FlexVF: Adaptive network device services in a virtualized environment

Brice Ekane, Dinh Ngoc Tu, Boris Teabe Djomgwe, Daniel Hagimont, Noel de Palma

FlexVF: Adaptive network device services in a virtualized environment

Brice Ekane[a], Tu Dinh Ngoc[b], Boris Teabe[b], Daniel Hagimont[b], Noel De Palma[a]

[a] *University of Grenoble, France*
[b] *University of Toulouse, France*

Abstract

Single-root I/O virtualization (SR-IOV) allows virtual machines direct access to physical network cards through so-called virtual functions (VFs), considerably reducing networking overhead compared to paravirtualized interfaces by avoiding the hypervisor's network stack. However, the maximum number of VFs on each card is often limited compared to the number of VMs running on each host, leading to the problem of choosing which VMs to allocate these VFs. In this paper, we introduce **FlexVF**, a mechanism for dynamically allocating and switching between VFs and paravirtualized networking on VMs based on network activity monitoring. We show that **FlexVF** improves VM network performance by 75% without affecting network operation.

1. Introduction

Virtualization has become a must in both private and public clouds such as Amazon, Microsoft and Google. It allows sharing of hardware resources between virtual machines (VM) using a system software called a hypervisor. Hypervisors use several techniques for sharing these resources among all VMs: a scheduler for the CPU, a memory management unit (MMU) for RAM, and a set of techniques for the I/O devices which we detail below.

Hypervisors expose accesses to I/O devices via one of three main mechanisms: (1) *device emulation* of a particular hardware model, such as a network card (e.g Intel's "e1000" network adapters); (2) *paravirtualized interfaces*, where the hypervisor exposes an optimized API to the VM for accessing the device through a common transport [1], (e.g. through a shared memory-mapped region); and finally via (3) *device passthrough*, where the VM is granted direct access to a physical device, which is in turn isolated from the rest of the system using an *I/O memory management unit* (IOMMU). These techniques listed above have different tradeoffs and are not applicable to all virtualization workloads. For example, device emulation is mostly used to ensure compatibility with legacy operating systems (OS) that are not virtualization-aware. Paravirtualized interfaces perform faster than device emulation thanks to virtualization interface-specific optimizations; however, they still impose an overhead by needing to communicate with the hypervisor and PV device model for every I/O request, and require the guest OS to be aware of paravirtualization extensions available on the host hypervisor.

In contrast, device passthrough requires giving the guest exclusive access to a hardware device. This means that the number of guests that can benefit from passthrough is constrained by the number of devices installed in the host. In the particular case of PCI passthrough, the restriction is per PCI physical function; for example, each 2-port ethernet card having 1 physical function (PF) per port, and therefore 2 PFs in total, can only be assigned to a maximum of 2 VMs. To overcome this restriction, the PCI Express specification defines **single-root input/output virtualization** (SR-IOV), where one PF can be partitioned into multiple virtual functions (VF). These VFs can then be assigned separately to different VMs, increasing the number of VMs that can utilize passthrough I/O on each host. However, implementing SR-IOV is not without cost. First of all, datacenter operators need to select the correct network card for their requirements, while taking into account factors such as SR-IOV support, number of VFs needed for their workload, etc. Secondly, SR-IOV requires additional logic in the hardware for each VF, therefore limiting the maximum number of VFs that can be implemented on each device [2]. This limitation leads to a VF allocation problem, namely: **in a situation with more VMs needing VFs than is available, which VMs should be allocated a VF?**

We conducted a study on the servers of two research data centers: Grid5000 [3] and CloudLab [4]. We found that the majority of servers are equipped with one SR-IOV-enabled dual-port Ethernet card for network communication, and that most cards have support for up to 64 VFs per network port. On Grid5000 servers in particular, out of these two ports, one port is used exclusively for host administration and management purposes, leaving only one port (and therefore 63 VFs on their Ethernet card model) available to all VMs that could be executed on the host. At the same time, these servers are installed with anywhere from 250 GB to 768 GB of memory, leaving plenty of resources for VM usage. Microsoft Research showed that 70% of VMs running on Azure datacenters have less than 3 GB of RAM [5]; this figure is especially true regarding serverless application hosts, where one server can host thousands of micro-VMs each with little CPU and memory resources [6].

In short, in a virtualized datacenter with servers providing large amounts of RAM, considering most of the time RAM

⋆⋆`https://doi.org/10.1016/j.future.2021.08.011`

capacity is the limiting factor for packing VMs on hosts, it is likely that each physical server will host more VMs than the number of VFs available on said host (e.g. 63 VFs). As a result, with the current trend of increasing core counts, RAM sizes and VM density, **we would need to decide which VM to equip with a passthrough device, and therefore better I/O performance**.

In fact, public cloud services already advertise different price tiers for different levels of network performance on their cloud VMs. Features like AWS enhanced networking [7] or Azure accelerated networking [8] directly expose SR-IOV-enabled network cards to the cloud guest at an increased price compared to the standard offering. However, this pricing model is not always applicable to private cloud scenarios; instead, it's more beneficial to assign these resources on a case-to-case basis depending on application demands in order to optimize general application performance across the whole datacenter. Nevertheless, private cloud administrators may lack the necessary tools to sufficiently understand running applications in VMs, as well as the necessary tooling and know-how to allocate VFs to VMs automatically as the workload changes.

To answer this problem, we present a flexible and generic framework named **FlexVF** that allocates VFs to VM based on their network activity. To this end, **FlexVF** allocates to each VM two virtual network interfaces: one paravirtualized interface and a VF configured using device passthrough. During the lifetime of the VM, **FlexVF** monitors its network activity, decides which network virtualization technique is appropriate for the current utilization levels, and then activates the corresponding interface. Building such a platform raised several challenges, namely the mechanism for monitoring a VM's network activity on a VF knowing that it has a direct access to the hardware, the fast hot-plugging and unplugging of network interfaces to VM, and finally avoiding impacting application performance during network reconfiguration. Furthermore, we designed **FlexVF** as a generic framework that can be integrated into private cloud platform such as OpenStack for automating VM resource allocation. We systematically evaluated each aspect of our framework using both micro- and macro-benchmarks; we find that **FlexVF** can help improve network application performance by up to 75% in virtualized environments.

