HyperNF: Building a High Performance, High Utilization and Fair NFV Platform

Kenichi Yasukata
NEC Laboratories Europe
kenichi.yasukata@neclab.eu

Felipe Huici
NEC Laboratories Europe
felipe.huici@neclab.eu

Vincenzo Maffione
Università di Pisa
vincenzo.maffione@ing.unipi.it

Giuseppe Lettieri
Università di Pisa
g.lettieri@iet.unipi.it

Michio Honda
NEC Laboratories Europe
michio.honda@neclab.eu

ABSTRACT

Network Function Virtualization has been touted as the silver bullet for tackling a number of operator problems, including vendor lock-in, fast deployment of new functionality, converged management, and lower expenditure since packet processing runs on inexpensive commodity servers. The reality, however, is that, in practice, it has proved hard to achieve the stable, predictable performance provided by hardware middleboxes, and so operators have essentially resorted to throwing money at the problem, deploying highly underutilized servers (e.g., one NF per CPU core) in order to guarantee high performance during peak periods and meet SLAs.

In this work we introduce HyperNF, a high performance NFV framework aimed at maximizing server performance when concurrently running large numbers of NFs. To achieve this, HyperNF implements hypercall-based virtual I/O, placing packet forwarding logic inside the hypervisor to significantly reduce I/O synchronization overheads. HyperNF improves throughput by 10%-73% depending on the NF, is able to closely match resource allocation specifications (with deviations of only 3.5%), and to efficiently cope with changing traffic loads.

CCS CONCEPTS

• **Networks** → **Middle boxes / network appliances**; *Routers*; • **Software and its engineering** → *Communications management*;

KEYWORDS

Hypervisor, Middlebox, NFV

ACM Reference Format:
Kenichi Yasukata, Felipe Huici, Vincenzo Maffione, Giuseppe Lettieri, and Michio Honda. 2017. HyperNF: Building a High Performance, High Utilization and Fair NFV Platform. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 13 pages.
https://doi.org/10.1145/3127479.3127489

1 INTRODUCTION

The promise of Network Function Virtualization (NFV) was a shift from expensive, difficult to manage and modify hardware middleboxes to a world of software-based packet processing running on general purpose OSes and inexpensive off-the-shelf x86 servers; this change came with several benefits for operators, including converged management, no vendor lock-in, functionality that was much easier to upgrade accompanied by much shorter deployment cycles, and reduced investment and operational costs.

Despite its advantages, the reality is that NFV platforms have unpredictable performance: the sharing of common resources such as last-level caches and CPU cores means that while the system may run reliably when a single or a few network functions (NFs) are running, under high load the platform may experience reduced throughput and high packet loss. Worse, its performance may be highly dependent on the traffic matrix and the actual NFs running on the system: for instance, a Deep Packet Inspection (DPI) function will consume many cycles on a packet with a suspicious payload, but will be cheap for "normal" packets.

Because of this, operators, for whom the imperative is to provide predictable performance and comply with Service Level Agreements (SLAs), have to opt for over-provisioning NFV platforms, often by assigning an entire CPU core to a single NF. This peak-level provisioning is one of the reasons hardware middleboxes are expensive, and results in severe under-utilization when load drops. This drop is significant: network traffic has a large gap between peak load to average load (five-fold), and between peak load and minimum load (ten-fold), and this gap is steadily increasing [6]. In addition, load peaks are rare (e.g., 1 hour on any given day): if we provision for peak performance, we will need five times more resources, on average, than a system that adapts to load.

On the research side, the work has largely focused on achieving high throughput. One of the most common techniques has been to leverage kernel bypass packet I/O frameworks. NetVM [17], for instance, uses DPDK [18] to obtain high performance, but suffers from rather poor CPU utilization due to DPDK's reliance on busy polling. As another point in that space, ClickOS [24] opts for the netmap framework [32] and unikernels to obtain high throughput, but uses pinning and CPU cores dedicated to I/O (the latter is also true of NetVM) for high performance, which hurts utilization and adaptability to changing traffic loads.

What is missing is an NFV platform that is able to provide both high throughput and high utilization, all the while allowing for accurate resource allocation and the adaptability to changing traffic loads that would make NFV deployments a better value proposition. In this work we introduce HyperNF, a high performance NFV platform implemented on the Xen [3] hypervisor that meets these requirements. Our contributions are:

- An investigation of the root causes behind the difficulty of simultaneously obtaining high throughput, utilization and adaptability to changing loads. We find that using a "split" model, with cores dedicated to I/O is wasteful and leads to unfairness; a "merged" model (VM and I/O processing happen on the same core) is better but has high synchronization overheads.
- The introduction of *hypervisor-based virtual I/O*, a mechanism whereby I/O is carried out within the context of a VM, reducing I/O synchronization overheads while improving fairness.
- Efficient Service Function Chaining (SFC), with higher throughput than baseline setups and delay of as little as 2 milliseconds for a chain of 50 NFs.
- A thorough performance evaluation of the system. On an inexpensive x86 server, HyperNF improves throughput by 10%-73% depending on the NF, is able to closely match specified resource allocations (with deviations of only 3.5%).

To show the generality of HyperNF's architecture, we further implement it on Linux QEMU/KVM [4] by leveraging the ptnetmap software [9, 23]. We release the Xen-based version of HyperNF as open source, downloadable from https://github.com/cnplab/HyperNF.

2 REQUIREMENTS AND PROBLEM SPACE

In order to improve the current status quo of over-provisioned NFV platform deployment, we would like our system to comply with a number of performance requirements [1]. In particular, in order to reduce costs and to be able to provide accurate resource allocation and meet SLAs we would need:

- **High Utilization**: Ensuring that CPU cycles are neither wasted (e.g., by busy polling when no packets are available) nor idle (e.g., by dedicating cores to I/O when they could be used by VMs).
- **Adaptability**: The platform should be able to dynamically assign idle resources in order to cope with peaks in traffic.
- **High Throughput**: To provide not only high per-NF throughput but high cumulative throughput for the platform as a whole.
- **Accurate Resource Allocation**: VMs and the packet processing they need to carry out should comply with the amount of resources allotted by the platform's operator (e.g., NF1 should use 30% of a CPU core, NF2 50%, etc.) with only minor deviations.

Next, we take a look at each of these in turn, investigating the problem space and especially the issues that prevent current virtual networking techniques to achieve these requirements.

[1]Other functional requirements such as ease of management and fault tolerance are outside the scope of this paper.

2.1 High Utilization and Adaptability

Given a set of CPU cores, a number of VM threads, and a number of I/O threads, there are a number of ways that we can assign the threads to the available cores (see Figure 1). One of the most common models is what we term "split" (Figure 1a), whereby some cores are solely dedicated to I/O threads and others are given to VM threads [2]. The idea here is to maximize throughput by parallelizing I/O and VM processing: while a VM/NF is busy processing packets (e.g., modifying a header for a NAT), a separate core running an I/O thread is in charge of sending and receiving other packets. Unfortunately, dedicating cores to particular tasks reduces utilization when those tasks are idle (e.g., an expensive set of regexs in a DPI need to be applied by an overloaded CPU core while an I/O core is idle because there are no incoming/outgoing packets to service).

This is essentially a static allocation of resources, which is inefficient and exacerbated by changing traffic conditions and the type of NFs currently running on the system. To quantify this effect, we conduct a simple experiment on a 14-core server. First, we assign a single CPU core to a VM running `netmap-ipfw`, a Netmap-based firewall [33], and another core to all I/O threads in dom0 (in our case this means all I/O threads of the VALE [15, 35] backend software switch). This configuration is identical to experiments in the ClickOS [24] paper. We then limit the number of cycles available to the VM and I/O threads by using Xen's credit scheduler's `cap` command and Linux's `cgroup`, respectively, in order to show what the effects of different static assignments are on performance.

With this in place, we then generate traffic using netmap's `pkt-gen` running on dom0, forward the traffic through the firewall VM, and count the resulting rate in another `pkt-gen` running also on dom0. We run this experiment for 64-byte and 1024-byte packets and a 10 and 50-rule firewall where packets have to traverse all rules. Figure 2 shows the results for various static assignments, all of which add up to 100% so that they can be directly compared. What's apparent from the graph is that no single assignment is ideal for all setups; in other words, static assignments, as is the norm with the split model, are practically guaranteed to under-utilize system resources and lead to sub-optimal performance.

