tenants). However, Buffet needs fast scheduling. Furthermore, evenly re-dispaching could lead to excessive tenants migration among workers and increase cache misses when accessing states. Therefore, we try to find a algorithm which could (1) find a relatively effective schedule solution, (2) within polynomial-time, and (3) minimize the migrated number of tenants. Finally, we propose a heuristic-based scheduling algorithm based on the observation that most tenants have relatively small throughput and could be accommodated on other underloaded workers. When overload happens, our heuristic algorithm only migrates tenants between overloaded workers and underloaded workers to trade accommodated tenants number for scheduling complexity and migration overhead.

D. SmartNIC Offloading

Implementing a software Packet Dispatcher not only occupies resource (e.g. CPU core), but also adds extra latency [10]. In response, benefiting from the wide deployment of SmartNIC [10], we offload Packet Dispatcher to hardware to reduce resource overhead and avoid performance degradation.

However, SmartNIC only provides constrained programmability [24], which requires us to carefully implement the scheduling algorithm especially on overload detection. In response, we design a *timeslot-based scheduling* mechanism in Packet Dispatcher. We slice time into timeslots. In each timeslot, we update the load information of tenants and workers. At the beginning of each timeslot, we detect whether overloaded workers exist in last timeslot. If so, we migrate some tenants of these workers to other underloaded workers. A natural question is how to choose the timeslot value. A large value may not be effective due to traffic dynamics [3], while a small value may degrade NF performance due to frequent cache thrashing. In our current implementation, we use 1 second timeslot. Tenant traffic is predictable in at least 1 second [25] and we can use the last 1 second traffic patterns

CUR_SLOT	PKT_NB	WORKER_ID	MAX_NB

(a) *Tenant Table*

CUR_SLOT	PKT_NB	LAST_NB	TIMES	MAX_NB

(b) *Worker Table*

Fig. 4: Data structures of tables in Load Monitor

for effective scheduling. Besides, scheduling only once per second could eliminate the impact of cache thrashing.

A barrier to implement the scheduling mechanism is that SmartNIC does not provide *timer* functionality. To address this problem, we leverage the *timestamp* attached to every arriving packet by SmartNIC. Before dispatching a packet to workers, we extract its timestamp and shift right fixed bits (25 in our implementation) as the timeslot value. If it changes, we start the overload detection and tenants migration process.

Algorithm 1: Scheduling algorithm

```
input  : $(Pkt, T, W)$ - Packet, Tenant table, Worker table
output : $(Action, ID_{worker})$ - Whether to drop, Assigned worker
1  $(timeslot, tid) \leftarrow (get\_timeslot(Pkt), get\_tenant\_id(Pkt))$
2  $wid = T_{WORKER\_ID}(tid)$
3  if $timeslot == T_{CUR\_SLOT}[tid]$ then
4      $T_{PKT\_NB}[tid] + +$
5      if $T_{PKT\_NB}[tid] > T_{MAX\_NB}[tid]$ then
6          return $Action = Drop$
7  else
8      $TMP = T_{PKT\_NB}[tid]$
9      $(T_{CUR\_SLOT}[tid], T_{PKT\_NB}[tid]) = (timeslot, 1)$
10     if $timeslot! = W_{CUR\_SLOT}[wid]$ then
11         $W_{LAST\_NB}[wid] = W_{PKT\_NB}[wid]$
12         $(W_{CUR\_SLOT}[wid], W_{PKT\_NB}[wid]) = (timeslot, 0)$
13     //Overload detection
14     if $W_{LAST\_NB}[wid] < W_{MAX\_NB}[wid]$ then
15         $W_{TIMES}[wid] = 0$
16     else
17         // Overload detected, tenant migration starts
18         $ID_{migrate} \leftarrow get\_worker\_id(min(W_{LAST\_NB}))$
19         if $W_{LAST\_NB}[ID_{migrate}] + Tmp < W_{MAX\_NB}[ID_{migrate}]$ then
20             $T_{WORKER\_ID}[tid] = ID_{migrate}$
21             $W_{LAST\_NB}[wid] - = TMP$
22             $W_{LAST\_NB}[ID_{migrate}] + = TMP$
23         $W_{TIMES}[wid] + +$
24         if $W_{TIMES}[wid] > Threshold$ then
25             Report overload to the Agent
26 $W_{PKT\_NB}[wid] + +$
27 return $(Action, ID_{worker}) = (Pass, wid)$
```

Fig. 4 shows two tables maintained in Load Monitor. The CUR_SLOT and PKT_NB fields indicate current timeslot and the number of packet in this timeslot, the MAX_NB field refers to the maximum packet number allowed in each timeslot and is configurable. In the *Tenant Table*, the $WORKER\_ID$ field indicates the assigned worker for this tenant and might be dynamically updated. In the *Worker Table*, the $LAST\_NB$ field records the number of packets in last timeslot and the $TIMES$ field is an indicator of server load, which is accumulated if sustained overloading is detected.