The rest of the paper is structured as follows. Section 2 presents an overview of networking in virtualization. Section 3 presents the motivations of the work. Section 4 presents the general overview of **FlexVF**. Section 5 presents the implementation of **FlexVF**. Section 6 presents the evaluation results. Section 8 discusses the related works. Section 9 concludes the paper.

2. Background

As stated in the previous section, hypervisors provide access to I/O devices using three main techniques: device emulation, paravirtualization, and device passthrough. In this section, we detail the implementations and characteristics of these interfaces, and assess their suitability in various I/O scenarios.

2.1. Device emulation

Device emulation aims to support an OS's I/O needs by mimicking the behavior of a certain hardware device. With this technique, the hypervisor needs to trap and emulate accesses to hardware resources to perform tasks such as managing virtual hardware states, delivering interrupts, responding to I/O from the guest OS, and so on. The emulation itself is often performed not directly inside the hypervisor kernel, but rather in a userspace process to minimize the kernel code base and separate the error-prone emulation component from the rest of the system. In general, the main goal of device emulation is not performance, but to closely follow the device model that it's imitating, in order to support legacy operating systems that lack knowledge of virtualization extensions. As a result, device emulation is naturally more complex and slower than its alternatives, and no longer used in high-performance I/O scenarios.

2.2. Paravirtualization

As opposed to device emulation, paravirtualization (PV) aims to provide a simple and performant I/O interface to VM guests. To this end, a virtualization-aware guest OS performs I/O requests through specialized channels defined by the hypervisor. These requests are then interpreted and serviced by the hypervisor's corresponding I/O component. Note that while a paravirtualized device might not resemble any particular hardware model, it is based on the same set of hardware resources as any other device class; for example, paravirtualized network devices often communicate with the guest by appearing as a virtual PCI interface, configured using the typical PCI configuration mechanisms, and exchanges data with the guest using shared memory-based "DMA" and virtual interrupts. Using these optimization techniques, paravirtualization mostly solves the problem of I/O performance in virtual machines; however, I/O requests to a PV device still need to be serviced by the hypervisor rather than by hardware. For example, every time a VM needs to send a network packet, this packet needs to pass through the hypervisor's network stack before it's received by the physical NIC. Consequently, these interfaces still cannot match the performance of a bare-metal OS running directly on hardware.

2.3. Device passthrough

Device passthrough gives guest VMs direct exclusive access to the underlying hardware. It is often used with devices needing high I/O performance, for example PCI Express network interfaces supporting high bandwidth. With device passthrough, the recipient guest VM directly communicates with the hardware using its own drivers, bypassing most of the costs associated with hypervisor-based trap-and-emulate during I/O operations. The hypervisor remains actively involved for certain management tasks; for a PCI device, this involves namely setting up the PCI configuration space, managing DMA, and brokering interrupt delivery into the VM's virtual interrupt controller. In general, device passthrough helps guests bypass the host OS's I/O stack, bringing virtual I/O performance in line with bare-metal environments.

As stated above, device passthrough requires exclusive access to the underlying hardware. As a result, a device being used in a passthrough configuration cannot be shared among multiple running VMs. The PCI Express specification defined single-root I/O virtualization (SR-IOV) as a solution to this issue, allowing each physical PCI function (e.g. network port) to appear as multiple PCI virtual functions (VF). Each VF is an isolated PCI function instance that can be assigned to a different VM and can communicate with the guest using the same driver as with the physical device. Note that VFs hosted by a PF all share the same physical capabilities of the PF; for example, each VF on the same 10 Gb Ethernet port only has a portion of the total 10 Gb link as dictated by the network card.

3. Assessment

We carried out evaluations to estimate application performance using the previously-mentioned network device virtualization techniques. Section 6 presents more details on the experimental environment, including hardware and software configurations. The experiment consists of executing a network benchmark in a VM with 4 GB of memory, 4 vCPU and a network device provided using each of the mentioned techniques. We used the results of the Xapian benchmark from TailBench [9], a benchmark suite for latency-critical applications providing request latency numbers under various workloads. Figure 1 presents the 95th percentile of request latency as obtained from Xapian, an online search workload that performs queries on a English Wikipedia-based corpus. We first observe that SR-IOV-enabled VMs equipped with a VF performs better than any other technique, whether PV (+22.7%) or device emulation (+30%), reaching close to bare-metal performance.

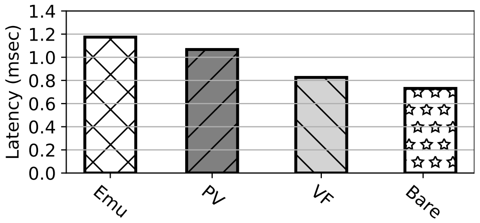


Figure 1: Xapian latency (95th percentile) using various network virtualization techniques.

With this result, using VFs on all VMs would therefore be the ideal configuration for best performance; however this is not possible given the limited number of VFs available on network cards. For instance, most 10GbE Intel cards only support up to 63 VFs per port [10]. Our study of hardware configurations of Grid5000 and CloudLab datacenters revealed that most servers on these datacenters are equipped with only one dual-port Ethernet card with up to 63 VFs per port; furthermore, one of these ports is commonly dedicated to administration operations, leaving only one port for use by VMs, or only 63 VFs. Knowing that each server can host hundreds of small VMs in a datacenter environment [5], this number of VFs is insufficient to cover all running VMs; naturally, we must consider the problem of which VMs can benefit from SR-IOV.

