Grus: Enabling Latency SLOs for GPU-Accelerated NFV Systems

Zhilong Zheng[†], Jun Bi[†], Haiping Wang[†], Chen Sun[†], Heng Yu[†], Hongxin Hu[§], Kai Gao[†], Jianping Wu[†]
[†]Institute for Network Sciences and Cyberspace, Tsinghua University
[†]Department of Computer Science, Tsinghua University
[†]Beijing National Research Center for Information Science and Technology (BNRist)
[§]School of Computing, Clemson University

***Abstract*—Graphics Processing Unit (GPU) has been recently exploited as a hardware accelerator to improve the performance of Network Function Virtualization (NFV). However, GPU-accelerated NFV systems suffer from significant latency variation when multiple network functions (NFs) are co-located in the same machine, which prevents operators from supporting latency Service Level Objectives (SLOs). Existing research efforts to address this problem can only guarantee a limited number of SLOs with very low resource utilization efficiency. In this paper, we present the Grus framework to support latency SLOs in GPU-accelerated NFV systems. Grus thoroughly analyzes the sources of latency variation and proposes three design principles: (1) dynamic batch size setting is needed to bound packet batching latency in *CPU*; (2) a reordering mechanism for data transfer over *PCI-E* is required to guarantee the stalling time; and (3) maximizing concurrency in *GPU* is necessary to avoid NF execution waiting time. Guided by the principles, Grus consists of two logical layers including an infrastructure layer and a scheduling layer. The infrastructure layer is equipped with an in-CPU *Reorder-able Worker Pool* that could adjust batch size and packet transfer order, and in-GPU *Controllable Concurrent Executors* to provide maximized concurrency. The scheduling layer runs a heuristic algorithm to perform accurate and fast scheduling to guarantee SLOs based on our prediction models. We have implemented a prototype of Grus. Extensive evaluations demonstrate that Grus can significantly reduce latency variation and satisfy 4.5 × more SLO terms than state-of-the-art solutions.**

I. INTRODUCTION

Network Function Virtualization (NFV) was recently introduced to address the limitations of traditional dedicated middleboxes. NFV implements network functions (NFs) on commodity hardware to improve service delivery flexibility and reduce overall costs. However, due to the adoption of virtualization techniques, software-based NFs suffer from low performance with respect to both latency and throughput [1], [2]. In response, recent research efforts have proposed to introduce Graphics Processing Units (GPUs) with massive computing cores [3] as hardware accelerators to enhance the performance of a wide range of NFs, such as routing [4], NIDS [5], and IPSec [6], [7]. The high performance brought by GPU enables operators to provide performance Service Level Objectives (SLOs) [8] of latency and throughput when processing multiple flows in NFV [8], [9].

However, when consolidating multiple NFs in one host, current GPU-accelerated NFV systems suffer from *significant latency variation* for each NF, making it challenging to effectively guarantee *latency SLOs* [8]. According to our evaluation in §II-A, the latency of a firewall increases as much as 2.1× after starting nine other NFs in the same GPU. Such *high latency variation* prevents operators from guaranteeing latency SLOs for latency sensitive applications, such as web search, online retail, and algorithmic stock trading [2].

Some research efforts have been devoted to guaranteeing latency in GPU-accelerated NFV systems. A typical packet processing pipeline of GPU-accelerated NFV systems is as follows. CPU first collects packets from network interface cards (NICs) and then delivers them to GPU through Peripheral Component Interconnect Express (PCI-E). GPU executes NF logic on packets, after which CPU copies packets back and sends them out. Existing researches focus solely on guaranteeing latency of *one stage of the above pipeline*. ResQ [8] provisioned latency guarantee for NFs in *CPU*. Silo [10] ensured latency during *packet transmission* between network devices. Baymax [11] provided QoS in latency inside *GPU*. However, above solutions fail to support effective and efficient SLO guarantee in GPU-accelerated NFV context in two aspects. First, solutions that guarantee latency in GPU [11], [12] could only support a limited number of concurrent tasks, resulting in very a low GPU utilization efficiency [12]. We will illustrate in §II-A that current solutions can only support a small set of SLOs with near up to 75% of GPU resources left vacant. Second, there is no coordinated scheduling solutions to jointly enforce latency guarantee in CPU, PCI-E, and GPU, which makes it challenging to guarantee end-to-end latency of the entire pipeline in GPU-accelerated NFV systems.

To address the above problems, we construct a *latency model* by performing a thorough analysis of each step in the packet processing pipeline to understand which steps may introduce latency and why latency variation occurs. We observe that the sources of latency variation are mainly threefold.

- *CPU collects packets as batches*. Currently the batch size is statically configured regardless of incoming packet rate. However, as packet rate drops, it takes longer time to collect a full batch of packets, causing latency variation.
- *CPU transmits prepared packet batches to GPU over PCI-E*. Due to the serial nature of PCI-E [13], concurrent NFs have to contend to monopolize PCI-E. Therefore, the unpredictable stalling time due to contention adds to variation.
- *The current task scheduler in GPU provides limited con-*

currency. Multiple NFs have to wait for free task executors. Such unpredictable waiting time also incurs variation.

According to latency variation sources, we propose three design principles to guarantee latency. First, *dynamic batch size setting* is necessary to bound the time of batching incoming packets to adapt to dynamically changing traffic. Second, a *reordering mechanism for data transfer* is required to achieve predictable stalling time on PCI-E. Third, *maximizing concurrency and minimizing interference for task execution* are essential to avoid waiting time in GPU.

Guided by the above design principles, in this paper, we present Grus, a framework to enforce latency SLOs in GPU-accelerated NFV systems. Grus consists of two logical layers including an infrastructure layer and a scheduling layer. We introduce three core components in the infrastructure layer. First, we design an in-CPU *Reorder-able Worker Pool* that could enable workers to adjust batch size and transfer data in a specific order according to scheduling policy to bound latency for packet batching in CPU and data transfer over PCI-E. Second, the default hardware scheduler in GPU is a black box that cannot be customized to provide maximum execution concurrency. In response, we design an in-GPU *Controllable Concurrent Executor* that circumvents the default GPU scheduler and provides maximum concurrent execution units for NFs. Third, we propose an in-GPU *NF Assignment Table* that enables launching NF kernels on a specific set of executors to make NF waiting time short and predictable. In the scheduling layer, Grus introduces a *Latency SLO-aware Scheduler* that jointly manages all resources and makes scheduling decisions to meet latency SLOs based on our *Prediction Models*. To make scheduling fast, we propose a heuristic algorithm to quickly find a feasible scheduling solution for all SLOs. In summary, Grus makes the following contributions:

- We identify the latency variation in GPU-accelerated NFV systems (§II), create a latency model by thoroughly analyzing each step of the processing pipeline, and present three design principles to guarantee latency (§III).
- We propose Grus, a GPU-accelerated NFV system, which enables latency SLOs for multiple co-located NFs. Grus introduces a new infrastructure design (§IV) and scheduler design (§V) to effectively enforce latency SLOs.
- We implement a prototype of the Grus system and perform extensive experiments. Evaluation results demonstrate that Grus can effectively guarantee latency and satisfy 4.5 × more SLO terms than state-of-the-art solutions. (§VII).

We discuss the design limitations of Grus in §VI. Specifically, Grus does not handle PCI-E contention during data transfer from GPU to CPU after packet processing in GPU. Moreover, Grus focuses on enabling latency SLOs for a single NF, which is the first step towards guaranteeing the latency of an entire chain.

II. Motivation and Challenges

A. Background and Motivation

Background. A modern programmable GPU acts as a co-processor that receives the code (called "kernels") and data from the host CPU. GPUs have on-board device memory, so data must be *copied* in from the server's main memory ("host memory") over the PCI-E bus after the workers in CPU prepare enough data (e.g., a batch of packets). Then GPU will process the data and inform CPU to copy processed data back to host memory.