An alternative to this is the "merge" model (see Figure 1b), in which a VM and its corresponding I/O thread runs on the same CPU core. In principle, in terms of high utilization, the merge model is better than the split one, since, assuming a work-conserving scheduler, idle CPU cycles can be used by the VM or the I/O thread as needed (i.e., there is no longer a static separation of resources between VM and I/O). Consequently, in order to meet the high utilization requirement we opt for the merge model. In addition, because this model allows the system to share CPU cycles between I/O and VM/packet processing as needed, it can much better adapt to changing traffic conditions, as we will show later on. Having said that, this model comes with its own problems, which we explore and quantify next.

2.2 High Throughput

The drawback of the merge model are the high overheads related to the fact that VMs have to be frequently preempted in order for the I/O thread to execute. Such switching includes an expensive

[2]This is the model used by systems like ClickOS and NetVM.

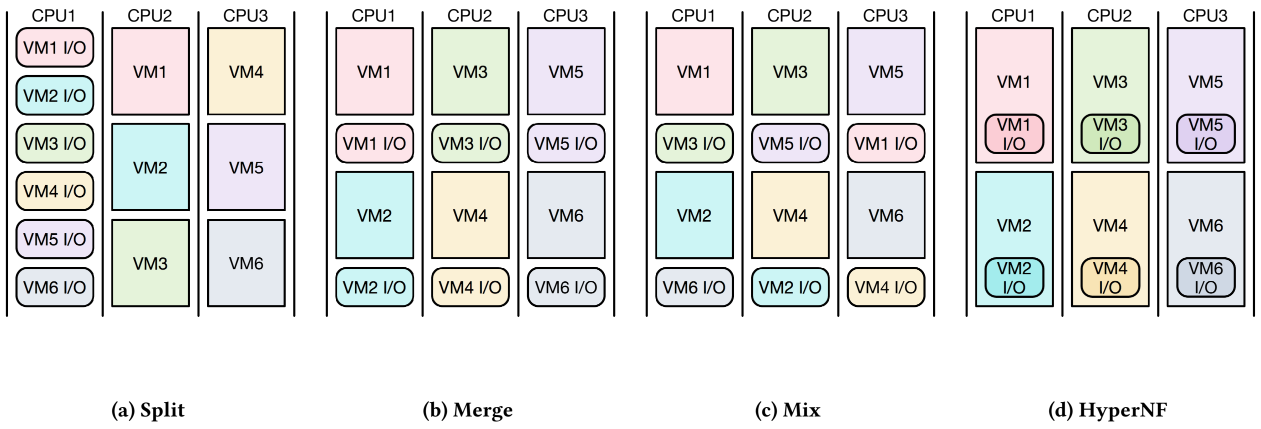


Figure 1: Different models for assigning CPU cores to VMs and I/O processing. Split uses different CPU cores for I/O and VMs, merge runs a VM and its corresponding I/O on the same core, and mix lets the scheduler assign resources.

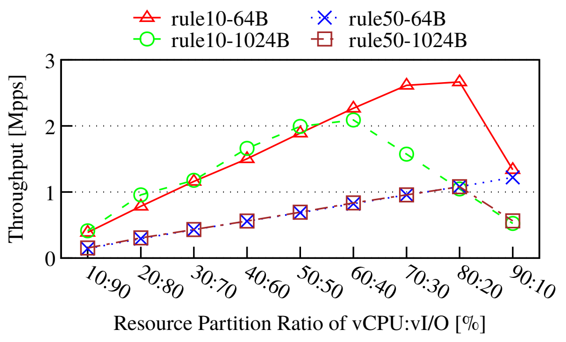


Figure 2: Throughput when forwarding traffic through a firewall VM for different static CPU assignments to the VM versus the I/O threads, different packet sizes, and different number of firewall rules. No assignment yields the highest throughput for all cases.

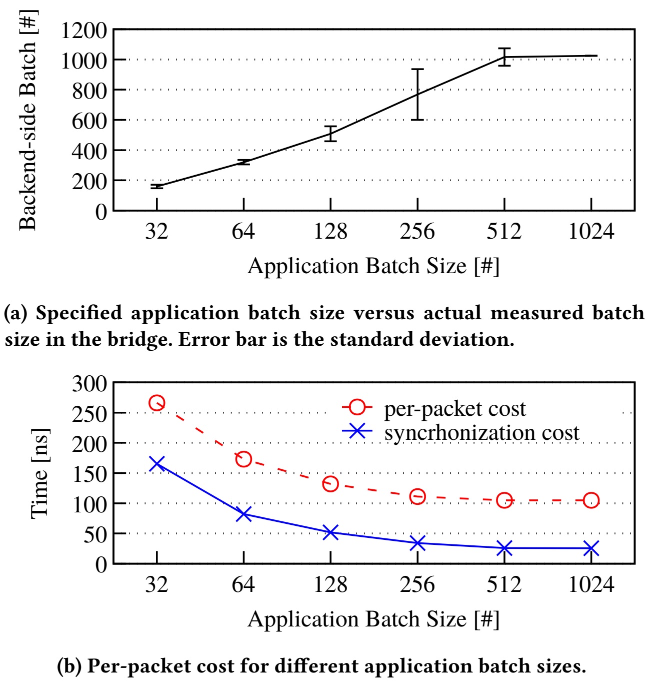


Figure 3: Synchronization (measured as per-packet) costs in the merge model for different batch sizes.

VMEXIT instruction, along with thread hand-off and synchronization mechanisms between the VM and the I/O thread (e.g., for the VM to notify the I/O thread that it has placed packets in a ring buffer).

To measure these overheads, we implement a simple event notification split driver in Xen which creates a shared memory region between a Linux guest (the VM) and a kernel thread in the Linux-based driver domain (the I/O thread). In the test, the VM begins by setting a variable in the shared memory region to 1 and kicking an event to wake up the kernel thread. When Xen schedules the driver domain and then Linux schedules the kernel thread, the thread sets the variable to 0 and goes back to sleep. Finally, the VM wakes up and checks that the value is 0, and, if it is, sets the variable back to 1. This setup mimics the synchronization procedure of the merge model while reducing any extraneous costs to a minimum, thus measuring context switch costs.

Using the same x86 server as before, we carry out the procedure above one million times and measure the results. For this test, we use the credit2 scheduler and disable its rate limit feature. On average, it takes 26.306 microseconds to complete a single synchronization

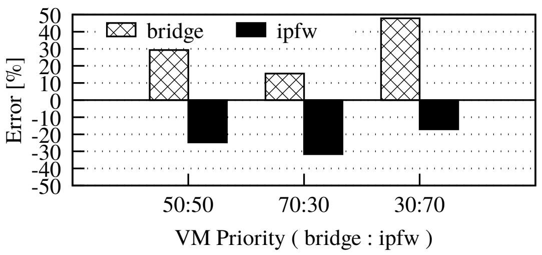


Figure 4: Error when comparing expected throughput from two VMs (as set by Xen's prioritization mechanism) versus actual forwarded throughput.

"round-trip" (for reference, the time budget for processing a single minimum-sized packet at 10Gb/s is roughly 67 nanoseconds).

To shed light on the impact of this synchronization cost on actual packet processing, we use a VM running netmap's `pkt-gen` to send 64B packets out as quickly as possible to a `pkt-gen` receiver on dom0 and connected via a VALE switch that also resides in dom0. Since delivery of notification takes time and the application keeps updating packet buffers, upon each notification received, the backend usually forwards more packets than the application batch size, up to 1024 which is our ring buffer size. We plot the correlation between the application and backend-side batch sizes in Figure 3a.

In Figure 3b, the solid line shows the per-packet synchronization costs which divide 26.306 microseconds of that synchronization cost by the backend-side batch sizes. Depending on the application batch size, it ranges between 25.68–165.55 nanoseconds. These synchronization costs are high, comprising 24.45–62.19 % of the entire per-packet processing time shown as the dashed line in the figure.

2.3 Accurate Resource Allocation

Schedulers in virtualization platforms have interfaces to control how many resources, in terms of percentage of a physical CPU, each VM is allowed to use. This is a useful tool that lets operators control per-NF resources (and thus throughput) in order to meet performance targets. But how accurately does this priority percentage apply to a VM that runs an NF? We expect that a VM that achieves 10 Mpps with 100% of CPU allocated achieves 5 Mpps with 50% of CPU allocated, as NF processing on fast packet I/O frameworks is usually CPU intensive.