In public cloud environments such as Amazon AWS or Microsoft Azure, the allocation of VF is based on price tiers, where customers can choose between cheaper VM sizes with paravirtualized networking or more expensive VMs with SR-IOV enabled. In a private cloud, this distinction does not exist; instead, the cloud administrator must choose which VMs to equip with VFs and which not to. Moreover, the performance gain from SR-IOV compared to PV networking depends on the network activity of the application running in the VM. For example, a VM running an application with little network activity will benefit less from having a VF assigned to it than another running an application with significant network activity. To demonstrate this, we carried out evaluations to estimate the performance gain from VFs compare to PV networking. We used various TailBench workloads, including Xapian, Sphinx (a speech recognition workload), Masstree (C++-based key-value store), Silo (a in-memory database) and img-dnn (a handwriting recognition application). We set up TailBench in its networked configuration, where each application under test runs on a VM configured with VFs or PV, and with the application client (load generator) running on a separate physical machine. Figures 2 and 3 respectively present each benchmark's packet rate and the performance gain from switching to VFs. We note a large variability in results depending on the workload; for instance, Masstree shows a performance gain of 12% compared to only 1.6% on the same configuration with Sphinx, which almost parallels the packet rates emitted by these two benchmarks. We therefore conclude that not all VMs will benefit equally from the addition of SR-IOV, and therefore VF allocation should be carefully considered based on each VM's current workload.

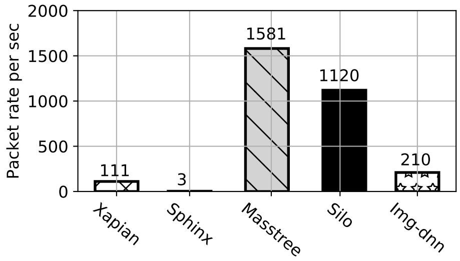


Figure 2: TailBench benchmark packet rates.

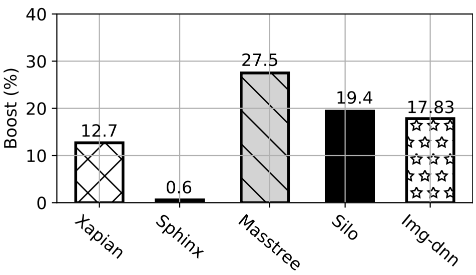


Figure 3: TailBench benchmark boost% from SR-IOV.

To answer the question of how to allocate VFs to VMs to optimize network performance, we describe our contribution **FlexVF** in the following section. **FlexVF** allows a cloud administrator to dynamically apply a VF allocation policy to VMs depending on their network activity.

4. Contributions

This section describes **FlexVF**, our framework for dynamically allocating VFs to VMs. With this goal in mind, we designed **FlexVF** using the following set of criteria:

- **Low intrusiveness. FlexVF** shall not require extensive changes to guest operating system kernels.
- **No application modification.** The platform must not require any modifications of networked applications running inside VMs.
- **Low overhead. FlexVF** shall have minimal overhead on VM performance.
- **Easily integrated.** We aim to design **FlexVF** so that it is easily integrated into an infrastructure-as-a-service manager platform such as OpenStack [11].

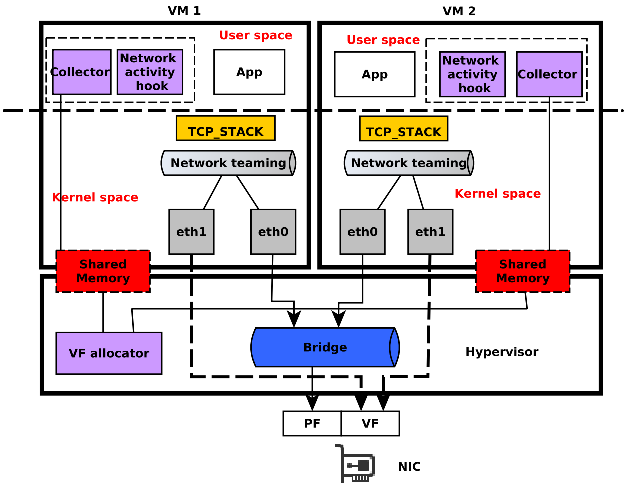


Figure 4: General overview of **FlexVF** components.

4.1. *General overview*

The main idea behind **FlexVF** is monitoring VM network activity and responding by dynamically hot-plugging/unplugging VFs. To make this reconfiguration transparent to networked applications, we combine multiple paravirtualized (PV) and VF-based network interfaces using a single synthetic *network team*, through which the VM can communicate with other hosts regardless of which interface type it is using. Figure 4 presents the two main parts of **FlexVF**'s general architecture: a VM-side subsystem to monitor its network activity and make this information available to the hypervisor, and a hypervisor-side component for selecting and allocating VFs to the selected VMs. The rest of the section presents these two elements in detail.

4.2. *FlexVF in VMs*

FlexVF's VM-side subsystem consists of three components:

Network teaming. This component aggregates multiple network interfaces on each VM into a single logical *teamed interface*. On Linux, network teams can operate in multiple modes: active/backup, multiple load balancing modes or link aggregation. We configure VMs to use network teams in active/backup mode to prioritize VFs for network transfers whenever available. This component helps ensure VMs' network connectivity even in cases where the VF is disconnected from the VM, for example during VM migration as described in [12].

Network activity hook. This component is a tracing hook installed in each VM's to monitor its network activity. The hook extracts network statistics such as packet rate, then transmits this information to the collector for processing.

FlexVF collector. Each VM's collector gathers network activity data from its corresponding hook, computes aggregate network statistics, and subsequently makes this information available to the hypervisor through shared memory for use by the VF allocator. The **FlexVF** collector is also responsible for following any commands issued by the host during network reconfiguration.

4.3. *FlexVF in the hypervisor*

FlexVF has a single component in the hypervisor, called the *VF allocator*. Its role is twofold: firstly to select VMs for VF allocation, and secondly to hot-plug/unplug VFs to these VMs. The VF allocator runs entirely in the userspace (as a background daemon in the KVM host), and requires no modification to either KVM or QEMU, thus fulfilling our requirement of low intrusiveness to the running hypervisor host.