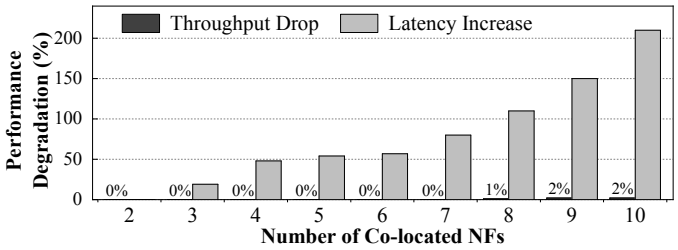


Fig. 1: Average latency increase of a firewall when starting other NFs in the GPU-accelerated NFV system

Adopting GPU as an accelerator has been shown to enable high-performance NFs by many existing research works [3]–[5], [7], [12], [14]–[16]. For example, PacketShader [4] presented a high-performance router by offloading workloads to GPUs. MIDeA [14] and Kargus [5] introduced a high-performance NIDS based on GPU. NBA [3] and GASPP [7] presented a general GPU-based framework to boost the performance of a wide range of NFs. G-NET [12] is the state-of-the-art work that aims to improve the utilization while achieving predictable latency for one solely running NF in GPU. However, according to our experiments, above efforts cannot guarantee end-to-end latency in GPU-accelerated NFV networks. Next, we will introduce the significant latency variation without effective guarantee methods. Then we reveal the insufficiency of existing techniques to provide latency guarantees in GPU-accelerated NFV.

Latency variation in GPU-accelerated NFV systems. To illustrate the performance variation of co-located NFs in GPU-accelerated NFV systems, we build a system based on the architecture and design from NBA [3]. We use a testbed with a server equipped with an NVIDIA Titan Xp GPU and a server as the traffic generator that generates a modest traffic rate of 2 Gbps of each flow, with the packet size distribution derived from [17] (more details are stated in §VII). We use the *firewall* NF in this experiment and measure the latency variation of the initial firewall instance when starting zero to nine new firewall instances (each processes different flows) in the same server.

Fig. 1 shows the latency variation of the initial firewall instance. We observe that as more NFs run concurrently, the average latency of the initial firewall increases significantly. Two more co-located NFs increases the latency of the initial firewall by 0.19×. When we start nine other NFs, the initial firewall suffers from a 2.1× latency increase. We also notice that there is no obvious throughput drop when starting one to nine co-located NFs. The above observations motivate us to guarantee latency for NFs in GPU-accelerated NFV systems..

Insufficiency of existing solutions. Many prior efforts have proposed solutions to provide guaranteed performance on one of the shared resources in GPU-accelerated NFV pipeline [8], [10], [11], [18]–[20]. A strawman approach is to naively com-

bine solutions for CPU, data transfer, and GPU to guarantee latency of the entire pipeline. However, as mentioned above, we identify two major shortcomings of this approach.

First, current solutions that guarantee latency in GPU [11], [12] provisions very limited concurrency. The default GPU task scheduler provides a limited number of concurrent executors named *Streaming Multiprocessors (SMs)* that could execute tasks in parallel [11]. Threads in GPU are equally divided to each SM in the granularity of *Warp*. Warp is the basic scheduling unit in a GPU, which contains 32 threads. Each SM may contain 64 or more warps according to the GPU model. For mainstream GPU applications such as big-data analytics [21] and machine learning [22], one application could fully occupy the resource of one or multiple SMs, which could achieve a high utilization efficiency of GPU. However, NFs in NFV are typical *narrow tasks* that usually occupies less than 16 warps in an SM [23]. Allocating an entire SM to an NF significantly compromises thread utilization efficiency. If we assign multiple NFs to an SM, NFs have to wait in line for execution, adding to unpredictable waiting time that may violate SLOs. Therefore, current solutions can only satisfy a limited number of SLOs. Suppose an NF consumes 16 warps in an SM equipped with 64 warps. We can only support a limited set of latency SLOs with 75% GPU resource vacant.

Second, existing solutions that enable guaranteed latency focus on either CPU [8], packet transmission [10], or GPU [11], [20], [24] individually, but not together. There is no coordinated scheduling solutions to jointly enforce latency guarantee in CPU, PCI-E, and GPU. However, in GPU-accelerated NFV, latency SLOs regulate the end-to-end latency [10] of packets across the entire processing pipeline. Without coordinated scheduling, we cannot decide how much latency budget can be allocated to each resource type, making it unavailing to guarantee latency in each resource respectively.

`Grus`. Based on above motivations, we propose a novel framework, `Grus`, to guarantee latency in GPU-accelerated NFV networks. With coordinated scheduling, `Grus` targets at reducing NF latency variation and supporting the maximum number of latency SLOs in GPU-accelerated NFV systems.

B. Design Challenges

We encounter three major challenges in the design of `Grus`.

Identifying the sources of latency variation. A GPU-accelerated NFV system is a heterogeneous platform with multiple types of resources, which introduces many potential causes of latency variation. Thus it is challenging to identify variation sources and build the latency model for scheduling. In response, `Grus` thoroughly analyzes the packet processing pipeline and presents our latency model (§III).

Infrastructure design to support latency SLOs. Enforcing latency SLOs in GPU-based NFV incurs several concerns on the infrastructure design. First, current infrastructure in CPU does not allow changing batch sizes or assigning transmission orders of packets over PCI-E. To address this challenge, we design a Reorder-able Worker Pool in CPU to support dynamic batching and transfer ordering. Second, the default GPU scheduler cannot be customized to provide adequate concurrency, making it challenging to effectively guarantee latency SLOs in GPU. In response, `Grus` circumvents the default GPU scheduler and designs Controllable Concurrent Executors to execute NFs with maximal concurrency (§IV).

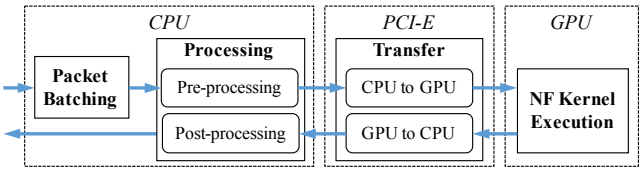


Fig. 2: A typical GPU-accelerated NF processing pipeline

Scheduler design to ensure accurate and fast scheduling. Finally, we are challenged to design a coordinated, accurate, and efficient scheduler across heterogeneous resources to guarantee latency SLOs of multiple NFs. Due to resource constraints, not all SLOs can be satisfied (or *admitted* [10]) and we are challenged to find the maximal set of SLOs that can be supported by the system. In response, we propose the `Grus` scheduler that targets at maximizing the admitted SLOs in an accurate and fast fashion (§V).

III. Latency Analysis and Design Overview

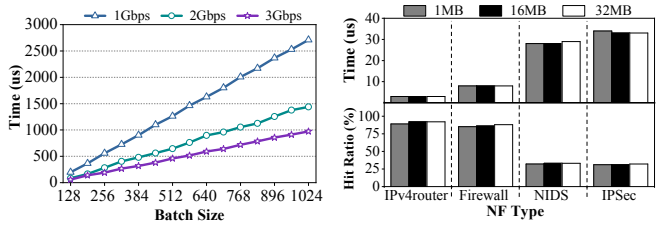
In this section, we first identify the sources of latency variation in a GPU-accelerated NFV systems. Based on our analysis, we propose three design principles to guarantee latency. Finally, we present the design overview of `Grus`.

A. Understanding Latency Variation of GPU-accelerated NFs

Fig. 2 shows a typical GPU-accelerated NF processing pipeline, where a packet travels through multiple types of resources (e.g., CPUs, PCI-E and GPUs). We identify four major components that introduces latency in the pipeline: (1) a worker thread in CPU first fetches packets from NICs and batches them together; (2) according to NF specification, the worker thread would pre-process packets before sending packets into GPU, and post-process them after retrieving from GPU; (3) the worker thread transfers the packet batch from host memory to GPU memory via PCI-E; and (4) the worker thread launches NF kernels to process packets in GPU.