Algorithm 1 shows the timeslot-based scheduling algorithm. For each packet, it first extracts the timeslot, tenant id and assigned worker id (line 1-2). If the packet is in the same timeslot, only PKT_NB field in both tables is increased (line 3-6, 26-27). Otherwise, it updates the CUR_SLOT and PKT_NB fields to indicate a new timeslot and overwrites the $LAST\_NB$ field for future scheduling (line 8-11). Next, it checks whether overloading occurs to the assigned worker in last timeslot (line 14-15). If so, the migration process starts. Firstly, it finds the least loaded worker in last timeslot based on the $LAST\_NB$ field and migrates to this worker only if not overloading the worker after migration (line 18-20). Then, the $LAST\_NB$ field is updated to avoid that subsequent tenants are migrated to the same worker in this timeslot (line 21-22). Finally, if the sustained overloading times of a worker ($TIMES$ field) exceeds a pre-configured value, it sends a message to the *Agent* as a overloading report (line 23-25).

E. Optimizations

To further improve the performance of Buffet, we also propose two optimization technologies.

Packet Tagging. After workers receive packets, they need to calculate the tenant id of these packets. However, tenant id of a packet has been calculated in SmartNIC. Therefore, after the Traffic Classifier get the tenant id, it also attaches this id to the packet as a tag. Thus, the worker can quickly get tenant id by extracting the tag filed from packet. Inspired by [26], we use the 6-bit DS field (part of the 8-bit ToS) as the tag field, which is sufficient to represent enough tenants in a server.

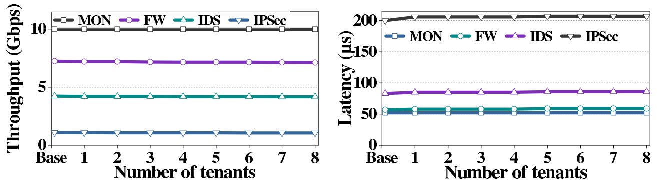
Packet Buffer. After identifying the tenant id of a packet, workers need to look up the Memory Table to get the memory region that belongs to this packet. However, this could introduce significant performance overhead if looking up table for every packet. In Buffet, as shown in Fig 3, we provide a *Packet Buffer* for each tenant and buffer their packets for batch processing to amortize the additional overhead.

IV. Implementation and Evaluation

We have implemented a prototype of Buffet on a testbed composed of two servers. One server carries the Buffet infrastructure and is equipped with two Intel Xeon E5-2620 v3 CPUs (2.4GHz, 12 physical cores), 128GB total memory (DDR4, 2133MHz) and two Netronome Agilio CX Dual-Port 10 SmartNICs [24]. The other server runs a packet generator, which sends and receives packets that follow the flow size distribution derived from [12]. The server for the generator has the same configurations as the previous one but with two dual-port 10G Intel NICs (X520-DA2). Both servers run Ubuntu 16.04 (with kernel 4.4.0) and use DPDK 18.02 for networking I/O. We have implemented four NFs for evaluation including a Monitor (maintaining flow statistics), a Firewall (performing hash table match), an IDS (performing signature match) and an IPSec (based on AES-256 algorithm).

A. Multitenancy Performance Overhead

To demonstrate that Buffet can support general tenant isolation with marginal performance overhead, we compare it with single-tenant NFs. We implement four single-tenant NFs in the same way of Buffet and run both systems on one CPU core. We take the performance of single-tenant NFs as the baseline, and vary the number of tenants in Buffet from 1 to 8. As shown in Fig. 5, compared with the performance of single-tenant NFs, Buffet introduces little overhead in both throughput (within 4%) and latency (within 4%), and is not impacted by the number of tenants.