VM selection. While network adapters with SR-IOV such as those from Intel already implement QoS mechanisms such as round-robin queues and rate limiting [13], the choice of which VM to bestow a VF can potentially impact the performance of other VMs if these mechanisms are not correctly configured in a situation of heavy network contention. To provide system operators with an additional form of traffic prioritization, **FlexVF** implements two strategies for VM selection, which the system operator can choose from depending on their system requirements:

– *Fair allocation.* In this allocation strategy, VFs are allocated in a round-robin fashion between all VMs on the host. In other words, each VM will have access to VFs for the same amount of time. This strategy is useful in preventing starvation, where multiple VMs with high network traffic demand compete for access to a small number of VFs. For example, in coflows that require barrier synchronization (e.g. MapReduce), the slowest flow decides the completion of the entire coflow [14]. In such cases, a fair allocation strategy ensures that all flows can progress at the same rate, therefore avoiding the starvation issue.

– *Activity-based allocation.* Section 3 showed our initial assessment of networking performance with VFs as compared to with PV interfaces, namely that applications with higher network utilization (i.e. higher packet rate) benefits more from SR-IOV than those with lower packet rates. To reiterate, we assume that network latency is linearly correlated with potential performance improvements from VFs, with minimal impact from packet sizes. Following this observation, we created an activity-based VF allocation strategy that aims to optimize network-heavy applications by considering the network activity of each VM, therefore establishing a ranking of VMs based on their potential need for network performance. We further evaluate and validate our assumption in Section 6.

With the above goal in mind, the ranking of VM network activity is simple. We sort the running VMs by descending order of packet rate; assuming the host has access to *N* VFs, we simply assign one VF to each of the *N* VMs with the highest packet rate. In other words, VMs running applications sending many small packets will be prioritized by the strategy, in contrast to applications sending few large packets which are less penalized by using PV networking. This strategy is aimed towards maximizing network utilization, and is useful in cases where a small number of VMs utilize most of the available network resources (e.g. in a large datacenter hosting multiple application types).

VF hotplugging. Once the VM selection is done, the VF allocator instructs the hypervisor to hot-plug/unplug VFs from VMs, using a simple rule: with *N* VFs available on the host, the *N* selected VMs will get access to one VF each. The selection is periodically refreshed after a configurable interval to ensure an up-to-date distribution of networking resources.

Algorithm 1 General algorithm of **FlexVF**.

```
 1: strategy = FAIR | ACTIVITY_BASED
 2: while 1 do
 3:     for vm in list_VM do
 4:         extract_statistics(vm)
 5:     end for
 6:     selected_VMs = select_VMs(list_VM, strategy, N)
 7:     for vm in list_VM − selected_VMs do
 8:         unplug_VF(vm)
 9:     end for
10:     for vm in selected_VMs do
11:         plug_VF(vm)
12:     end for
13:     sleep(T)
14: end while
```

4.4. *FlexVF allocation algorithm*

Let us consider a server with *N* available VFs. When VMs are started on the server, as long as there are sufficient VFs for each VM ($num_{VM} \leq N$), all VMs will be assigned an idle PV interface and an active VF. Once $num_{VM} > N$, any new VM will be initially associated with a PV interface. After each scheduling interval *T* (configurable by the system administrator), the VF allocator executes the VF allocation algorithm 1. The VF allocator first extracts the network utilization statistics of each VM (line) 4, then uses this information along with the selected VF allocation strategy to decide which VMs will receive a VF (line 6). Finally, the VF allocator performs the desired hotplugging based on the aforementioned decision (lines 8 and 11).

5. Implementation

This section presents the implementation details of a *FlexVF*. We implemented **FlexVF** on Ubuntu 18.04 with KVM/QEMU version 5.1.0-git on Linux 4.1 using Libvirt as the VM control interface. For our prototype, we opted for an implementation based mostly in user-space. Such an implementation has several advantages: Firstly, it takes advantage of existing libraries (e.g, shared memory tools that can be used in a VM), and also network statistics available in the Linux kernel, thus minimizing the development effort. Secondly, it allows easy reuse of our system across different virtualization technologies, for example the Xen hypervisor [15]. Finally, it facilitates the installation of our system by system administrators, as we don't require an upgrade of the current hypervisor before using our system, and it can also be easily integrated with standard cloud computing platform such as OpenStack [11].

5.1. *FlexVF in the VM*

Our synthetic networking feature in **FlexVF** is based on the network teaming driver, and the network activity hook extracts its information from the */sys/class/net* directory, both of which are existing features in the Linux kernel. For the data exchange between **FlexVF** collector and VF allocator, we rely on the KVM-based zero-copy memory sharing mechanism as described in [16] to let the host map a memory buffer allocated in the guest environment. Upon VF assignment from the VF allocator, the **FlexVF** collector is responsible for reacting to the hotplug/unplug event initiated by the VF allocator. Each time the VF allocator plugs/unplugs a VF to a VM, it triggers a corresponding virtual interrupt to the VM, which is then handled by the **FlexVF** collector running inside. The functionality of the **FlexVF** allocator can be integrated into existing guest services provided by most current hypervisors.

5.2. *FlexVF in the hypervisor*

During hot-plugging and unplugging of network interfaces, the task of the VF allocator is twofold: firstly, it uses the Libvirt API to reconfigure the VM's network interfaces to match its VM selection; and secondly, it notifies the VM of this change through virtual interrupts so that the collector can take the corresponding action. To ensure that our configuration correctly reflects the current network usage, we chose an interval of 300 seconds between each VM ranking. This interval was chosen so as to collect sufficient networking information, while maintaining a quick response to changes in network activity throughout the datacenter.

6. Evaluations

In this section, we aim to answer the following questions:

- What is the overhead of **FlexVF**, including the costs of hot-plugging and unplugging of network interface, as well as the CPU and memory costs incurred by **FlexVF** components?

Hardware	**Description**
CPU	2x Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz (8 cores/16 threads each)
RAM	128 GB
NIC	Intel Corporation 82599ES 10-Gigabit
Software	**Description**
Host OS	Ubuntu 18.04, Linux 4.15
Hypervisor	KVM with QEMU 5.1.0
VM	**Description**
Size	4 vCPU, 4GB of RAM
OS	Ubuntu 18.04, Linux 4.15
NIC	PV: virtio-net-pci; VF: 82599ES

Table 1: Hardware and software setup

- Is packet rate an appropriate VM ranking criterion for activity-based allocation?
- What is the impact of **FlexVF** on application performance?