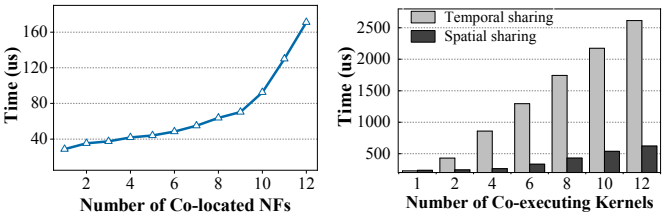
When multiple NFs are competing for resources in a consolidated system, each component may suffer latency variation and cause SLO violations. Next we discuss how they can vary when co-locating with other NFs and how to bound them to achieve guaranteed latency.

***Packet batching:* variation due to traffic dynamics.** Traffic of NFs could be dynamically changed due to diverse SLO specifications for throughput. Intuitively we know that different traffic rates could vary the time of packet batching with a fixed batch size (i.e., the number of packets in a batch). To study the impact of traffic dynamics over packet batching latency, we measure how the batching time changes when we vary the traffic rate. As shown in Fig. 3(a), we observe that the batching time is varied a lot for a fixed size with different traffic rates. Moreover, the traffic of an NF is also dynamically changed at runtime when dynamic-SLOs are required [8]. Hence, if



(a) Average batching time with different batch sizes and packet rates

(b) Average CPU processing time and LLC hit ratio with different LLC sizes



(c) Average data transfer time with different number of co-located NFs

(d) Average kernel execution time with different number of co-located NFs

Fig. 3: Latency of an NF for packet batching, CPU processing, PCI-E transferring, and GPU execution

the batch size is not adjusted according to the traffic rate, the batching time can vary significantly. A naive approach is to adopt a small batch size so that the batching time tops at an endurable level when traffic rate drops. However, it would sacrifice throughput when packet rate increases [3], [16], [25].

Pre-/post-processing: **near zero variation.** Normally, applications running in a CPU may contend computing resources, such as CPU cycles and caches. However, existing NFV solutions [1]–[3], [26] demonstrate that a today's NF usually runs on a dedicated CPU core, which eliminates contention over CPU cycles among NFs. Therefore, as mentioned in recent work [8], [27], we consider the contention over last-level cache (LLC) as the potential cause of time variation. To study its impact in GPU-accelerated NFV context, we isolate different sizes of LLC to different NFs and measure the processing time. We use Intel Cache Allocation Technology (CAT) [28] to isolate LLC to a dedicated core (i.e., an NF). As the upper half of Fig. 3(b) shows, we discover that there is almost *no variation* of the pre-/post-processing time in all four NFs we measured when allocating different amounts of LLC. For example, this time of an NIDS is 31μs, 31 μs and 32 μs with 32 MB, 16 MB, and 1 MB LLC respectively. To understand the results, we use Intel Performance Counter Monitor (PCM) [29] to monitor the cache hit ratio of LLC. The lower half of Fig. 3(b) shows that for an NF with different LLC allocations, the cache hit ratio almost never changes. This is because most compute-intensive instructions are relieved from CPU to GPU. Thus, we conclude that the time of pre-/post-processing suffers from *near zero* variation.

PCI-E transfer: **variation due to contention.** Prior works [11], [12] have revealed that the time of transferring on PCI-E is only relevant to the size of transferred data. However, transferring over PCI-E is serial [13]. When multiple packet batches belonging to different NFs contend for PCI-E, the transfer time of a packet batch can be varied out of control due to uncertain queuing and waiting. Fig. 3(c) shows that the transfer time of a packet batch (batch size is 256) of a firewall increases significantly when more NFs contending to PCI-E.

Kernel execution: **variation due to task waiting.** For an NF kernel that executes on GPU solely, its execution time is relevant to kernel's complexity and data size (i.e., batch size) [11], [12]. State-of-the-art GPUs [12], [30] support sharing a GPU via two ways, *temporal sharing* and *spatial sharing*. However, even if the above two sharing approaches are enforced, the execution time still increases as shown in Fig. 3(d). We observe that temporal sharing increases the execution time significantly. This is because temporal sharing delays executions until another NF yields the whole GPU. Hence, waiting for execution significantly increases the overall latency. Meanwhile, spatial sharing supports concurrent executions and could potentially eliminate waiting. However, it still introduces significant latency variation once the number of NFs increases, which happens due to the limited concurrency (we will provide more details in §IV). It exposes the need for a highly concurrent spatial sharing mechanism to GPU resource to serve more kernels at the same time.

B. Latency Model and Design Principles

Based on above analysis, in a GPU-accelerated NFV system with multiple co-located NFs, the end-to-end latency of an NF includes: the time of packet batching $T_{batching}$, pre-/post-processing T_{pre_post}, PCI-E transfer waiting $T_{w\_transfer}$, data transfer over PCI-E $T_{transfer}$, NF kernel execution waiting $T_{w\_execution}$, and NF kernel execution in GPU $T_{execution}$. We model the latency T_{nf} as follows.

$$\begin{aligned} T_{nf} &= T_{batching} + T_{pre\_post} \\ &+ (T_{w\_transfer} + T_{transfer}) \\ &+ (T_{w\_execution} + T_{execution}) \end{aligned} \quad (1)$$

Among them, $T_{transfer}$, T_{pre_post} and $T_{execution}$ are predicable, while others may introduce unexpected variation. Thus, we present three design principles to enable latency SLOs. First, an *adaptive batch size setting* is necessary to bound $T_{batching}$ with the requirement of dynamically changeable traffics. Second, a *reordering mechanism for data transfer* is required to achieve predictable stalling for $T_{w\_transfer}$ on PCI-E. Third, *maximizing concurrency and minimizing interference for task execution* are essential to eliminate waiting time on GPU, i.e., $T_{w\_execution}$.

C. Grus Design Overview

Guided by the above design principles, we introduce Grus with an infrastructure layer and a scheduling layer to provide guaranteed latency for co-located NFs in GPU-accelerated NFV. Fig. 4 presents the Grus system overview. In the infrastructure layer, we first introduce a *Reorder-able Woker Pool* that performs concurrently requested data transfer over PCI-E in a configurable order. Additionally, we design *Controllable Concurrent Executors* that could maximize NF execution concurrency. Finally, we introduce an *NF Assignment Table*

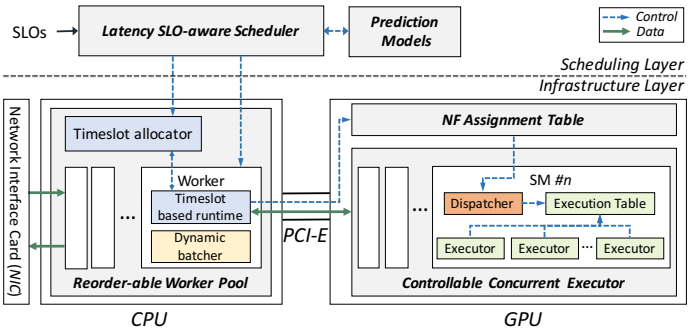


Fig. 4: Grus design overview

that maintains information about pending NFs as well as completed NF tasks. The scheduling layer is equipped with two logical components including *Latency Prediction Models* and a *Latency SLO-aware Scheduler*. We maintain a set of latency prediction models to predict the processing time of packet batching, data transfer, and NF kernel execution. The scheduler takes the prediction models and requested SLOs as input to accurately and quickly produce an optimal batch size and order for each SLO term. We present detailed infrastructure design in §IV, and scheduling layer design in §V.

IV. Infrastructure Layer Design

In this section, we introduce detailed design of three components in Grus's infrastructure: *Reorder-able Worker Pool*, *Controllable Concurrent Executors*, and *NF Assignment Table*.