(a) Throughput (b) Average Latency

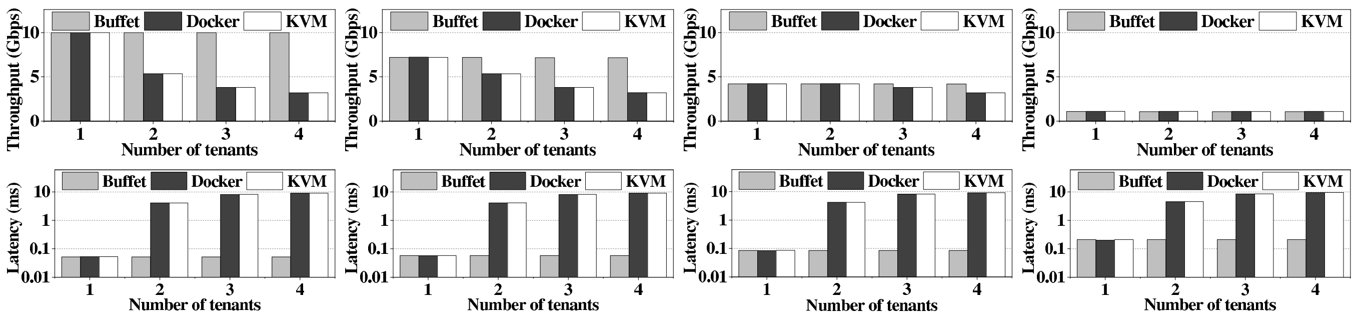
Fig. 5: Performance with different # of tenants

B. Performance Improvement

Buffet imposes lightweight intra-process isolation to co-locate multiple tenants. To demonstrate the performance improvement of Buffet over conventional isolation technologies, we compare it with the VM and container based isolation mechanisms. We choose two open source platforms in our evaluation (Docker [27] for container and KVM [28] for VM) and run the single-tenant NFs in multiple Docker or KVM instances for different tenants. Especially, we run Docker in *privileged* mode to directly access host NIC ports and enable NIC *SR-IOV* for KVM to avoid I/O performance bottleneck. Besides, we consolidate all dockers or KVMs in one CPU core to ensure equal resource usage with Buffet and compare the total throughput as well as average latency of all tenants. As shown in Fig. 6, for all four NFs, with the increase of tenant number, the performance of Buffet does not degrade, while the performance of Docker and KVM drops significantly (up to 3x in throughput and 172x in latency).

C. Efficiency of Packet Dispatching Algorithm

Buffet improves resource efficiency by over-subscribing workers with tenants and actively schedules traffic to prevent potential overload. To demonstrate this, we compare our scheduling algorithm with fixed tenant-worker mapping mechanism and optimal scheduling algorithm in simulation. We choose IDS as our simulated NF and allocate 3 workers to provide 12.6 Gbps throughput in total. We vary the number of tenants from 8 to 20 and each tenant randomly generates traffic from 0 to 4.2 Gbps but the total is 10 Gbps. We implement these three algorithms and generate one million test cases for evaluation. As shown in Fig. 7, compared with the fixed tenant-worker mapping, the optimal and our algorithm could reduce overload percentage significantly. However, as shown in Fig. 8, with the increasing number of tenants, the calculation time of optimal algorithm raises quickly and becomes unacceptable while our algorithm always finishes in a limited time. Besides, the average migration percentage of tenants with our algorithm is far less than the optimal algorithm which could further improve performance (The results of 16 and 20 in optimal algorithm are not shown due to long calculation time).



Monitor Firewall IDS IPSec

Fig. 6: Performance with different isolation solutions

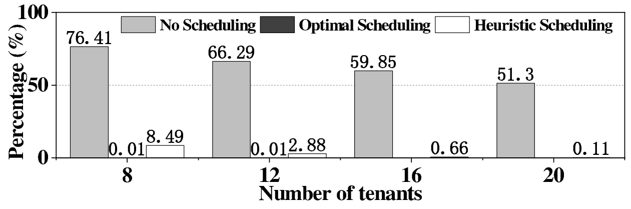


Fig. 7: Effect of scheduling

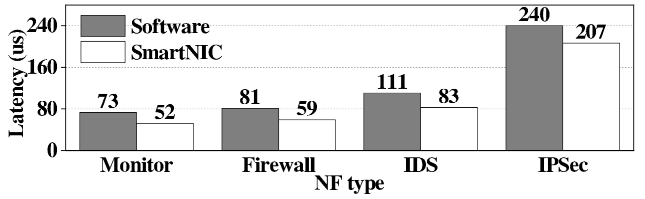
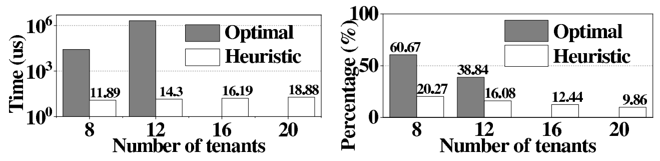


Fig. 9: Latency reduction of different NFs



(a) Time complexity (b) Migration percentage

Fig. 8: Scheduling overhead

D. Latency Reduction with SmartNIC

SmartNIC offloading could reduce overall latency by avoiding additional packet processing on CPU. To demonstrate it, we implement a software based Packet Dispatcher on CPU and compare it to our SmartNIC based solution among four NFs. As shown in Fig. 9, for these NFs, SmartNIC could reduce the average latency from 14% to 29%.