To see what happens in reality, we run two VMs and their I/O threads on a single CPU core; one runs a bridge application and the other runs the netmap-ipfw application with 50 filtering rules. We set their priorities to 50%/50%, 30%/70% and 70%/30 respectively. For each of the VMs we run a sender and receiver `pkt-gen` process in dom0.

We then measure how much the forwarded traffic deviates from the split we specified and calculate: $error = (goodput - expectedthroughput)/expectedthroughput \times 100$. The results in Figure 4 show rather large errors ranging anywhere from 15% to 47%, meaning that the prioritization mechanism is not able to provide prioritized performance isolation.

Indirect Scheduling Problem: The reason for this is Xen-specific: traffic for all VMs goes through the same driver domain, and so the hypervisor has no way to separately account for I/O performed on behalf of the different VMs [8]. While it would be in principle possible to create a separate, dedicated driver domain for each VM, solving the accounting problem, this would add twice the number of VMs to the system, resulting in additional overheads. This indirect scheduling problem has also been pointed out in [11]. A solution for this and the other issues presented in this section is provided in the next section.

3 DESIGN AND IMPLEMENTATION

The goal of HyperNF, our high performance NFV platform, is to meet the four requirements previously outlined. As a starting point, the analysis in the previous section would seem to suggest the use of a merge model, since its flexible use of (idle) CPU cycles allows us to meet the high utilization and adaptability requirements. However, the merge model does not scale to high packet rates because of high context switching costs.

Is it then possible to use the merge model but significantly reduce these overheads? One possibility would be to offload the I/O thread operations to the hypervisor, performing the entire packet forwarding in the context of hypercall. Before designing a system around this model, we would like to understand how much it would help in reducing the synchronization overheads in the merge model. To do so, we measure hypercall overhead by implementing a NOP hypercall in Xen. On the same server as before, we execute the hypercall one million times and measure the time it takes to complete. On average, the hypercall costs about 507 nanoseconds, 51 times cheaper than the VM context switch cost of 26 microseconds reported before. This measurement shows that hypercall-based I/O has the potential to eliminate a large portion of the synchronization overheads present in the merge model.

3.1 HyperNF Architecture

Through the analysis in the previous sections we arrive at the following three design principles:

(1) There should be no CPU cores dedicated solely to virtual I/O. This principle suggests the use of the merge model, and allows us to meet the high utilization and adaptability requirements.
(2) Virtual I/O operations should be accounted to the VM they are being carried out for. This also points to a merge model and ensures that the NFV platform can provide accurate resource allocation.
(3) VM context switches between VMs/packet processing and virtual I/O should be avoided. Without this, it is difficult to meet the high throughput requirement.

These principles point us to HyperNF's main architectural design: virtual I/O, and in particular packet forwarding, should be done within the context of a hypercall; we call this *hypervisor-based virtual I/O*.

A naive approach would be to attempt to embed an entire software switch such as VALE or Open vSwitch [29, 41] in the hypervisor, but this would not only unduly increase the hypervisor's

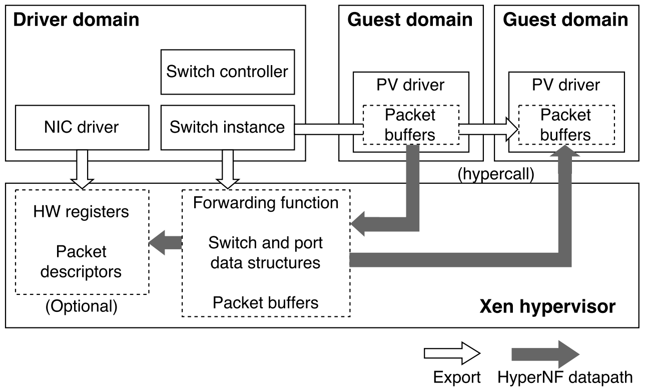


Figure 5: HyperNF architecture. Packet switching and forwarding are done through a hypercall by exporting the backend software switch's data path to the hypervisor and, optionally, the fast path of NIC drivers. Tx/Rx operations issued from a VM are carried by the hypercall and do not require VM switches.

Trusted Computing Base (TCB), but would also constitute a significant porting effort, since part of the code would be OS-dependent.

Instead, HyperNF runs the VALE software switch [3] in a privileged VM (in our implementation on dom0) but *exports* to the hypervisor all structures and state needed to implement the switch's packet forwarding code (see Figure 5). All of these objects are device-independent structures, and the packet forwarding code is even OS-independent.

In all, the changes to the Xen hypervisor are kept small, and are easy to maintain throughout version updates: most of the LoC and complex parts of VALE are left in dom0. This means that the porting effort is reasonable, the amount of code is fairly minimal, and the code itself is relatively straightforward. To be more specific, the netmap code base consists of 17,655 LoC not including NIC driver code, versus 1,173 LoC for the in-hypervisor packet forwarding code (i.e., only about 6.6% of the code base). In addition, the HyperNF architecture allows us to maintain the switch's management interfaces unmodified, so it does not break support from automation frameworks.

In sum, HyperNF works as a *fastpath* to the backend switch, executing all the jobs typically done by I/O threads in a hypercall. This has multiple advantages. First, the CPU time for virtual I/O is taken directly from the time allocated to the vCPU, i.e., vCPU and virtual I/O are a single resource entity as far as the hypervisor scheduler is concerned, allowing for sharing of cycles between packet and I/O processing, for better accounting of I/O operations, and for lower overheads since there is no longer expensive VM scheduling when doing I/O. Second, since VM and I/O are now merged, the hypervisor's CPU load balancing mechanism performs well and vCPU and virtual I/O are always migrated together to the same CPU core. Third, the platform can now guarantee that the batch size set by the packet sender is respected since packet forwarding is executed within the hypercall, thus offering not only high but predictable performance.

[3]While our implementation mainly focuses on a VALE software switch, HyperNF's architecture can be applied to other software switches such as Open vSwitch.

In the rest of this section we give a more detailed description of how HyperNF sets up the switch and its structures, how packet I/O is done, and what its paravirtualized and NIC drivers look like.

3.2 HyperNF Setup

After the driver domain initializes a VALE switch instance and attaches NICs and virtual ports to it as usual, HyperNF exports internal and shared objects used by the packet forwarding path to the hypervisor. These objects include device-independent structures such as locking status objects and packet buffers, and a switch structure used mainly to obtain a list of ports.

When a VM is created, HyperNF's backend driver (described later) creates a new virtual port through the standard mechanism provided by VALE. After this, HyperNF maps packet buffers, ring indexes and locking status structures of the virtual port into the hypervisor, essentially exporting VALE's forwarding plane. For the actual memory mapping, the driver domain provides the page frame numbers of the objects to the hypervisor. At this point, Xen allocates a new virtual memory region and inserts these page table numbers in the entries of the newly allocated virtual memory region. After this operation, the virtual port and switch-related objects can be accessed from the hypervisor over the mapped virtual memory. Next, HyperNF maps the objects related to a VM's virtual port(s) into the VM's address space by using Xen's grant tables. Once this is all done, a VM can access the netmap rings and buffers into this contiguous shared memory area. Since the layout of these data structures follows the standard netmap API, VMs can access packet buffers and ring objects by offsets.

Additionally, the driver domain has to map switch structures including hash tables and associated port information to the hypervisor's address space so that the hypervisor has access to them when making forwarding decisions.

3.3 Packet I/O

HyperNF implements a new `hyperio` hypercall so that VMs can access the packet switching logic exported to the hypervisor. The hypercall interface allows for specifying either a TX or an RX operation.

For transmitting packets, a VM puts payloads on packet buffers and updates the variables of ring indexes. To start packet switching, the VM invokes the hypercall specifying a TX operation. Through the hypercall, the execution context is immediately switched into the hypervisor and reaches the entry point of the newly implemented hypercall. The hypercall executes the same logic as VALE implemented in the driver domain. Since all required objects are mapped in the hypervisor, the VALE logic in the hypervisor can maintain consistency. After the execution of packet switching, which includes destination lookup and data movement (memory copy), the hypercall returns, and the VM resumes its execution.