A. Data Transfer in Order: Reorder-able Worker Pool

Each NF has its own corresponding *worker*, which is run in dedicated CPU cores. When multiple workers request data transfer over PCI-E simultaneously, a specific order should be assigned to guarantee data transfer latency. A strawman mechanism is to design a *centralized data transfer engine* that handles data transfer for all workers [11]. Workers delegate data transfer tasks to the engine, which assigns orders to each task and performs transfer accordingly. However, this solution has two major shortcomings. First, now that the centralized engine has to receive all transfer requests and enforce the transfer tasks, it may be heavily burdened and become the performance bottleneck. Second, a worker has to ensure that the data transfer is finished, after which it can launch the NF kernel in GPU. Thus, the worker must stall and perform synchronization with this engine to check transferring status, which fully occupies the CPU in the worker, delays packet batching, and seriously compromises performance.

To provide an effective reordering mechanism with low overhead, we design a *Reorder-able Worker Pool* in CPU. The key idea is to *decouple* order assignment and data transfer by enabling workers to obtain orders from the centralized engine and transfer data via themselves. However, after an order is assigned to a worker, it has to synchronize with other workers to wait until workers with frontier orders finish data transfer, which still introduces performance overhead. Inspired by [31], we enable allocating *timeslot* for each worker as an indicator of the order. A timeslot directly regulates when the

```
Runtime(worker_id)
1  request_buffer ← requests_buffers[worker_id]
2  timeslot_buffer ← timeslot_buffers[worker_id]
3  while true do
4      if batch_prepared() == true then
5          Send a request via request_buffer
6      timeslot ← Get from timeslot_buffer
7      if timeslot == NULL then
8          continue
9      while true do
10         cur_time ← Get system clock
11         if cur_time ≥ timeslot then
12             Start data transfer
13             break
```

Fig. 5: Pseudo code of *Timeslot based runtime*

worker submits data transfer task on PCI-E. Note that as PCI-E handles tasks in serial, a worker may need to wait for its turn to transfer the data after submitting the task. However, this mechanism ensures that data transfers of multiple workers could happen in a configurable order. To achieve this goal, as shown in Figure 4, we design two components including (1) a *Timeslot allocator* that receives transfer requests from workers and calculates timeslots for workers according to their orders, and (2) a *Timeslot based runtime* in each worker that receives timeslot from the allocator and enforces data transfer accordingly. Next we introduce the two components in detail.

Timeslot allocator. The *Timeslot allocator* maintains the orders of all workers, which can be dynamically configured by the global scheduler, and use the orders to calculate the right timeslots for the data transfer requests from workers. First, the allocator selects a pending request according to the configured order of each worker. Then it calculates a timeslot for the selected request using its order o and system time t: $timeslot = t + \alpha * o$. Here, we adopt a small α to slightly differ the task submission time of workers from each other. Finally, it sends the timeslot information back to the worker.

Note that using system time t is accurate enough for reordering mechanism since the local clock system of all workers and the allocator are the same clock system of the host. The Timeslot allocator is lightweight enough as it performs simple calculation without the burden of actual data transfer, which minimizes its performance overhead.

Timeslot based runtime. We design a *Timeslot based runtime* in each worker to interact with the *Timeslot allocator* and enforce data transfer according to the allocated timeslot. A naive scheme for the interaction between the runtime and the allocator is *request-wait-response*, *i.e.* a synchronized approach where the worker's CPU has to busily wait for timeslot allocation, which introduces serious performance degradation. Instead, we introduce a *request buffer* and *timeslot buffer* for each worker to make all interactions *asynchronous*. Fig. 5 presents the pseudo code. When initializing the runtime, every worker gets its individual buffers (lines 1-2). Next the runtime enters a loop. It first checks whether a batch of packets has been prepared. If so, it sends a request to the allocator

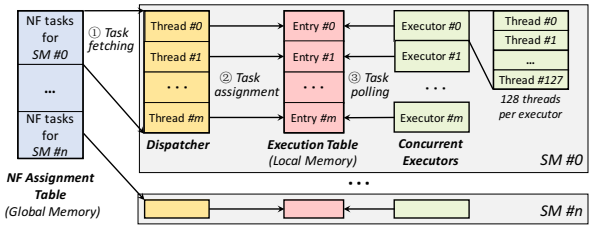


Fig. 6: Task scheduling in GPU based on *NF Assignment Table* and *Controllable Concurrent Executors* in SMs

(lines 4-5). Afterwards, it attempts to get a timeslot from *timeslot_buffer* (line 11). If no timeslot is available, it goes back checking packet batch (lines 7-8), which prevents waiting for the timeslot allocation. Otherwise, it enters a loop and keeps checking the clock, if current time reaches the acquired timeslot, it starts data transfer on PCI-E (lines 10-13).

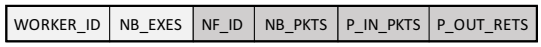
B. Maximal Concurrency: Controllable Concurrent Executors

The default task scheduler in GPU uses Streaming Multiprocessors (SMs) as basic concurrent execution units for parallel execution. The number of SMs is often limited. The latest NVIDIA TITAN Xp GPU contains merely 30 SMs. Moreover, an NF task may use a set of SMs to reduce processing latency meanwhile maintaining resource efficiency [3], [12], which implies that only a small fractions of NF kernel can execute concurrently. This lack of available concurrency for multiple co-executing NF kernels could result in task queuing and waiting, which leads to unpredictable processing latency. Furthermore, an SM is a coarse-grained resource block that contains thousands of threads. However, the default hardware scheduler in GPU might allocate an entire SM to a task even if the task only needs 10s of threads. This results in a very low thread utilization and may reduce the number of supported SLOs. Finally, the behaviour of the default scheduler cannot be customized, making it challenging to control the scheduling and resource allocation policy.

In response, we design *Controllable Concurrent Executors*, which enable controllable task scheduling with maximal concurrency by slicing an SM into fine-grained execution units named *executors*. Next we introduce our scheduling mechanism that circumvents the default scheduler, and how we slice coarse-grained SMs into executors. Note that threads in different SMs cannot be allocated to the same task. Therefore, scheduling related modules are inserted into *each SM* to control executors in *the same SM*.

Task scheduling mechanism. To circumvent the default GPU scheduler and own control over underlying executors, we adopt the *persistent threads* technique proposed by [16], [20], [32]. A persistent thread indicates it sinks into a loop and will never be torn down. As shown in Fig 4, we configure a small fraction of persistent threads in an SM named *Dispatcher* and enable them to fetch and assign tasks. This is possible because a kernel in GPU can not only be started by in-CPU workers, which will then be scheduled by the default GPU scheduler, but also by other threads in GPU. Thanks to the sustained lifecycle of persistent threads, we are able to use dispatchers as our scheduler to control underlying executors.

(a) *Execution Table*



(b) *NF Assignment Table*

Fig. 7: Data structures of the two tables in GPU

Fine-grained concurrent executors. Our intuition behind increasing concurrency is the observation that NFs are typical *narrow* tasks which usually occupies less than 16 warps in an SM [23]. Allocating an entire SM to them significantly compromises thread utilization. Therefore, inspired by [23], [33], we adopt a novel *intra-SM slicing mechanism* to achieve higher concurrency. As shown in Fig. 6, a *Concurrent Executor* includes 128 threads, which is fine-grained enough compared with an entire SM. State-of-the-art GPU contains at most 2048 threads in an SM, meaning that we could have 16 executors that could run in parallel. The reason why we do not use the most fine-grained execution units, *i.e. Warps* in [23], as the executor is to avoid performance degradation due to frequent dispatching. Suppose an NF needs 16 warps. If we use 128 threads as an execution unit, the Dispatcher only needs to dispatch 4 times, instead of 16. Dispatching is expensive since it has to guarantee *atomic write*. Moreover, as reported in [3]–[5], [12], an NF often adopts a large set of threads in an SM to achieve better system efficiency. Therefore, we pack 128 threads in an executor to provide high concurrency while maintaining utilization.