6.1. Experimental setup

Hardware and software. Our hardware and software setup are described in Table 1.

Benchmarks. We used well-known micro- and macro-benchmarks to evaluate the performance of **FlexVF**, using network latency as the main evaluation metric. The list of benchmarks used in our evaluations is as follows:

- *TailBench. [9]* A benchmark suite for latency-critical applications, it provides request latency figures for multiple types of workloads including search, databases and machine learning.
- *Sysbench and MySQL.* We tested **FlexVF** on a system running the MySQL relational database, using Sysbench as the load generator software.
- *Sockperf.* A socket-based network benchmarking utility, it provides network latency figures under various background loads at predefined packet rates.

6.2. FlexVF overhead

In this section, we estimate the overhead of **FlexVF** regarding the hotplugging and unplugging of network interfaces, as well as the computing resources used by **FlexVF** components. In general, the hot-plugging/hot-unplugging process happens in two steps, one in the initial hypervisor to configure the relevant device, and another inside the guest VM itself to start up the necessary device driver. From our observations, most of the device hotplugging steps on a VM are handled by the VM itself and not the hypervisor, and therefore should not affect the operation of other VMs. Table 2 presents the time required for each operation on 1 VM. While each hotplug and unplug operation takes up to 640 milliseconds, the network teaming inside each guest maintains network connectivity during the transition thanks to the PV interface remaining active throughout this process.

We additionally studied the CPU and memory costs of **FlexVF** components. Profiling of the CPU activities shows that the CPU usage from the **FlexVF** daemon remains under 1% both during normal operation and network reconfiguration, showing **FlexVF**'s efficient use of CPU resources. The **FlexVF** daemon additionally consumes up to 10 MB of memory per guest for the storage and processing of VM network statistics. This figure is sufficiently small so as not to significantly impact the memory capacity of the host.

	Hot-plugging	Hot-unplugging
Time (ms)	640	56

Table 2: Time taken by each network hotplug operation.

6.3. VM selection

As stated in Section 4.2, with an activity-based allocation strategy, **FlexVF** creates a ranking of VMs based on their packet rates, while assuming a linear relationship between the packet rate and potential network performance improvements from SR-IOV. To verify this assumption, we deployed 1 VM running Sockperf to measure network performance under multiple network loads as characterized by the background packet rate traversing in the VM's network interfaces. We collected the resulting median packet latency and calculated a *SR-IOV performance boost percentage* for each network load using the following equation:

$$Boost\% = \frac{lat_{PV} - lat_{VF}}{lat_{PV}} \times 100 \qquad (1)$$

where lat_{PV} and lat_{VF} are network latencies under PV networking and SR-IOV respectively. Figure 5 shows the results obtained under various packet sizes: 16, 64, 256, 1024 and 1472 bytes. We observe that $Boost\%$ increases with the background packet rate, yet is largely unaffected by the packet size used in our experiment. This is explained by our benchmark loads not fully saturating the 10 GbE network used by our machines. Combined with the evaluation of TailBench as shown in Section 3 we can conclude that improvements brought by SR-IOV depends mostly on each VM's network packet rate. Therefore, a packet rate-based ranking is appropriate in this case.

6.4. Macro-benchmark evaluations

MySQL. In this section, we investigate the impact of **FlexVF** on application latencies using the MySQL relational database. We deployed MySQL on the VM under test, used Sysbench to generate a OLTP mixed read/write workload, then measured the resulting request latency. After approximately 15 seconds of benchmarking, we triggered the network switchover using **FlexVF** and observed the change in database performance. Figure 6 shows the resulting request latency as recorded by Sysbench. We observe that with a PV interface, the request latency fluctuates between 7 to 10 msec throughout the benchmark; as soon as **FlexVF** assigns a VF to the server VM, this latency drops to a stable 2 msec with little to no fluctuation, representing a 75% latency reduction. Additionally, we observe no increase in

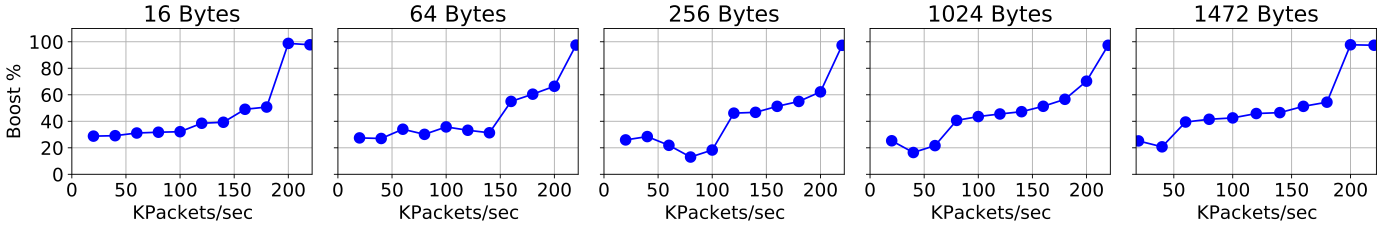


Figure 5: Relation between *Boost%* and background packet rate

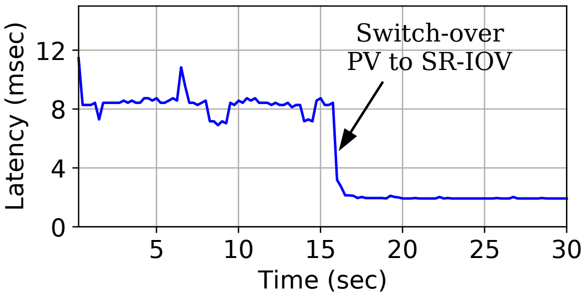


Figure 6: MySQL performance under **FlexVF**

latency during the interface switchover, suggesting that Linux's network teaming mechanism does not affect network performance even during network reconfiguration by **FlexVF**.