For receiving packets, a VM calls the hypercall this time with the RX parameter. In the hypercall, HyperNF updates the indexes of the RX rings' according to the packets that have arrived. After the index update, HyperNF returns to the VM's execution context. The applications running in the VM can then consume the available packets based on the updated ring indexes.

This Tx/Rx behavior closely mimics that of netmap's standard execution model, with the slight difference that we use a hypercall instead of a syscall and that I/O is not executed in the kernel but in the hypervisor.

3.4 Split Driver

Following Xen's split driver model, HyperNF implements frontend and backend drivers as separate Linux kernel modules. The backend driver running in the driver domain (dom0 in our case) creates the switch's virtual ports and exposes those ports' (netmap) buffers. The frontend driver then maps those buffers into the VM's kernel space. Further, the frontend driver provides applications with an interface consisting of `poll( )` and `ioctl( )` syscalls so that they can invoke the `hyperio` hypercall, and allows VMs to run unmodified netmap applications.

3.5 Physical NIC Driver

Just like VALE, HyperNF allows NICs to be directly connected to the switch. In particular, HyperNF supports two modes of operation for NICs:

- **Hypervisor driver**: We port only the Tx/Rx functions of the Intel Linux 40Gb driver (`i40e`) to the hypervisor (about 1,248 LoC versus 48,780 for the whole driver). As with the switch, we keep most of the logic in the driver domain (e.g., the driver can still be configured by `ethtool`) and export any necessary structures such as NIC registers to the hypervisor. Driver initialization is still done by the driver domain (i.e., Linux).
- **Driver domain driver**: The driver resides in the driver domain/dom0 with I/O thread(s). The `hyperio` hypercall notifies dom0 via an event channel after packet forwarding takes place so that dom0 updates the relevant device registers.

The first option has the advantage of eliminating I/O threads and any synchronization costs deriving from them, but has the downside of increasing the hypervisor's TCB and needing a porting effort (even if this is sometimes small) for each individual driver.

The second option has the advantage that it gives the driver domain the opportunity to mediate packets destined to the NIC, for example, for packet scheduling [37]. In addition, it does not require changes to the hypervisor. A downside is inherent overheads due to use of I/O thread. We will present evaluation results from these two execution models in Section 4.

Finally, in both modes, packet reception from a physical NIC is always handled by I/O threads in the driver domain. The number of CPU cores required depends on packet demultiplex logic in addition to link speeds [15].

4 BASELINE EVALUATION

In this section we carry out a thorough evaluation of HyperNF's basic performance. We show high throughput and utilization results when running different individual network functions (NFs), when concurrently running many of them on the same server, and results that show HyperNF's ability to provide performance guarantees.

For all tests we use a server with an Intel Xeon E5-2690 v4 CPU at 2.6 GHz (14 cores, Turbo Boost and Hyper-Threading disabled), 64GB of DDR4 RAM and two Intel XL710 40 Gb NICs.

We use Xen version 4.8 and the credit2 scheduler, unless otherwise stated. We disable the rate limit feature of credit2. The rate limit specification assures a VM can run for a minimum amount of time, and is designed to avoid frequent VM preemption. However, in our use cases, frequent VM preemption is essential for fast packet forwarding especially in the merge model.

For all VMs including dom0, we use Linux 4.6 in PVH mode since this gives better performance than PV mode [40].

For experiments with NICs we use an additional pair of servers, one for packet generation and one as a receiver; both of these have a four-core Intel Xeon E3-1231 v3 CPU at 3.40GHz, 16GB RAM and an Intel XL710 40 Gb NIC. Both of these computers are connected via direct cable to our 14-core HyperNF server.

As a baseline, we use an I/O thread-based model whose architecture is similar to the network backend presented in the ClickOS [24] paper. VMs transmit and receive packets using the paravirtual frontend driver adopting the netmap API, and packets are forwarded by the VALE switch in the driver domain. The same application binaries are run for both the baseline virtual I/O implementation and HyperNF.

We use the VALE switch as a learning bridge which is the default implementation in VALE [15]. We set the bridge batch size of VALE to 100 packets so that the performance scales with multiple senders [15]. Each virtual port has 1024 slots for TX and RX rings respectively. In NF tests, each VM has two virtual ports, and NF applications forward packets from one to another. The VM's two virtual ports are attached to two different VALE switches so that packets are forced to go through the NF VM (rather than forwarded locally by a VALE switch).

For all cases, we assign sufficiently higher priority to the driver domain than VMs using Xen's CPU resource prioritization mechanism so that the driver domain can be scheduled enough for handling packets. In addition, for all experiments we use netmap's `pkt-gen` application to generate and receive packets.

Unless otherwise stated, in this section we use a single CPU core for a VM. We always pin a VM's vCPU to it, as well as the VMs' I/O threads (except for the mix model). The split model requires, by definition, two CPU cores, which would make it unfair when comparing to other models and to HyperNF. To work around this, for the split model we still assign it two CPU cores (one for the VM/vCPU, one for the I/O threads running in dom0) but cap each of the cores to 50% [4]. For the driver domain driver benchmarks, we apply the same configuration as split. One CPU core is dedicated for driver I/O and the other is dedicated to the vCPU. For these cases, we use the credit scheduler in order to apply the `cap` command. [5]

4.1 Single NF

For the first experiment, we measure packet I/O performance by running a `pkt-gen` generator in one VM and a `pkt-gen` receiver on another VM (i.e., a baseline measurement without an NF). The VMs

[4]Another option would have been to downclock the cores to half, but this is not supported in PVH mode.

[5]The credit2 scheduler in Xen-4.8 does not support `cap` yet.

are connected through a VALE switch instance running a learning bridge module.

As shown in Figure 6a, HyperNF achieves the highest throughput for all packet sizes (as much as 60% higher than split for 64B packets), with the split and merge models reaching roughly similar packet rates. Next, we measure throughput when using an external host and a NIC by running a `pkt-gen` generator in a VM on the HyperNF server and a `pkt-gen` receiver on one of our 4-core servers. We run separate tests for the NIC driver running in the driver domain (DDD) and when running in the hypervisor (HVD) (Figure 6e). As shown, the split model beats merge because of merge's synchronization issues we reported on earlier. HyperNF outperforms both, achieving close to 15Mp/s for 64B packets on a single CPU core.

Although the graphs also show that HVD achieves higher throughput than DDD, the main reason is that the sender VM is allocated only 50 % of one CPU core, limiting traffic generation, whereas HVD allocates 100% of the CPU to the sender VM. We ran the case that allocates 100% of the CPU to the sender VM in DDD, and confirmed that the performance drop in comparison to HVD was at most 12.6 % (not plotted in the graphs).

Bridge: For our first NF we use a netmap-based bridge running in a VM. The VM has two virtual ports each connected to two different VALE switches. In addition, we run two `pkt-gen` processes in dom0, one acting as a generator and one as a receiver, and each connected to one of the two VALE switches (so that the bridging is done by the VM and not "locally" by the dom0 VALE switches).

Figure 6b shows the results. As in the baseline experiments, HyperNF outperforms the split and merge models. This time, split does worse than merge because of a speed mismatch between vCPU and the I/O thread [34]: whenever the I/O thread finishes quickly, the driver domain's vCPU goes into a sleep state, and waking it up incurs overheads that lower throughput. One solution would be to add a busy loop in the I/O thread, which would certainly increase throughput but at the cost of reducing efficiency and increasing energy consumption.

Next, we carry out a similar experiment using two physical NICs each connected to an external 4-core server, one running a `pkt-gen` generator and another one a receiver, with the VM acting as a forwarder as before. Figure 6f shows that once again HyperNF comes out on top, achieving the best performance when using the hypervisor-based driver (HVD).

Firewall: For the second NF we use the `netmap-ipfw` firewall with 10 rules, making sure that packets are not filtered so that all rules have to be traversed before a packet is forwarded [6]. As with the bridge NF, we use two separate VALE switches and a `pkt-gen` sender/receiver pair.

Both in the virtual port case (Figure 6c) and when using physical NICs (Figure 6g) HyperNF outperforms the merge and split models.

IP Router: For the final NF in this section we use FastClick [2], a branch of the Click modular router [20] that adds support for netmap and DPDK, among other features such as batching. We use FastClick to run a standards-compliant IP router with 10 forwarding entries in a VM.

[6]We run the test with different number of rules but this does not change HyperNF's performance relative to split/merge.