Task scheduling workflow in an SM. Based on the Dispatcher and Concurrent Executors, we present the task scheduling workflow inside an SM. As shown in gray blocks in Fig. 6, inside an SM we present the *Dispatcher* that runs as multiple persistent threads to fetch tasks and assign them to executors. We design an *Execution Table* which records necessary parameters for executing a task and the status of the current task. As shown in Fig. 7(a), the *STATUS* field records the status of a current task. $STATUS$ = 0 indicates that the task has finished, while $STATUS$ = 1 shows that the task with the ID NF_ID is being executed on packets *P_IN_PKTS* and will generate the output packets at *P_OUT_RETS*. Each thread in the Dispatcher assigns a task to an executor by configuring its corresponding entry in this table. The Dispatcher keeps visiting the *STATUS* field in the entry and safely assigns a new task by configuring NF parameters in the entry if $STATUS$ turns into 0. Finally, assigned tasks are executed on Concurrent Executors.

An intuitive approach for executors to receive tasks is letting them contend over all assigned tasks recorded in the Execution Table. In this way, the task fetched by a dispatcher can be finally executed on any executor. However, this *full-mapping* scheme could seriously degrade performance, as distributed task polling requires expensive synchronization and locking mechanisms. In contrast, as shown in Fig. 6, we maintain a *1:1 mapping* between the Execution Table entry and the Executor. Thus, a thread in Dispatcher can only assign a task to its corresponding Executor to avoid contention. After finishing a

task, the executor informs its Execution Table entry. Then it keeps querying the *STATUS* field of the entry until it turns into 1, which indicates that a new task is assigned. The executor then polls the task and performs execution.

C. Safe NF Task Assignment: NF Assignment Table

Before scheduling a task inside an SM, a thread in the Dispatcher has to fetch a task from workers in CPU. The number of workers is equal to the number of NFs, which might be large, while the number of threads in the Dispatcher is equal to the number of Executors, which is rather limited. A straightforward design is for workers to contend over the limited Dispatcher threads to assign tasks. However, such contention needs synchronization and may be expensive. To address the above challenges, Grus introduces *NF Assignment Table*, a data structure that works as the *task queue*, to enable safe task assignment. Fig. 7(b) depicts the table structures. When a worker attempts to launch an NF kernel with m executors, it first inserts an entry with *WORKER_ID* (its id), *NB_EXES* (m executors), *NF_ID* (which NF to execute), *NB_PKTS* (how many packets), *P_IN_PKTS* (the pointer to packets), and *P_OUT_RETS* (the pointer to output results). Then, it copies this entry to overwrite the corresponding slot in this table. As shown in Fig. 6, threads in the dispatcher fetch tasks from the *NF Assignment Table* and schedule them.

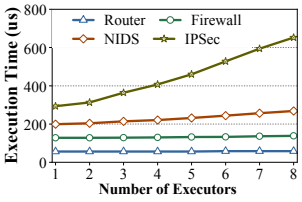
V. Scheduling Layer Design

In this section, we describe how Grus makes scheduling decisions in the scheduling layer to achieve latency SLOs for co-located NFs. We first build prediction models for processing time on different components. Next, we propose a heuristic scheduling algorithm to produce fast schedules.

A. Latency Prediction Modeling

Packet batching. As introduced in §III, the time for forming packet batches follows a near-linear relationship with batch size. Thus, we use a linear function to model the relationship between batch size and batching time. According to our evaluation in Fig. 3(a), as traffic rate changes, the linear function of batching time and batch size has different slope a_f and intercept b_f. Hence, our latency prediction model for packet batching is modeled as $T_{batching} = a_f \cdot batch\_size + b_f$. To obtain the value of a_f and b_f, we generate a wide range of traffic workloads, of which the rate is from 0.1 Gbps to 10 Gbps and increased by 0.1 Gbps. Under different traffic rate, we measure the batching time under different batch sizes and establish the latency model for packet batching.

Data transfer on PCI-E. Previous works [11], [12] have demonstrated that the time of data transfer on PCI-E can be modeled as a linear function with *batch size*. However, according to our evaluation in Fig. 3(c), the parameters of the linear model may vary significantly with data characteristics, such as the data size. To build a more accurate prediction model, we measure the latency under NF context in Grus. According to the data required for processing, we classify NFs into three categories including NFs that need (1) only header, (2) only payload, and (3) header and payload. For each category, we train a different slope d_{class} and intercept e_{class}, which is presented as $T_{transfer} = d_{class} \cdot batch\_size + e_{class}$.



(a) On a dedicated SM

NF_i \ NF_j	Router	Firewall	NIDS	IPSec
Router	4	12	17	20
Firewall	1	16	26	35
NIDS	0	1	113	158
IPSec	0	2	14	257

(b) Penalty (s) on a shared SM (all NFs use 4 executors)

Fig. 8: Profiling result examples of kernel execution time

Kernel execution on GPU. Suppose an NF task in GPU occupies one or more executors on a *specific* SM for execution. We refer to existing research [11], [12] and use a linear function to model the execution time of an NF kernel with the number of executors for this NF. We adopt performance profiling [8], [12] to get the parameters of this model. We profile the kernel execution time when using different numbers of executors for each NF. An example is shown in Fig. 8(a).

However, sometimes ensuring the throughput SLO of an NF requires massive concurrency, which exceeds the number of threads that one SM can provide. In this case, we have to divide a large batch of packets into multiple smaller batches (*i.e. minibatches*) and process them on different SMs. Mini-batches make it more complex to predict the kernel execution time since minibatches from different NFs may share one SM and therefore may suffer from latency overhead. To understand its effect, we profile the execution time of an NF kernel under all possible sharing cases with other NFs. Fig. 8(b) shows that the increased execution time due to SM can be easily profiled. Therefore, we introduce *penalty* (denoted as β_{ij}) to indicate the overhead incurred by SM sharing and model the kernel execution time as $T_{kernel} = (g \cdot executors + h) + \beta_{ij}$. Note that the execution time of an NF should be calculated as the longest processing time among all minibatches.

B. Scheduling Decision

Problem description. Given a set of SLOs $S = \{1, 2, ..., N\}$ where each term is associated with target latency L_i and expected throughput T_i, we need to decide which SLOs can be admitted on available resources. Moreover, we should generate the batch size, data transfer order, and executors for each NF. For each NF $i \in S$ in SLO_i, let b_i denote the batch size used for packet batching and data transfer, b_{im} denotes the size of the mth minibatch. We denote the time of packet batching as t_{1i}, the time of data transfer on PCI-E as t_{2i}, the kernel execution time as t_{3i}, the time of waiting on PCI-E as t_{4i}, and pre-/post-processing time on CPU as t_{PP}. For the scheduling problem, there are two binary decision variables,

$$y_i^n = \begin{cases} 1, & \text{if NF } i \text{ admitted with order } n \text{ to transfer data} \\ 0, & \text{otherwise} \end{cases}$$

$$x_{im}^k = \begin{cases} 1, & \text{if the } m\text{th minibatch of NF } i \text{ assigned on SM } k \\ 0, & \text{otherwise} \end{cases}$$

Suppose SMs are numbered from 0 to K, each with E executors. One executor can process c packets. We use a step function $U(x)$ as the utility function. $U(x) = 1$ if $x > 0$, otherwise $U(x) = 0$. The objective is to maximize the number of admitted latency SLOs with respect to resource constraints. If required latency SLO is larger than our predicated latency, the latency SLO is considered to be admitted. We formulate the problem as:

$$max \sum_{i \in S} U(L_i - t_{1i} - t_{2i} - t_{3i} - t_{4i} - t_{PP}) \quad (2)$$

s.t.