Multiple VM evaluation. We aim to evaluate both allocation strategies proposed in Section 4.3. We started 100 VMs each allocated with 1 vCPU and 1 GB of RAM on a physical host, and launched Sockperf on each VM to simulate network loads. These 100 VMs were divided into 4 groups, each sending 1472-byte packets under varying rates: 20 000, 40 000, 80 000 and 100 000 packets per second for groups 1, 2, 3 and 4 respectively. We evaluated network performance under 4 different scenarios: (1) all VMs use PV interfaces; (2) each available VF is allocated to a random VM; (3) using **FlexVF** fair allocation; and (4) using **FlexVF** activity-based allocation to VMs.

Figure 7 presents a boxplot of the resulting packet latency as reported by Sockperf, with (a), (b), (c) and (d) corresponding to the 4 previously-described scenarios. We observe that in scenario (1) with only PV interfaces, packet latencies are the highest; while in scenario (2), we see a range of latency figures. This is simply due to random VF allocation, resulting in some VMs having a VF and therefore having lower I/O latency, while others with a PV interface having a higher I/O latency. Scenario (3) results are quite similar to (2) because all VMs of each group will have access to VFs (i.e. fairness). In scenario (4), we observe that groups 3 and 4 with the highest packet rates (and therefore benefiting the most from VFs) get prioritized for VF allocation as expected. Only group 2 in this scenario shows a variation in latency figures due to some VMs in this group having access to VFs, while others only have access to PV interfaces. Overall, we conclude that with activity-based allocation, **FlexVF**'s VF allocator ensures that VMs benefiting from SR-IOV the most are provided with a VF accordingly, following our criteria as described in Section 4.3.

To further highlight the impact of the allocation strategy, we kept our 100 VMs all running Sockperf along with another VM running an FTP server hosting a 3 GB file. We executed all these benchmarks in two different scenarios: (1) Sockperf VMs are configured with a packet rate higher than that of the FTP VM; and (2) the other way around, using both allocation strategies (Fair allocation and Activity-based allocation). We then measured the time taken to download the file from the FTP VM; Figure 8 presents our results. We observe that with the fair allocation strategy, the FTP VM provides a stable performance regardless of the activity of neighboring VMs, while the activity-based strategy causes a reduction in download throughput of up to 93% depending on the network traffic of neighboring VMs. In short, the two allocation strategy effectively provides a choice between balanced performance of all VMs and optimization of total system throughput.

7. Discussion

We note that **FlexVF** does not guarantee performance predictability in all cases as is often required in public clouds. The performance of a VM with **FlexVF** depends on the allocation strategy, the number of co-located VMs, and their network activities. This is why **FlexVF** is designed to be used in a context where raw throughput is preferable over performance predictability, e.g in private clouds. In these cases, **FlexVF** guarantees a good use of VFs on a server through its various allocation strategies.

8. Related work

Multiple works have explored the problem of I/O performance on virtual machines. In general, these works can be categorized into the following approaches:

Placement-based approaches. This approach involves calculating the placement of virtual resources using a graph-based topology of the network. There have been numerous works such as [17, 18, 19, 20, 21] using such an approach in a way that optimizes service latency, bandwidth or energy consumption. In particular, Carpio, Jia and Wang [18, 19, 20] focus on the context of network function virtualization, aiming to answer the question of virtual network function (VNF) placement. Similarly, VNF-EQ [22] is a solution that uses genetic algorithms to calculate a placement of VNFs in a network of datacenters

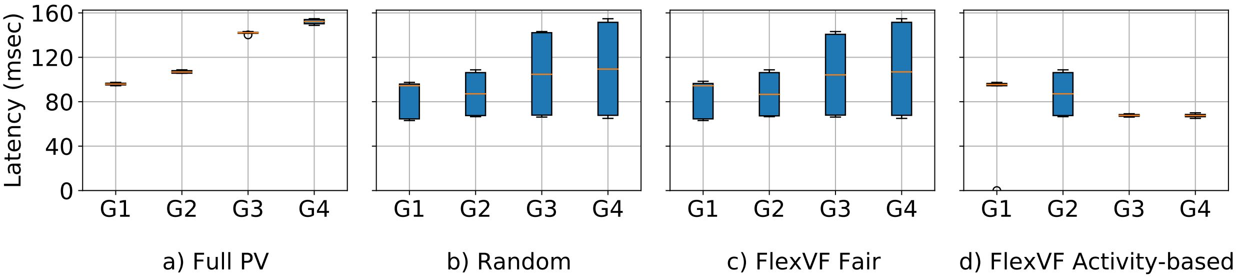


Figure 7: Multiple VMs evaluation with **FlexVF**

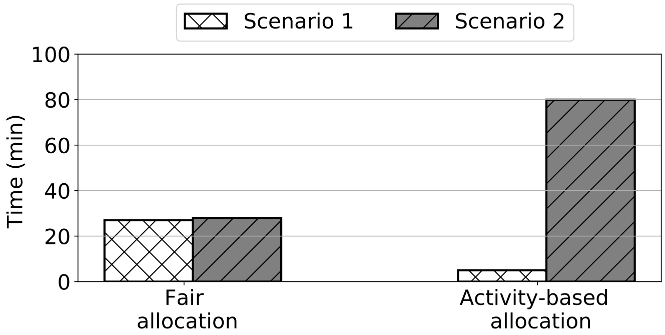


Figure 8: Allocation strategy impact on FTP performance when packet rate is (1) lower than neighbor VMs and (2) higher.

to provide quality-of-service guarantees for network flows and optimize energy efficiency.

Reactive adjustment (autoscaling) approaches. Works in this approach are based on monitoring various system utilization metrics, then adjusting system parameters for load-balancing purposes. For example, Damola and Johnsson [23] propose a framework for monitoring performance criteria such as packet flow and jitter, then using the resulting metrics to migrate VMs onto different servers to balance network loads and avoid performance degradation. Related is Beda and Kadatch [24], whose work monitors the usage of a virtual resource (e.g. virtual disk instance), and adapts its performance by adjusting its I/O rate and throughput as the utilization reaches a certain threshold. Richter et al. [25] describe an anti-DoS mechanism for preventing packet flooding on SR-IOV-enabled guests by counting PCI Express operations and throttling any malicious guest.