We use the same setup as with the previous NFs but tune FastClick's batch size for each model to obtain the highest throughput: 64 for HyperNF (the recommended size from the FastClick paper [2]), 128 for split and 512 for merge (the larger size for merge is to amortize VM switch costs). The pattern is as before: merge outperforms split, and HyperNF has the highest throughput, up to 4.2Mp/s for 64B packets (Figure 6d); the same holds for the physical port case, where, as before, the hypervisor-based driver beats the set-up where the driver runs in the driver domain.

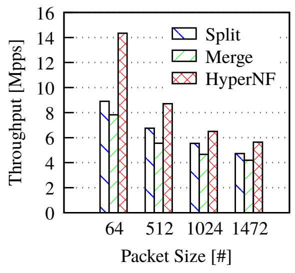
Looking at these results, the reader may be wondering why in some places HyperNF DDD performs worse than split. The performance difference comes from the balance (or unbalance) of workload in I/O threads in the driver domain and NFs running on the VMs. In the split case, packet switching, including memory copies, is executed by an I/O thread whose CPU resource is set to 50%. On the other hand, in the DDD case, all packet switching (except for updating the physical NIC registers) is accounted to the VMs' vCPU time, which is limited to 50%. As a result, the CPU resources assigned to the kernel thread only update physical NIC registers and so are mostly idle (and so wasted), while the CPU resources for the VM's vCPU are always busy because they have to execute both the NF application and virtual I/O. As a result, the split model provides better load balancing by splitting a CPU's resources 50/50 as opposed to the DDD case which does not.

Finally, note that we could have selected a wider range of NFs, or more complex ones (e.g., a full-fledged DPI based on Bro [5]). We opted not to since we do not expect overly different results from them: whether a particular NF is more CPU or I/O-intensive, HyperNF should outperform models that statically assign resources. Further, HyperNF reduces context switch overheads which would of course apply to any NF. Having said all of this, it would be interesting to actually quantify these effects, so we leave this as future work.

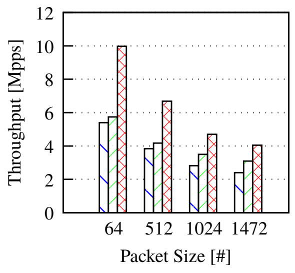
4.2 NF Consolidation

In the next set of experiments we show what happens when we start consolidating multiple NFs onto shared CPU cores. We begin by using different numbers of firewall VMs with 10 rules each, up to 36 VMs total. We dedicate 6 CPU cores to the VMs, and we assign/pin the VMs to the cores in a round-robin fashion. Out of the remaining 8 cores on the server, and in order to ensure high offered throughput, we assign 7 cores to the `pkt-gen` generator processes (there are as many of these as VMs) and the last core to a single `pkt-gen` receiver (receiving is much cheaper so a single process is enough to match all the generators). For the mix model, I/O threads are scheduled by the Linux scheduler, and for the split case 1 out of the 6 cores normally given to the VMs is dedicated to I/O. Further, we set packet size to 256B.

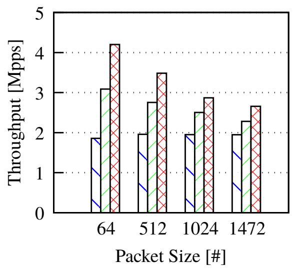
Figure 7 shows that HyperNF outperforms the split, merge and mix models, achieving close to 18Mp/s when a single NF is running on each core (6 VMs). The graph further demonstrates that HyperNF scales well when running additional numbers of NFs per core, reaching roughly 13Mp/s when 6 NFs are assigned to each core (36 VMs total). In all, this shows that HyperNF makes better



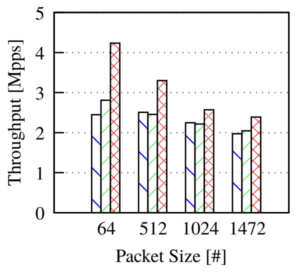
(a) No NF - virtual ports.



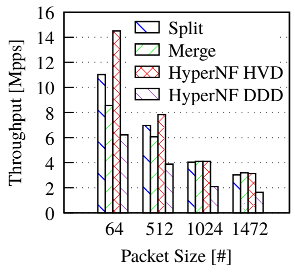
(b) Bridge NF - virtual ports.



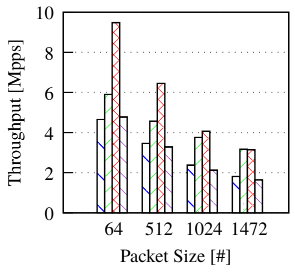
(c) Firewall NF - virtual ports.



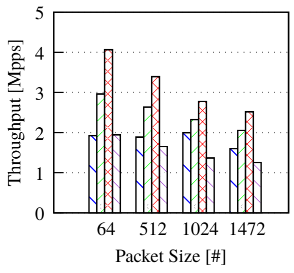
(d) Router NF - virtual ports.



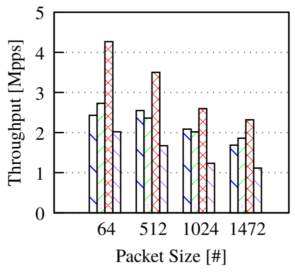
(e) No NF - physical ports.



(f) Bridge NF - physical ports.



(g) Firewall NF - physical ports.



(h) Router NF - physical ports.

Figure 6: Throughput when running different NFs one at a time using the split model, the merge model and HyperNF with virtual (top graphs) and physical ports (bottom graphs).

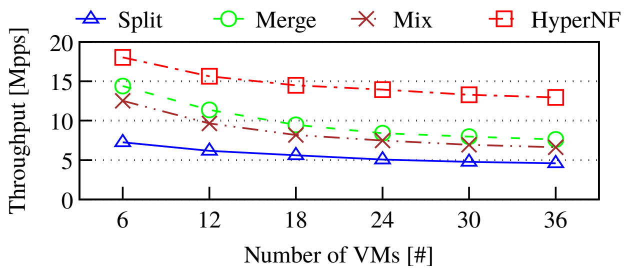


Figure 7: Cumulative throughput on the NFV platform when multiple NFs share cores. In this experiment 6 cores are dedicated to the VMs and assigned in a round-robin fashion. Packet size is 256B.

	Application		Packet Size [B]	
Setup	VM1	VM2	VM1	VM2
A	bridge	fw 50 rules	64	1472
B	fw 10 rules	fw 50 rules	1024	512

Table 1: Configurations for experiments measuring the accuracy of resource allocation.

use of available resources than split (because there's no separation between I/O and vCPU cores) and has lower synchronization overheads than merge.

4.3 Accurate Resource Allocation

We measure how closely HyperNF can comply with specified resource allocations, and compare it to the split and merge models. To set the experiment up, we create two VMs, both sharing the same CPU core without resource capping, and we set different priority settings for different runs of the experiments. To add variability in terms of load on the system, we vary packet sizes and the NFs run by the VMs according to the configurations listed in Table 1.

Each VM is connected through its own VALE instance to a pkt-gen sender and receiver pair running in dom0, with the sender generating packets as fast as possible. As the metric we calculate how much the system's goodput differs from the priority settings as $error = (goodput - expectedthroughput)/expectedthroughput \times 100$, where $expectedthroughput = baseline \times VMpriority/100$ (the baseline is the performance when a VM has 100% of a CPU core for running the NF).

The results are shown in Figure 8 (top row graphs). We observe an error of up to 47% for the merge model and for the split model errors ranging from 4% to as much as 175%. The reason for the latter's large deviation is that the baseline throughput is relatively low because of dom0 going idle and into a sleep state, as previously mentioned before in the single NF/firewall experiment. When put under higher load (two VMs requesting I/O operations) the driver domain never goes to sleep and so overall throughput jumps up, causing the error. At any rate, in all setups HyperNF produces a maximal error of only 2.9%.