(1) $\sum_{m \in [1, M_i]} b_{im} = b_i$

(2) $1 \leq M_i \leq \frac{b_i}{c}$

(3) $\sum_{m,k} x_{im}^k = \sum_n y_i^n \cdot M_i$

(4) $\sum_{i \in S} y_i^n \leq 1, \sum_{n \in S} y_i^n \leq 1, \sum_k x_{im}^k \leq 1,$
$\sum_m x_{im}^k \leq 1, \forall n, i \in S, k \in [0, K]$

(5) $\frac{b_i}{L_i} \geq T_i$

(6) $\sum_{i,m} x_{im}^k \cdot b_{im} \leq E \cdot c, \quad \forall k \in [0, K]$

where

$t_{1i} = a_f \cdot b_i + b_f, \; t_{2i} = d_{class} \cdot b_i + e_{class}$

$t_{3i} = max\{g \cdot \frac{b_{im}}{c} + h + \beta_{ij} \cdot I_{\{x_{im}^k > 0\}} \cdot I_{\{x_{jt}^k > 0\}},$
$\forall k \in [0, K], j \in S, t \in [1, M_j]\}, \forall m \in [1, M_i]$

$t_{4i} = \sum_{p(j) < p(i)} t_{2j}, \forall j \in S$

Constraint (1) shows the relationship between the size of minibatches and required batch size. Constraints (2) $\sim$ (4) represent the scheduling requirements where M_i denotes the amount of minibatches of NF i. Constraint (5) satisfies the throughput requirement. Constraint (6) ensures that allocated executors of each SM does not exceed physical limit.

Scheduling decision should quickly identify SLOs that cannot be supported and respond the operators. However, the optimization objective is a piece-wise function, which makes it hard to find a solution within limited time [34]. In response, we propose an online heuristic algorithm for fast scheduling.

This algorithm exploits three major intuitions:

1) *Minimal batch size*. From Constraint (5), we can calculate a minimal batch size to satisfy the throughput SLOs. Minimizing batch size also helps reduce latency.
2) *Shortest headroom first*. Headroom [11] is the maximal waiting time that can be added to an NF without latency SLO violation. We first allocate transfer order and GPU executor for those SLOs with the shortest headroom.
3) *Penalty avoidance*. We prefer assigning an NF to one SM to avoid SM sharing penalty. If an NF is too large to be supported by any single SM, we split the NF to multiple SMs with the lowest penalty according to our modeling.

We show the online scheduling algorithm in Algorithm 1. The notations are the same as before and the headroom is denoted as T_{hdr}. For each SLO, we first calculate an optimal batch size (line 2) and calculate the headroom by subtracting the time of packet batching and pre-/post-processing from the latency SLO (line 3). Then, we consider SLOs in an increasing order of headroom (line 5) and refer to the *penalty avoidance* intuition when allocating executors for them (line 7). Finally we subtracts execution time in GPU from headrooms and decide the data transfer order of NFs (lines 8-21).

Algorithm 1: Scheduling algorithm

input : $(L; T); (f_{t1}; f_{t2}; f_{t_{PP}}; profiles)$ - SLO terms, prediction models and profiling results.
output : $(S_{accepted}; S_{batch}; S_{order}; S_{SM}; S_{exes})$ - Whether to accept, batch size, order, assigned SMs and executors

1 **foreach** $i \in SLOs$ **do**
2 $\quad S_{batch}[i] = T_i * L_i$; // An minimal batch size
3 $\quad T_{hdr}[i] = L_i - f_{t1}(T_i; S_{batch}[i]) - f_{t_{PP}}(i)$;
4 // Shortest headroom first
5 **foreach** $i \in T_{hdr}$; *in increasing order of hdr value* **do**
6 $\quad$ // Penalty avoidance
7 $\quad (S_{SM}[i]; S_{exes}[i]; t_{kernel}) \leftarrow$ *selecting SMs and executors, meanwhile getting the kernel execution time from profiles*
8 $\quad T_{hdr}[i] -= t_{kernel}$;
9 $\quad$ **for** $order = i$ **to** 0 **do**
10 $\quad\quad t2 = f_{t2}(S_{batch}[i])$;
11 $\quad\quad t2_w \leftarrow \sum_{k=0}^{order-1} f_{t2}(S_{batch}[k])$;
12 $\quad\quad hdr = T_{hdr}[i] - t2 - t2_w$;
13 $\quad\quad$ **if** $hdr >= 0$ ***and*** *NoViolations(order, i, t2)* **then**
14 $\quad\quad\quad accepted = true$;
15 $\quad\quad\quad$ **break**;
16 $\quad$ **if** $accepted == true$ **then**
17 $\quad\quad S_{accepted}[i] \leftarrow$ true;
18 $\quad\quad S_{order}[i] \leftarrow order$;
19 $\quad\quad$ **for** $j = order + 1$ **to** i **do**
20 $\quad\quad\quad S_{order}[j] + +$;
21 $\quad\quad\quad T_{hdr}[j] -= t2$;
22 **function** *NoViolations* ($start; end; t2\_w$)
23 $\quad$ **for** $i = start$ **to** end **do**
24 $\quad\quad hdr = T_{hdr}[i] - t2\_w$;
25 $\quad\quad$ **if** $hdr < 0$ **then**
26 $\quad\quad\quad$ **return** false
27 $\quad$ **return** true

VI. Discussion of Grus Limitations

In this section, we discuss two major limitations of Grus.

Grus does not handle PCI-E contention during data transfer from GPU to CPU. After GPU processing, data are copied from GPU to CPU over PCI-E for further processing or transmission through the NIC. If multiple tasks finish execution in GPU at roughly the same time, data transfer may suffer from PCI-E contention and latency variation.

To handle this contention, an intuitive approach is to schedule the transfer order of in-GPU tasks according to the time when they finish execution (denoted as T_{finish}). Theoretically, T_{finish} is the sum of (i) the start time of data transfer from CPU to GPU, (ii) data transfer time from CPU to GPU, and (iii) task execution time in GPU. Therefore, estimating T_{finish} suffers from accumulated estimation inaccuracy of these three latency components. *Using this inaccurate T_{finish} for scheduling may result in a bad scheduling plan that brings even larger latency variation than the situation without scheduling*. For example, suppose the CPU schedules data transfer of kernel A before kernel B, but kernel B still finishes first. Kernel B will need to wait for kernel A to finish and transfer data before it initiates data transfer, which introduces larger latency variation. Therefore, current Grus design does not handle PCI-E contention from GPU to CPU. We will carefully handle this contention in future work.

Grus focuses on guaranteeing latency for single NFs. Grus

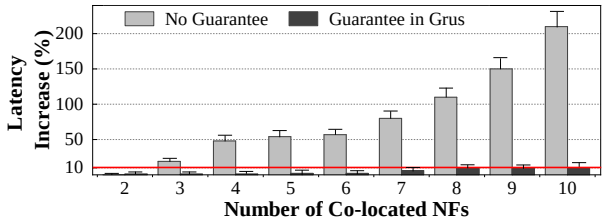


Fig. 9: Average latency increase of the baseline firewall

focuses on providing latency SLOs for a single NF but, in its present form, it will not enable latency SLOs for entire service chains that may span multiple NFs. We consider Grus only as a first step towards a fully fledged system for guaranteeing SLOs for service chains. Conceptually, this may be solved by incorporating the NF-to-NF packet forwarding time in our latency model (Eqn. 1); we leave this as future work herein.

VII. Implementation and Evaluation

We have implemented a prototype of Grus. The infrastructure layer uses DPDK for networking I/O and CUDA as the programming toolkit for GPU. The scheduler is written in the C language. We implement the scheduler on the same server as the infrastructure. It is easy to migrate this scheduler to other servers that can communicate with the infrastructure server. We have implemented four NFs including *IPv4Router*, *Firewall*, *NIDS*, and *IPSec*. Key logics of these NFs are five-tuple hash value matching, trie-based pattern search, Aho-Corasick search, and HMAC-SHA1 & AES-128, respectively.