SR-IOV-based VNF implementations. Intel's whitepaper [26] introduces the advancements brought by SR-IOV, including its usage with DPDK, methods of utilizing SR-IOV in NFV architecture, and evaluations of SR-IOV compared to virtio and Vhost under various traffic flow patterns. Kourtis et al. [27] describe and evaluate a deep packet inspection implementation based on SR-IOV and DPDK, while comparing its performance to a libpcap-based implementation, and discusses the usage of said solutions with consideration to cloud deployments. Leivadeas et al. [28] describe various VNF topologies derived from multiple VNF chaining mechanisms: no chaining, direct virtual Ethernet bridge in hardware, KVM bridge and VEPA bridge. Hamed et al. [29] analyzes in detail VNF deployment scenarios with single- or multi-feature VNFs with different chaining traffic paths, and evaluates the performance and scalability of multiple-VNF setups under each scenario while taking into account VNF oversubscription.

Other approaches and evaluations. These works investigate general opportunities for increasing I/O performance, for example by improving software-hardware interfaces as used by operating systems. Taguchi et al. [30] introduces a packet aggregation solution that dynamically merges and splits packets in a VNF dataflow to reduce virtual network packet routing overhead, thus increasing throughput while reducing latency. Dong et al. [12] evaluates the characteristics of SR-IOV-based virtual networking, designs an architecture for providing said virtual interfaces to VM guests, and details how SR-IOV can be utilized in a migration context by combining an SR-IOV virtual interface with a paravirtualized interface through use of a bonding driver. The same technique is used by Microsoft Azure to ensure VM connectivity in case the VF is disconnected [8]. Shea et al. [31] provides a detailed evaluation of network performance on virtualized environments, including measurements of throughput, roundtrip time and CPU load with different types of traffic, and suggests a method for optimizing network performance of said environments with changes in the Xen scheduler configuration. Huang and Liu [32, 33] investigate the performance of SR-IOV-enabled 10 GbE Ethernet NICs under various conditions; works such as [32, 34, 35] particularly focus on the interrupt handling by the operating system, and considers optimizations through interrupt throttling, multithreaded processing or intelligent polling to improve network performance on NICs.

Position of our work. To our best knowledge, we are the first work to suggest using an adaptive I/O interface that dynamically allocates VFs for the purpose of optimizing network accesses for virtual machines.

9. Conclusion

Hardware constraints limit the number of virtual functions available on each SR-IOV-enabled virtualization host, raising the question of which VMs should benefit and which VMs should not.

We introduced **FlexVF**, a solution that monitors VMs and dynamically allocates VFs with two strategies: fair allocation or based on their network activity. We assessed **FlexVF** under various workloads, and demonstrated that **FlexVF** takes advantage of SR-IOV to improve network latencies without affecting compatibility with networked applications or causing application overhead.

Acknowledgements

This work is supported by the French National Research Agency (ANR-20-CE25-0005) and the CIMI research program. Some experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr/).

References

[1] R. Shea, J. Liu, Network interface virtualization: challenges and solutions, IEEE Network 26 (5) (2012) 28–34.

[2] Implementing Large Numbers of Virtual Functions with PCI Express SR-IOV, https://www.synopsys.com/designware-ip/technical-bulletin/implementing-large-numbers.html.

[3] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, L. Sarzyniec, Adding virtualization capabilities to the Grid'5000 testbed, in: I. I. Ivanov, M. van Sinderen, F. Leymann, T. Shan (Eds.), Cloud Computing and Services Science, Vol. 367 of Communications in Computer and Information Science, Springer International Publishing, 2013, pp. 3–20. doi:10.1007/978-3-319-04519-1_1.

[4] R. Ricci, E. Eide, C. Team, Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications, ; login:: the magazine of USENIX & SAGE 39 (6) (2014) 36–38.

[5] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, R. Bianchini, Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms, in: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 153–167. doi:10.1145/3132747.3132772.
URL https://doi.org/10.1145/3132747.3132772

[6] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, D.-M. Popa, Firecracker: Lightweight virtualization for serverless applications, in: 17th {usenix} symposium on networked systems design and implementation ({nsdi} 20), 2020, pp. 419–434.

[7] Enhanced networking on Linux, https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html.

[8] Create a Linux virtual machine with Accelerated Networking using Azure CLI, https://docs.microsoft.com/en-us/azure/virtual-network/create-vm-accelerated-networking-cli.

[9] H. Kasture, D. Sanchez, Tailbench: a benchmark suite and evaluation methodology for latency-critical applications, in: 2016 IEEE International Symposium on Workload Characterization (IISWC), 2016, pp. 1–10.

[10] Intel Ethernet Converged Network Adapter X550, https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-x550-brief.pdf.

[11] A. Shrivastwa, S. Sarat, K. Jackson, C. Bunch, E. Sigler, T. Campbell, OpenStack: Building a Cloud Environment, Packt Publishing, 2016.

[12] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, H. Guan, High performance network virtualization with SR-IOV, Journal of Parallel and Distributed Computing 72 (11) (2012) 1471–1480.

[13] Configuring QoS Features with Intel Flexible Port Partitioning, https://www.intel.in/content/dam/www/public/us/en/documents/white-papers/config-qos-with-flexible-port-partitioning.pdf.

[14] M. Chowdhury, I. Stoica, Coflow: A networking abstraction for cluster applications, in: Proceedings of the 11th ACM Workshop on Hot Topics in Networks, 2012, pp. 31–36.

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the Art of Virtualization, SIGOPS Oper. Syst. Rev. 37 (5) (2003) 164–177. doi:10.1145/1165389.945462.
URL https://doi.org/10.1145/1165389.945462

[16] C. Pinto, B. Reynal, N. Nikolaev, D. Raho, A zero-copy shared memory framework for host-guest data sharing in KVM, in: 2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), 2016, pp. 603–610.