For the test with physical ports we use the same set up as above but move the pkt-gen receivers to an external host (one of our

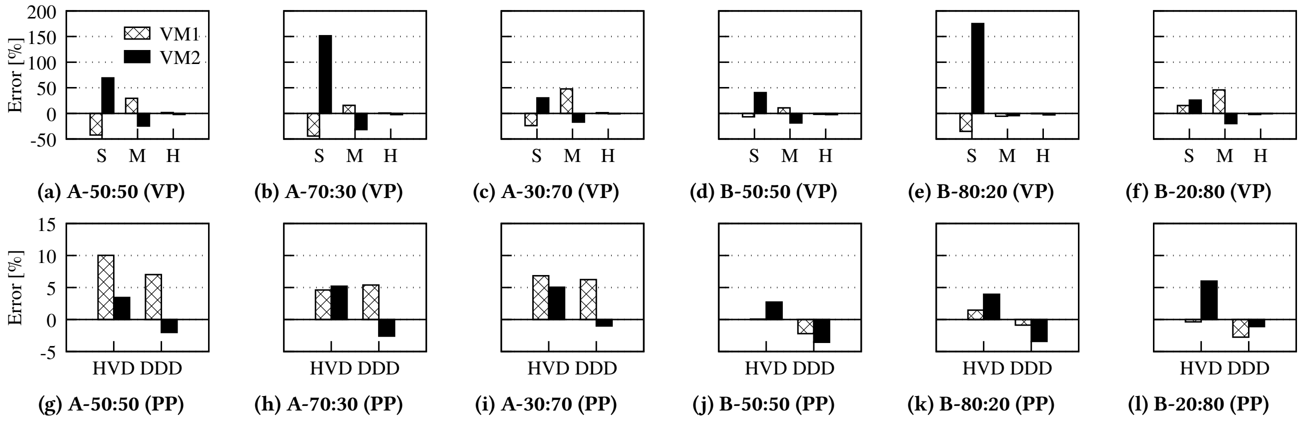


Figure 8: Accuracy of resource allocation when running on virtual ports (VP) and physical ports (PP) with two VMs and different priority ratios. In VP graphs, S, M and H represent split, merge and HyperNF, respectively. For PP, the generator runs on the same host as the VMs and the receiver on an external host. The experiment configurations (A or B) are listed in Table 1.

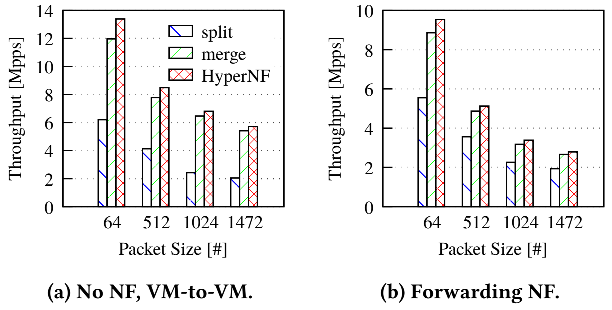


Figure 9: Comparison between the split and merge models versus HyperNF on QEMU/KVM. Figures 9a and 9b are comparable to the Xen-based Figures 6a and 6b, respectively.

4-core servers), and, as usual, run two experiments, one with the hypervisor driver (HVD) and another one with the NIC driver in the driver domain/dom0 (DDD). The results are shown in Figure 8 (bottom row graphs). In all cases, the maximum performance reduction error for DDD is 3.55% and only 0.37% for HVD.

These results show that HyperNF achieves accurate resource allocation even if we leave the physical NIC driver in the driver domain. This is because the most expensive parts of the processing, such as packet destination look-ups and memory copies from Tx to Rx rings are done in the hypercall context, with the driver domain only needing to update the NIC's register variables when invoking packet transmission.

4.4 Platform Independence

To show that HyperNF's architecture is not dependent on a particular hypervisor (i.e., Xen), we implemented packet transmission and forwarding in the context of the sender VM in Linux KVM by extending ptnetmap [9, 23], which by default supports the split and merge models.

Figure 9a plots throughput between two VMs using the split and merge models versus HyperNF, and Figure 9b does so when a third VM runs an NF that forwards packets between the other 2 VMs. As with the Xen implementation, the graphs confirm that HyperNF outperforms the other models in all cases. Margins of improvement over the merge model are smaller than those in the Xen case (Figure 6a and 6b); this is because in QEMU/KVM, a context switch to and from an I/O thread is cheaper as it is not mediated by a separate hypervisor.

These results speak to the generality of HyperNF, since it is able to speed up performance on Xen and KVM, two hypervisors with important architectural differences. The gains are smaller on KVM mostly because the hypervisor has better visibility into the resources used by the VMs; however, we believe the gains to still be significant enough to show that HyperNF can be applied to a range of hypervisors, not just Xen.

5 NFV EVALUATION

Having provided a baseline evaluation of HyperNF, the purpose of this section is to conduct experiments that would more closely mimic the sort of loads that an NFV platform is likely to encounter when deployed. We begin with an evaluation of a number of different NFV scenarios and finish with an evaluation of Service Function Chaining (SFC).

5.1 Dynamic NFV Tests

As stated in the introduction, one of the challenges when building an NFV platform is not only to be able to provide high aggregate throughput, but also to be able to adapt to changing conditions not only in terms of the traffic load but also with respect to the network functions currently running on the system.

To test HyperNF's ability to cope with such changing conditions, we conduct experiments for three sets of scenarios, each with an increasing level of variability:

- **Dynamic per-VM traffic**: Throughout the test, we vary the offered throughput sent to the NFs. The cumulative offered throughput for the entire system is constant, as are the number of VMs which run on the system; all VMs run the same NF, a firewall.

- **Dynamic cumulative traffic**: Both the cumulative throughput and the per-VM throughout is dynamic. The number of VMs is constant and they all run the same firewall NF.
- **Dynamic traffic, dynamic NFs**: Both the cumulative throughput and the per-VM throughout is dynamic. The number of VMs changes throughout the lifetime of the experiment.

In all tests we run 30 VMs on 6 CPU cores but do not pin them, leaving Xen's credit2 scheduler to perform allocation. To generate traffic, we modify `pkt-gen` so that it is possible to script what the traffic rate should be at each point in time with a second-level granularity. This lets us not only set-up the scenarios described above, but also replay/reproduce the tests both to perform fair comparisons against the split/merge/mix models and for confidence interval purposes. Each VM has its own dom0 `pkt-gen` generator, and all generators send to a single dom0 `pkt-gen` receiver (as before, we assign 7 cores to the generators and one to the receiver). In addition, each VM has two virtual ports, one connected to a VALE switch instance to which the generator is attached, and another port connected to a separate switch instance (this one common to all VMs) to which the single receiver is attached. We set packet size to 256B.

Regarding the different models, for split we dedicate two of the cores to I/O, leaving 4 for the VMs. In the merge model, the VM and related I/O thread (i.e., dom0) should run on the same core. The problem is that since we are not pinning these (pinning them would mean under-utilization and lower throughput with changing traffic conditions), the Xen scheduler will periodically shift them to idle cores, but not necessarily to the same one. To ensure that both I/O and VM stay merged, we use a simple script that, every 10 ms, ensures that the I/O runs on the same core by looking up which core the VM is running on and pinning the I/O thread to that same core. Finally, for the mix model, I/O threads run on the same 6 cores as the VMs, but they are not pinned to a CPU core: all assignments of I/O threads to CPU cores are handled by the Linux scheduler running in dom0.

With all of this in place, we first run the dynamic per-VM traffic scenario (Figure 10). HyperNF yields the highest cumulative throughput (i.e., the aggregate of forwarded packets from all VMs).

We plot the results for the second scenario, dynamic cumulative traffic, in Figure 11a. HyperNF not only results in the highest throughput peaks, but it is also able to quickly adapt to changing traffic conditions, closely matching, for the most part, the offered throughput (i.e., the black line labeled "Total"). One exception is at $t = 18$, where HyperNF is still the highest curve but experiences a noticeable drop with respect to the offered throughput. To explain this, Figure 11b plots a CDF of the traffic rates of each of the 30 flows (one per VM) present at $t = 18$. The CDF shows a large number of middle-sized flows (1.5-2Mp/s). What happens is that while a VM is running on a core, because the flows have fairly high traffic rates the queues of other VMs waiting to be scheduled fill up quickly and drop packets. When the distribution is more skewed towards a few large flows and a number of small ones (e.g., at $t = 24$ and $t = 33$), the large flows are serviced, and while they are, the queues of the VMs that are not running are sufficiently large to not drop packets.