Experimental setup. Currently we run Grus and NFs on one server equipped with two Intel Xeon E5-2650 v4 CPUs (2.20 GHz, 12 physical cores), 128GB total memory (DDR4 2400 MHz 16GB x8), two dual-port 10G NICs (Intel X520-DA2), and an NVIDIA TITAN Xp (30SMs, 3840 cores, 12GB G5X memory). For test traffic, we implement a traffic generator on a separate server, which sends and receives packets that follow the flow size distribution of data center traffics [17]. The server for the generator has the same configurations as the previous one. Both servers run Ubuntu 14.04 (with kernel 3.16.0-30), DPDK version 17.11, and CUDA version 8.0.

SLO generation. SLOs are defined in different styles across different service providers, making it hard to decide universally recognized SLOs for evaluation. As mentioned in § I, the key to an SLO is guaranteeing performance variation. Therefore, similar to ResQ [8], we generate the target latency of each SLO term as a tolerated increase (denoted by *tolerance*) of the baseline latency of a target NF in our evaluation. i.e., *target_latency* = *baseline_latency* * (1 + *tolerance*). The baseline latency is the latency of when an NF runs solely.

Evaluation goals. We evaluate Grus with the following goals: (1) the effectiveness to stabilize the latency of multiple co-located NFs (§VII-A); the accuracy of latency prediction models (§VII-B); and (3) the improvement of resource efficiency and fast scheduling (§VII-C).

A. Effect on Reducing Latency Variation

We first evaluate that Grus can reduce latency variation of an NF when other NFs co-locate in the same machine. We use the same setup and traffic workloads as §II-A. We run the initial firewall with *order* 0 running on two *executors* in SM#0. Fig. 9 shows the increase in average latency with different numbers of co-located NFs. We observe that the latency variation is significantly reduced in Grus. We observe that even if 9 other NFs are running together with the initial firewall, its average latency increase is below 10%. In comparison, when there is no latency guarantee, introducing two additional NFs can increase the latency of the initial firewall by 19%. This demonstrates the effectiveness of Grus to guarantee latency.

Note that even though we try to reduce the latency variation as thorough as possible, there still exists slight variation as more NFs co-locate. This is because the reordering engine is non-preemptive, which implies that even an NF with *the first order* submits a data transfer task over PCI-E, it cannot directly capture the PCI-E. Instead, it still has to wait for the completion of previous tasks.

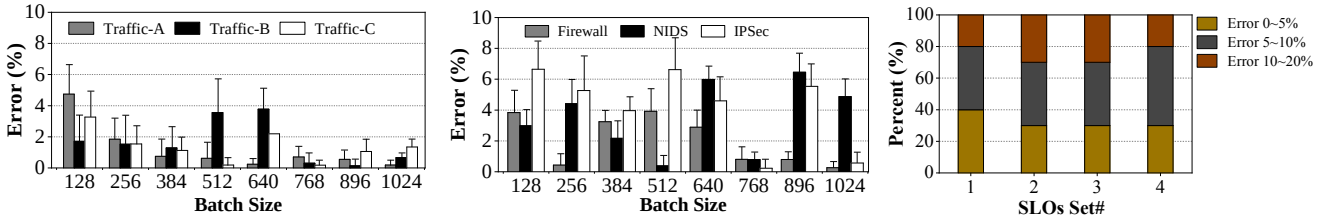
B. Latency Prediction Accuracy

Latency prediction accuracy is critical for generating accurate scheduling decisions. We evaluate the accuracy of the latency model for packet batching and data transfer, respectively. Evaluation for kernel execution accuracy is presented in Fig. 8. Furthermore, we evaluate the prediction accuracy of overall end-to-end latency when multiple NFs co-locate.

Packet batching latency predication. We evaluate the predication accuracy using *three sets of real-world traces*. *Traffic-A* is the trace from CAIDA recorded in 2016 from an ISP backbone link [35]. *Traffic-B* is sampled from a Facebook open-source Hadoop cluster data [36]. Traffic-C is a private trace collected from the gateway in a large enterprise data center. We vary the batch size and predict the batching latency. We input the three traces into Grus infrastructure to measure real batching time as the baseline. As shown in Fig. 10(a), we observe a reasonable average predication errors of 1.2%, 1.6% and 1.3% in average for the three traces. The accuracy varies modestly (*e.g.*, from 0.2% to 4.7% for Traffic-A) due to flaws in the data traces (e.g., badly-distributed samples).

Data transfer latency predication. As mentioned above, Grus classifies NFs into three types according to the packet fields they process (headers, payloads, and headers+payloads), and build three predication models for them respectively. Therefore, we evaluate the data transfer (two directions) predication accuracy by using three representative NFs: Firewall for *header-only*, NIDS for *payload-only*, and IPSec for *header+payload*. We use Traffic-A as the input traffic. Fig. 10(b) presents the average prediction error when configuring different batch sizes for the three NFs. We observe that our models are able to accurately predict the time of data transfer across three NFs with a deviation of 2.2%, 3.5%, and 4.1% in average compared with the real transfer time.

End-to-end latency predication. Prediction accuracy for end-to-end latency is critical to avoid SLO violations when consolidating multiple SLO terms. To evaluate its accuracy, we generate four sets of latency SLOs each with 10 terms. For each term, we randomly select a target NF, and randomly



(a) Average prediction error of packet batching (b) Average prediction error of PCI-E data transfer (c) Average prediction error of end-to-end latency

Fig. 10: Average prediction error of packet batching, data transfer on PCI-E and end-to-end latency

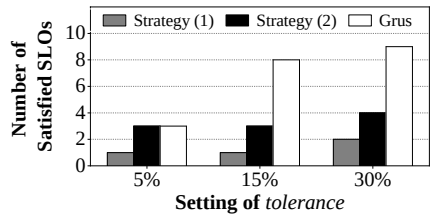


Fig. 11: Maximal satisfied SLOs

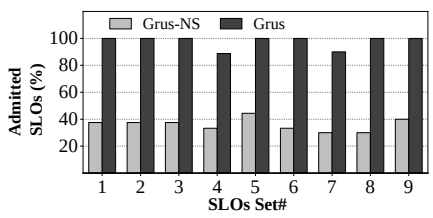


Fig. 12: Number of admitted SLOs

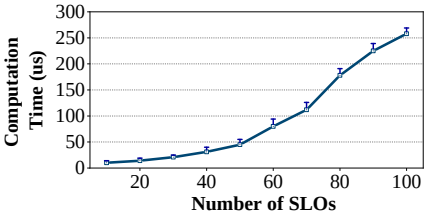


Fig. 13: Average calculation time

set target latency by setting *tolerance* from 5% to 15%. The baseline latency is measured when these NFs run with batch size 256 on Traffic-A. Fig. 10(c) shows the average prediction errors in four SLOs sets. We observe that Grus suffers from very low prediction error for end-to-end latency. For example, in SLOs Set#1, only 20% SLO terms (i.e., two terms) have 10-20% deviation from the real end-to-end latency. Note that although deviation exists, it does not imply that Grus cannot provide guaranteed end-to-end latency. The Grus scheduler adds these deviations to corresponding SLO terms.