[17] J. Dong, H. Wang, X. Jin, Y. Li, P. Zhang, S. Cheng, Virtual machine placement for improving energy efficiency and network performance in IaaS cloud, in: 2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops, IEEE, 2013, pp. 238–243.

[18] F. Carpio, S. Dhahri, A. Jukan, VNF placement with replication for Loac balancing in NFV networks, in: 2017 IEEE International Conference on Communications (ICC), IEEE, 2017, pp. 1–6.

[19] M. Jia, W. Liang, Z. Xu, QoS-aware task offloading in distributed cloudlets with virtual network function services, in: Proceedings of the 20th ACM International Conference on Modelling, Analysis and Simulation of Wireless and Mobile Systems, 2017, pp. 109–116.

[20] F. Wang, R. Ling, J. Zhu, D. Li, Bandwidth guaranteed virtual network function placement and scaling in datacenter networks, in: 2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC), IEEE, 2015, pp. 1–8.

[21] L. Yu, Z. Cai, Dynamic scaling of virtual clusters with bandwidth guarantee in cloud datacenters, in: IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE, 2016, pp. 1–9.

[22] S. Kim, S. Park, Y. Kim, S. Kim, K. Lee, VNF-EQ: dynamic placement of virtual network functions for energy efficiency and QoS guarantee in NFV, Cluster Computing 20 (3) (2017) 2107–2117.

[23] A. Damola, A. Johnsson, Network performance monitor for virtual machines, uS Patent 9,063,769 (Jun. 23 2015).

[24] J. S. Beda III, A. Kadatch, Adjustable virtual network performance, uS Patent 8,276,140 (Sep. 25 2012).

[25] A. Richter, C. Herber, S. Wallentowitz, T. Wild, A. Herkersdorf, A hardware/software approach for mitigating performance interference effects in virtualized environments using SR-IOV, in: 2015 IEEE 8th International Conference on Cloud Computing, IEEE, 2015, pp. 950–957.

[26] SR-IOV for NFV Solutions - Practical Considerations and Thoughts, https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/sr-iov-nfv-tech-brief.pdf.

[27] M.-A. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, F. Liberal, Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration, in: 2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN), IEEE, 2015, pp. 74–78.

[28] A. Leivadeas, M. Falkner, N. Pitaev, Analyzing service chaining of virtualized network functions with SR-IOV, in: 2020 IEEE 21st International Conference on High Performance Switching and Routing (HPSR), IEEE, 2020, pp. 1–6.

[29] A. Ben Hamed, A. Leivadeas, M. Falkner, N. Pitaev, VNF chaining performance characterization under multi-feature and oversubscription using SR-IOV, in: Informatics, Vol. 7, Multidisciplinary Digital Publishing Institute, 2020, p. 33.

[30] Y. Taguchi, R. Kawashima, H. Nakayama, T. Hayashi, H. Matsuo, Pa-flow: Gradual packet aggregation at virtual network i/o for efficient service chaining, in: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2017, pp. 335–340.

[31] R. Shea, F. Wang, H. Wang, J. Liu, A deep investigation into network performance in virtual machine based cloud environments, in: IEEE INFOCOM 2014-IEEE Conference on Computer Communications, IEEE, 2014, pp. 1285–1293.

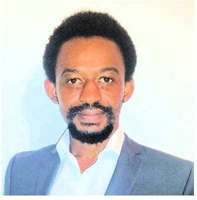
[32] Z. Huang, R. Ma, J. Li, Z. Chang, H. Guan, Adaptive and scalable optimizations for high performance SR-IOV, in: 2012 IEEE International Conference on Cluster Computing, IEEE, 2012, pp. 459–467.

[33] J. Liu, Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support, in: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), IEEE, 2010,

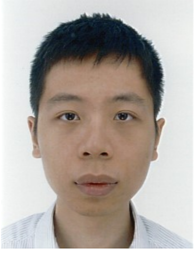
pp. 1–12.

[34] H. Guan, Y. Dong, K. Tian, J. Li, SR-IOV based network interrupt-free virtualization with event based polling, IEEE Journal on Selected Areas in Communications 31 (12) (2013) 2596–2609.

[35] J. Li, S. Xue, W. Zhang, Z. Qi, et al., When I/O interrupt becomes system bottleneck: Efficiency and scalability enhancement for SR-IOV network virtualization, IEEE Transactions on Cloud Computing (2017).

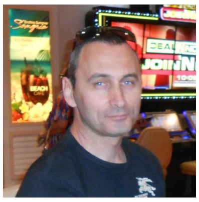


Brice Ekane received his M.S. in University of Yaounde Cameroon in 2011. Since September 2018 he is carrying out a PhD at IRIT lab, Toulouse France. He is a member of SEPIA research group. His main research interests are in Virtualization, Cloud Computing, and Operating System.



Tu Dinh Ngoc received his M.S. in UST Hanoi, Vietnam in 2017. Since September 2020 he is carrying out a PhD at IRIT lab, Toulouse France. He is a member of SEPIA research group. His main research interests are in Virtualization, Cloud Computing, and Operating System.

Boris Teabe received his PhD in computer science in 2017, at the IRIT lab, Toulouse, France. Since September 2019 he is Lecturer and Research Assistant at Polytechnic National Insitute of Toulouse, France. He is a member of SEPIA research group. His main research interests are in Scheduling in virtualization, Cloud Computing and Operating System.



Daniel Hagimont is a Professor at Polytechnic National Institute of Toulouse, France and a member of the IRIT laboratory, where he leads a group working on operating systems, distributed systems and middleware. He received a PhD from Polytechnic National Institute of Grenoble, France in 1993. After a postdoctorate at the University of British Columbia, Vancouver, Canada in 1994, he joined INRIA Grenoble in 1995. He took his position of Professor in Toulouse in 2005.

Noel De Palma received his PhD in computer science from the Grenoble Institute of Technology in 2001. From 2002, he was Associate Professor in computer science at University of Grenoble. Since 2010 he is professor at University of Grenoble Alpes.