In the final scenario (dynamic traffic, dynamic NFs) we use the same setup as in the previous scenario, except this time we begin with only 15 VMs at $t = 0$ and bring three new VMs up every 5

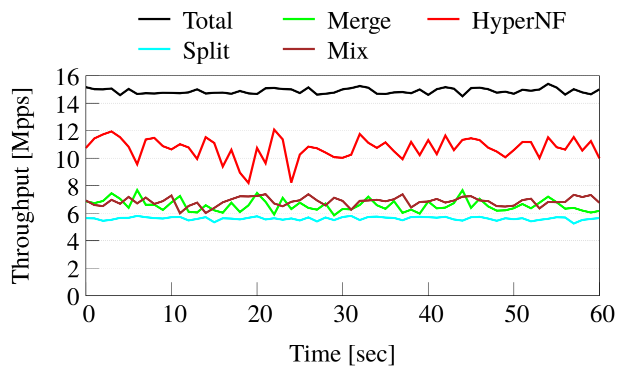
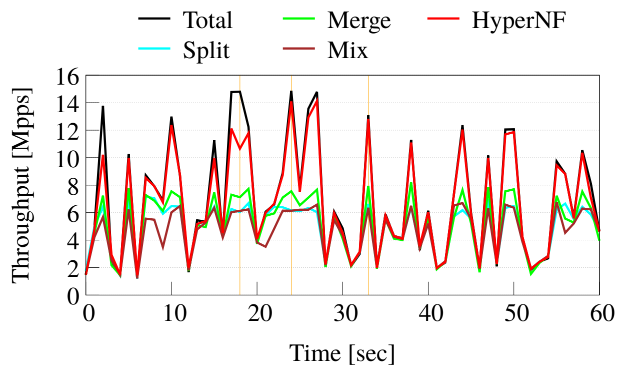
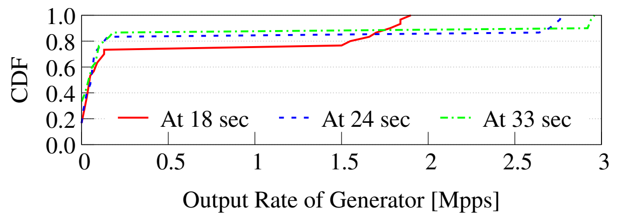


Figure 10: Cumulative forwarded throughput for the dynamic per-VM traffic scenario. Packet size is 256B and the VMs all run a 10-rule firewall. The "Total" curve is the offered throughput.

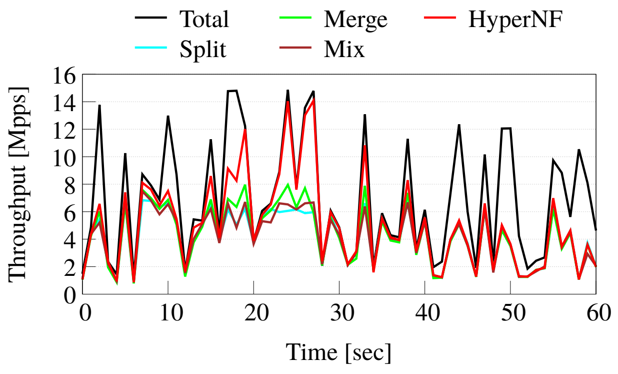


(a) Throughput.

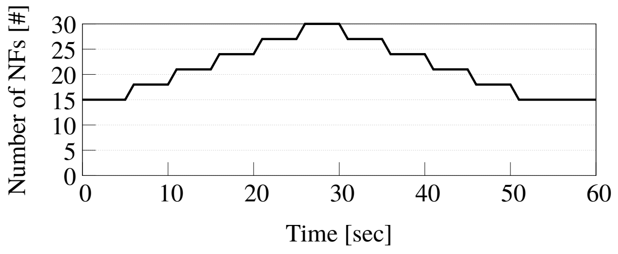


(b) Traffic matrix.

Figure 11: Cumulative forwarded throughput for the dynamic cumulative traffic scenario. Packet size is 256B and the VMs all run a 10-rule firewall. The "Total" curve is the offered throughput.



(a) Throughput



(b) Number of Active NFs

Figure 12: Cumulative forwarded throughput for the dynamic traffic, dynamic NFs scenario. Packet size is 256B and the VMs all run a 10-rule firewall. The "Total" curve is the offered throughput.

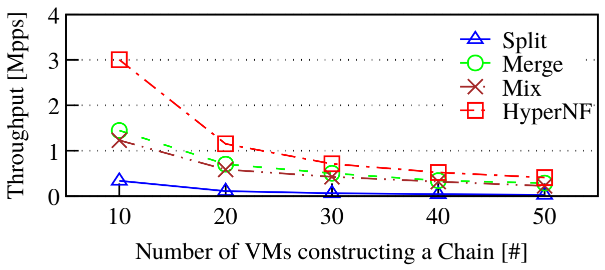
seconds (in actuality we unpause existing VMs so that they become immediately active), up to a total of 30 VMs. Then, starting at $t = 30$, we pause 3 VMs every 5 seconds until we reach the original 15 VMs; Figure 12b plots the number of NFs over time.

The results in Figure 12a demonstrate HyperNF provides higher throughput than merge, split and mix, and that its curve closely followed the offered throughput (except for a peak at roughly $t = 18$ as in the previous experiment). This shows HyperNF's ability to cope with not only changing traffic loads but with variability in the number of concurrently running NFs.

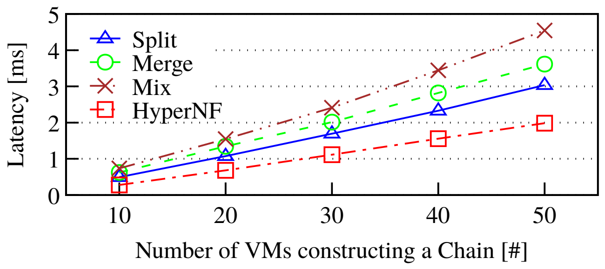
5.2 Service Function Chaining

Service Function Chaining has become an essential part of an NFV platform's toolkit. In this final experiment, we evaluate HyperNF's performance when running potentially long chains, measuring how throughput decreases and delay increases as a function of chain length.

Each VM has two virtual ports and runs a 10-rule firewall. We run a single `pkt-gen` sender in dom0 to send 256B packets to the first VM in the chain, and a single `pkt-gen` receiver also in dom0 to sink packets from the last VM in the chain. In addition, we use 10 CPU cores for the VMs except for the split model, which uses 9 cores for the VMs and the leftover core for I/O. In addition to throughput, we measure latency using `pkt-gen` in ping-pong mode.



(a) Throughput



(b) Latency

Figure 13: Service chain throughput and latency performance for increasing chain lengths.

The results are plotted in Figure 13. For all chain lengths, HyperNF shows the highest throughput and the lowest drop in performance as chains gets longer. In terms of latency, HyperNF yields a maximum of 2 milliseconds for a very long chain (50 NFs); this is low compared to typical end-to-end latencies in the Internet.

6 RELATED WORK

NFV frameworks: The research community has explored joint optimizations of NFV platforms and the individual NFs with the goal of achieving higher performance and density. NetBricks [28] and SoftFlow [19] run middleboxes as function calls in their own context; Flurries [44] and OpenNetVM [45] run lightweight containers; E2 [27] and BESS [12] provide rich middlebox programming and placement interfaces. However, these frameworks require middlebox software to be rewritten, not just to use new packet I/O APIs but to adapt to their specific execution and protection models, imposing radical software architecture redesigns. HyperNF supports a standard NFV architecture and does not impose any modifications to existing software [7, 13, 26].

Virtual networking architectures: Elvis [14] reduces I/O threads in the KVM backend to reduce context switches between them. vRIO [21] runs such threads in dedicated VMs. ptnetmap [9, 23] and ClickOS [24] keep I/O threads in the backend switch in order to reduce VM scheduling. All of the above are categorized into either the split or merge models, both of which are, as we have found, inefficient for NFV workloads. HyperSwitch [30] improves intra-VM communication for bulk TCP traffic (e.g., 64KB "packets"), running an Open vSwitch datapath within the Xen hypervisor. Its architecture is thus somewhat similar to HyperNF, but HyperNF provides an order of magnitude higher inter-VM packet rates, addresses problems such as VM switch overheads, and can adapt to changing traffic conditions and to the number of NFs being run on the system.