V. Conclusions

We have presented Buffet, a general framework to enable multitenancy for a wide range of NFs. Buffet proposes a general programming abstraction to achieve automatic tenant isolation and designs an in-SmartNIC Packet Dispatcher to enforce dynamic tenant-level affinity. Evaluation results have demonstrated that Buffet can achieve high performance and resource efficiency for various multi-tenant NFs.

Acknowledgment

This work is supported by the National Key Research and Development Program of China (2016YFB0801301, 2016QY12Z2103), and the Internet Architecture Key Lab Program (BNR2019ZS01003). Jilong Wang (wjl@cernet.edu.cn) is the corresponding author.

References

[1] V. Sekar *et al.*, “Design and implementation of a consolidated middlebox architecture,” in *NSDI*, 2012.
[2] J. Sherry *et al.*, “Making middleboxes someone else’s problem: network processing as a cloud service,” *ACM SIGCOMM CCR*, 2012.
[3] D. Xie *et al.*, “The only constant is change: incorporating time-varying network reservations in data centers,” *ACM SIGCOMM CCR*, 2012.
[4] G. N. Purdy, *Linux iptables Pocket Reference: Firewalls, NAT & Accounting*. ” O’Reilly Media, Inc.”, 2004.
[5] P. Barham *et al.*, “Xen and the art of virtualization,” in *SIGOPS*, 2003.
[6] S. Soltesz *et al.*, “Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors,” in *SIGOPS*, 2007.
[7] K. Tan *et al.*, “Protego: cloud-scale multitenant ipsec gateway,” 2017.
[8] A. Panda *et al.*, “Netbricks: Taking the v out of nfv.” in *OSDI*, 2016.
[9] D. E. Eisenbud *et al.*, “Maglev: A fast and reliable software network load balancer.” in *NSDI*, 2016.
[10] D. Firestone *et al.*, “Azure accelerated networking: Smartnics in the public cloud,” in *NSDI 18*, 2018.
[11] A. Roy *et al.*, “Inside the social network’s (datacenter) network,” in *ACM SIGCOMM CCR*, 2015.
[12] T. Benson *et al.*, “Network traffic characteristics of data centers in the wild,” in *SIGCOMM conference on Internet measurement*, 2010.
[13] S. A. Baset, L. Wang, and C. Tang, “Towards an understanding of oversubscription in cloud.” in *Hot-ICE*, 2012.
[14] S. Rajagopalan *et al.*, “Split/merge: System support for elastic execution in virtual middleboxes.” in *NSDI*, 2013.
[15] A. Gember-Jacobson *et al.*, “Opennf: Enabling innovation in network function control,” in *ACM SIGCOMM CCR*, 2014.
[16] J. Sherry *et al.*, “Rollback-recovery for middleboxes,” in *ACM SIGCOMM CCR*, 2015.
[17] M. Kablan *et al.*, “Stateless network functions: Breaking the tight coupling of state and processing.” in *NSDI*, 2017.
[18] J. Khalid *et al.*, “Paving the way for nfv: Simplifying middlebox modifications using statealyzr.” in *NSDI*, 2016.
[19] Microsoft. (2018) Receive side scaling (rss). [Online]. Available: https://technet.microsoft.com/en-us/library/hh997036.aspx
[20] C.-P. Bezemer and A. Zaidman, “Multi-tenant saas applications: maintenance dream or nightmare?” in *IWPSE*, 2010.
[21] S. Woo *et al.*, “Elastic scaling of stateful network functions,” in *NSDI 18*, 2018.
[22] N. McKeown *et al.*, “Openflow: enabling innovation in campus networks,” *ACM SIGCOMM CCR*, 2008.
[23] M. Cieliebak *et al.*, “On the complexity of variations of equal sum subsets.” *Nord. J. Comput.*, 2008.
[24] Netronome. Netronome agilio cx dual-port 10 gigabit ethernet smartnic.
[25] T. Benson *et al.*, “Microte: Fine grained traffic engineering for data centers,” in *CONEXT*, 2011.
[26] S. K. Fayazbakhsh *et al.*, “Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags.” in *NSDI*, 2014.
[27] Docker. (2018). [Online]. Available: https://www.docker.com/
[28] A. Kivity *et al.*, “kvm: the linux virtual machine monitor,” in *Proceedings of the Linux symposium*, 2007.