C. *Efficiency of* Grus *to Support Latency SLOs*

Grus can support more SLOs than existing solutions. With different latency targets (i.e., differentiated service levels), two alternative strategies have the potential to enable latency SLOs: (1) reordering kernel executions to satisfy the performance requirement of QoS-required applications (i.e., solution from Baymax [11]), and (2) spatially sharing a GPU and allocating SMs to multiple NFs to achieve predictable performance (i.e., solution from G-NET [12]). To demonstrate the efficiency in Grus as opposed to them, we implemented both strategies in the system we used in §II-A. we randomly generate 9 sets of SLOs, and mark them from number 2 to 10. We vary the *tolerance* value of all SLOs. We measure the number of maximal satisfied SLOs in each setting. Fig. 11 shows that Grus can support more SLOs than both strategies. Even with *tolerance* 30%, it is able to support 4.5× more SLOs than Strategy (1) and >2× than Strategy (2). Note that for Strategy (2),when the number of SLOs is larger than 7 in all settings, all SLOs suffer violations due to the lack of scheduling for multiple NFs (not shown in the figure).

The improved efficiency in Grus is by enabling slicing GPU resources into fine-grained executors to improve concurrency and resource utilization efficiency. To demonstrate this, we randomly generate 9 sets of SLOs. We use a different number of SLOs in each set, i.e., 8 terms for SLOs set 1-3, 9 terms for SLOs set 4-6, 10 terms for SLOs set 7-9. Fig. 12 shows the ratio of admitted SLOs. We denote the traditional no-slicing solution as Grus-*NS*. In all SLOs sets, Grus-NS only admits a small fraction of them, *i.e.* less than 50% in all cases. In comparison, Grus admits over 90% of these terms. The reason for such a gap is that Grus utilizes all available resources in an SM and carefully places different NFs in one SM to avoid SLO violations with improved resource utilization efficiency.

Grus can quickly generate scheduling plans. To perform fast scheduling, we propose a heuristic algorithm for Grus scheduling. Fig. 13 shows that the average computation time of this algorithm is below 300 μs when handling 10 to 100 SLO terms (100 is large enough as the number of NFs that can be supported in one server [1]), which demonstrates that the Grus scheduler can perform fast SLO admission control.

VIII. Related Work

Guaranteeing end-to-end latency. Many efforts focused on providing latency guarantee, such as Internet QoS [37], performance isolation in datacenter and cloud [10], [18], [38]–[40]. They worked well in the context where applications share network, CPU and storage resources. However, they cannot be directly applied to the context GPU-accelerated NFV. A few works [8], [19], [27] proposed solutions to enable performance isolation for packet processing in NFV. But they become ineffective when introducing GPUs into NFV. G-NET [12] presented a model to achieve predictable latency for single GPU-based NFs. However, it can not guarantee latency when multiple NFs co-locate on the same server.

Predictable latency on GPUs. To reduce the response latency of real-time or user-facing applications, the GPU community has proposed abundant works to enable QoS support for these applications. They enabled this by introducing preemption primitives via new hardware design [41]–[44], software framework [11], [20], [45], or scheduling [24], [46], [47]. However, as illustrated in §II-A, these solutions are not suitable for NFs due to a much finer granularity at latency target. Furthermore, previous works did not jointly consider all latency components including CPU, PCI-E, and GPU.

Predication of in-CPU processing time and data transfer time. ResQ [8] and Dobrescu et al. [27] have proposed using

predicted LLC allocation to predict latency of CPU-driven NFs. However, our experiments show that the contention in LLC has slight impact on GPU-accelerated NFs. Some works [11], [20] used a linear model to predict the duration of data transfer on PCI-E and kernel execution for general-purpose applications. Grus also uses a linear model to predict them; nevertheless, we specify these models in NFV context to improve accuracy. G-NET [12] also provided performance models for data transfer and kernel execution. However, it could not predict latency when multiple NF kernels co-locate in the same GPU and the latency for of packet batching.

IX. Conclusion

We have presented Grus, a GPU-accelerated NFV system that enables latency SLOs for multiple co-located NFs. We present the *infrastructure design* of Grus to support controllable concurrent executors for the NF kernel and reorder data transfer over PCI-E . Moreover, Grus introduces a *Latency SLO-aware scheduler*, which takes our latency prediction models and SLO terms as input to accurately and quickly maximize the admitted SLOs. Our evaluations have demonstrated the effectively and efficiently of Grus to support latency SLOs.

X. ACKNOWLEDGEMENT

We thank our shepherd Gabor Retvari and anonymous reviewers for their thoughtful feedback. This work is supported by the National Key R&D Program of China (2017YFB0801701), and the National Science Foundation of China (No.61472213). Jun Bi (junbi@tsinghua.edu.cn) is the corresponding author.

References

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

[2] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in nfv," in *SIGCOMM*, 2017.

[3] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon, "Nba (network balancing act): a high-performance packet processing framework for heterogeneous processors," in *EuroSys*, 2015.

[4] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *SIGCOMM*, 2010.

[5] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: a highly-scalable software-based intrusion detection system," in *CCS*, 2012.

[6] J. Park, W. Jung, G. Jo, I. Lee, and J. Lee, "Pipsea: A practical ipsec gateway on embedded apus," in *CCS*, 2016.

[7] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "Gaspp: A gpu-accelerated stateful packet processing framework." in *ATC*, 2014.

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

[9] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *SOSP*, 2015.

[10] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," in *SIGCOMM*, 2015.

[11] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ASPLOS*, 2016.

[12] 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.

[13] Wikipedia, "Pci express," 2014. [Online]. Available: https://en.wikipedia.org/wiki/PCI_Express

[14] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "Midea: a multi-parallel intrusion detection architecture," in *CCS*, 2011.

[15] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using gpus in software packet processing." in *NSDI*, 2015.

[16] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "Apunet: Revitalizing gpu as packet processing accelerator." in *NSDI*, 2017.

[17] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *IMC*, 2010.

[18] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!" in *NSDI*, 2015.

[19] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-resource fair queueing for packet processing," *SIGCOMM*, 2012.

[20] B. Wu, X. Liu, X. Zhou, and C. Jiang, "Flep: Enabling flexible and efficient preemption on gpus," in *ASPLOS*, 2017.

[21] D. Singh and C. K. Reddy, "A survey on platforms for big data analytics," *Journal of Big Data*, 2015.

[22] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, 2016.

[23] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: Fine-grained gpu resource virtualization for narrow tasks," in *PPoPP*, 2017.

[24] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *ATC*, 2011.

[25] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "Stackmap: Low-latency networking with the os stack and dedicated nics." in *ATC*, 2016.

[26] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," in *TNSM*, 2015.

[27] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packet-processing platforms," Tech. Rep., 2012.

[28] Intel, "Cat," 2016. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology

[29] PCM, "Pcm," 2017. [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[30] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," in *ASPLOS*, 2018.

[31] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," in *SIGCOMM*.

[32] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *Innovative Parallel Computing (InPar)*, 2012.

[33] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming," *ISCA*, 2016.

[34] C. Sun, J. Bi, Z. Meng, X. Zhang, and H. Hu, "Ofm: Optimized flow migration for nfv elasticity control," in *IWQoS*, 2018.

[35] CAIDA, "The caida ucsd anonymized internet traces 2016," 2016. [Online]. Available: http://www.caida.org/home/

[36] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *SIGCOMM*, 2015.

[37] D. D. Clark, S. Shenker, and L. Zhang, "Supporting real-time applications in an integrated services packet network: Architecture and mechanism," in *SIGCOMM*, 1992.

[38] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska, "End-to-end performance isolation through virtual datacenters." in *OSDI*.

[39] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters," in *EuroSys*.

[40] P. L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection." in *NSDI*.

[41] NVIDIA, "Cuda compute preemption," 2018. [Online]. Available: https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#preemption

[42] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *ISCA*, 2014.

[43] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared gpu," *ASPLOS*, 2015.

[44] Z. Lin, L. Nyland, and H. Zhou, "Enabling efficient preemption for simt architectures with lightweight context switching," in *SC*, 2016.

[45] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "Effisha: A software framework for enabling effficient preemptive scheduling of gpu," in *PPoPP*.

[46] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time gpu management," in *RTSS*, 2013.

[47] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Quality of service support for fine-grained sharing on gpus," in *ISCA*.