[25, 31, 38] are optimizations for the standard network backend of Xen. [36] optimizes NIC drivers for QEMU/KVM. ELI [10] mitigates overheads caused by VM exits. Reducing VM exits [1, 22] has been a common practice for performance improvements, but HyperNF shows that aggressively exploiting batching with reducing scheduling overheads dramatically amortizes the costs of kernel and hypervisor crossing [16, 39, 43]. vTurbo [42] reduces the latency of virtual I/O execution with fine-grained scheduling; HyperNF achieves the same goal by executing virtual I/O within a hypercall.

7 DISCUSSION AND CONCLUSION

This paper addressed the problem of building a high performance and efficient VM-based NFV platform. We demonstrated the effects of CPU assignment to VMs and their I/O threads under different models and showed that running VMs and their I/O threads on the same CPU core yields the best results. However, we showed that even with this best practice, existing VM networking architectures either under-utilize the available resources (especially under changing load conditions) or incur excessive synchronization overheads when switching between VMs and I/O threads. To address these issues, we presented HyperNF, an NFV platform that adopts an effective virtual I/O model. By implementing the data paths of a virtual switch and of a physical NIC driver in the hypervisor, and by using a hypercall as a control path to access these data paths, HyperNF eliminates scheduling and VM switching costs for each I/O operation and mitigates the cost of sharing a core between VM and I/O thread. The end result is a platform that provides high throughput, utilization and adaptability to changing loads.

HyperNF could also be applicable to other, non-NFV workloads that at least partly depend on I/O operations. Such applications include in-memory key-value stores, web servers, data analytics frameworks and DNS servers, to name a few. We leave an investigation of this direction of research as future work.

8 ACKNOWLEDGMENTS

This paper has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 671566 ("Superfluidity"). This paper reflects only the author's views and the European Commission is not responsible for any use that may be made of the information it contains.

REFERENCES

[1] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. 2012. Software Techniques for Avoiding Hardware Virtualization Exits. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, Boston, MA, 373–385. https://www.usenix.org/conference/atc12/technical-sessions/presentation/agesen

[2] T. Barbette, C. Soldani, and L. Mathy. 2015. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 5–16. https://doi.org/10.1109/ANCS.2015.7110116

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[4] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401

[5] Bro. 2017. The Bro Network Security Monitor. https://www.bro.org/. (2017).

[6] Cisco. 2014. The Zettabyte Era—Trends and Analysis - CISCO White Paper. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html. (2014).

[7] Fortinet. 2014. Fortigate Next Generation Firewall Virtual Appliances. https://www.fortinet.com/products/virtualized-security-products/fortigate-virtual-appliances.html. (2014).

[8] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. 2004. *Reconstructing i/o*. Technical Report. University of Cambridge, Computer Laboratory.

[9] S. Garzarella, G. Lettieri, and L. Rizzo. 2015. Virtual device passthrough for high speed VM networking. In *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*. 99–110. https://doi.org/10.1109/ANCS.2015.7110124

[10] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. ELI: Bare-metal Performance for I/O Virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 411–422. https://doi.org/10.1145/2150976.2151020

[11] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. 2006. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware (Middleware '06)*. Springer-Verlag New York, Inc., New York, NY, USA, 342–362. http://dl.acm.org/citation.cfm?id=1515984.1516011

[12] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. SoftNIC: A software NIC to augment hardware. *Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155* (2015).

[13] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. 2005. Designing extensible IP router software. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 189–202.

[14] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. 2013. Efficient and Scalable Paravirtual I/O System. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX, San Jose, CA, 231–242. https://www.usenix.org/conference/atc13/technical-sessions/presentation/har

[15] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. 2015. mSwitch: A Highly-scalable, Modular Software Switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. ACM, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/2774993.2775065

[16] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. 2014. Rekindling Network Protocol Innovation with User-level Stacks. *SIGCOMM Comput. Commun. Rev.* 44, 2 (April 2014), 52–58. https://doi.org/10.1145/2602204.2602212

[17] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 445–458. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang

[18] Intel. 2017. Intel DPDK: Data Plane Development Kit. http://dpdk.org/. (2017).

[19] Ethan J. Jackson, Melvin Walls, Aurojit Panda, Justin Pettit, Ben Pfaff, Jarno Rajahalme, Teemu Koponen, and Scott Shenker. 2016. Softflow: A Middlebox Architecture for Open vSwitch. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 15–28. http://dl.acm.org/citation.cfm?id=3026959.3026962

[20] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* 18, 3 (Aug. 2000), 263–297. https://doi.org/10.1145/354871.354874

[21] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. 2016. Paravirtual Remote I/O. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 49–65. https://doi.org/10.1145/2872362.2872378

[22] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. 2011. SplitX: Split Guest/Hypervisor Execution on Multi-core. In *Proceedings of the 3rd Conference on I/O Virtualization (WIOV'11)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=2001555.2001556

[23] V. Maffione, L. Rizzo, and G. Lettieri. 2016. Flexible virtual machine networking using netmap passthrough. In *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. 1–6. https://doi.org/10.1109/LANMAN.2016.7548852

[24] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function

Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 459–473. http://dl.acm.org/citation.cfm?id=2616448.2616491

[25] Aravind Menon, Alan L Cox, and Willy Zwaenepoel. 2006. Optimizing network virtualization in Xen. In *USENIX Annual Technical Conference*.

[26] netgate. 2017. pfSence Virtual Appliances. https://www.netgate.com/appliances/pfsense-virtual-appliances.html. (2017).

[27] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 121–136. https://doi.org/10.1145/2815400.2815423

[28] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 203–216. http://dl.acm.org/citation.cfm?id=3026877.3026894

[29] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 117–130. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff

[30] Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha, and Scott Rixner. 2013. Hyper-Switch: A Scalable Software Virtual Switching Architecture. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX, San Jose, CA, 13–24. https://www.usenix.org/conference/atc13/technical-sessions/presentation/ram

[31] Kaushik Kumar Ram, Jose Renato Santos, Yoshio Turner, Alan L. Cox, and Scott Rixner. 2009. Achieving 10 Gb/s Using Safe and Transparent Network Interface Virtualization. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '09)*. ACM, New York, NY, USA, 61–70. https://doi.org/10.1145/1508293.1508303

[32] Luigi Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=2342821.2342830

[33] Luigi Rizzo. 2017. netmap-ipfw. https://github.com/luigirizzo/netmap-ipfw. (2017).

[34] Luigi Rizzo, Stefano Garzarella, Giuseppe Lettieri, and Vincenzo Maffione. 2016. A Study of Speed Mismatches Between Communicating Virtual Machines. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems (ANCS '16)*. ACM, New York, NY, USA, 61–67. https://doi.org/10.1145/2881025.2881037

[35] Luigi Rizzo and Giuseppe Lettieri. 2012. VALE, a Switched Ethernet for Virtual Machines. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2413176.2413185

[36] Luigi Rizzo, Giuseppe Lettieri, and Vincenzo Maffione. 2013. Speeding Up Packet I/O in Virtual Machines. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '13)*. IEEE Press, Piscataway, NJ, USA, 47–58. http://dl.acm.org/citation.cfm?id=2537857.2537864

[37] Luigi Rizzo, Paolo Velente, Giuseppe Lettieri, and Vincenzo Maffione. 2016. *PSPAT: Software Packet Scheduling at Hardware Speed*. Technical Report. Università di Pisa.

[38] Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Ian Pratt. 2008. Bridging the Gap Between Software and Hardware Techniques for I/O Virtualization. In *USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, Berkeley, CA, USA, 29–42. http://dl.acm.org/citation.cfm?id=1404014.1404017

[39] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 33–46. http://dl.acm.org/citation.cfm?id=1924943.1924946

[40] Elena Ufimtseva. 2015. PVH: Faster, improved guest model for Xen. http://events.linuxfoundation.org/sites/events/files/slides/PVH_Oracle_Slides_LinuxCon_final_v2_0.pdf. (2015).

[41] Open vSwitch. 2017. Open vSwitch. http://www.openvswitch.org/. (2017).

[42] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. 2013. vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX, San Jose, CA, 243–254. https://www.usenix.org/conference/atc13/technical-sessions/presentation/xu

[43] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 43–56. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata

[44] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. 2016. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. ACM, New York, NY, USA, 3–17. https://doi.org/10.1145/2999572.2999602

[45] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMIddlebox '16)*. ACM, New York, NY, USA, 26–31. https://doi.org/2940147.2940155