SUPERFLUIDITY

A SUPER-FLUID, CLOUD-NATIVE, CONVERGED EDGE SYSTEM

Research and Innovation Action GA 671566

DELIVERABLE D5.2

MECHANISMS FOR NETWORK SERVICE DYNAMICS AND PERFORMANCE

Deliverable Type:	Report
Dissemination Level:	PU
Contractual Date of Delivery to the EU:	31/12/2017
Actual Date of Delivery to the EU:	14/2/2018
Workpackage Contributing to the Deliverable:	WP5

Editor(s):	Felipe Huici (NEC)
Author(s):	Felipe Huici (NEC), Costin Raiciu (UPB), Emil Matus (TUD), George Tsolis (CITRIX) Claudio Pisa, Stefano Salsano, Francesco Lombardo, Nicola Blefari-Melazzi, Giuseppe Bianchi (CNIT), Louise Krug, Paul Veitch, Philip Eardley (BT), Tomasz Kanceki, Edel Curley (Intel), John Thomson, Michail Flouris, Anastassios Nanos, Xenia Ragiadakos, Julian Chesterfield, Kevin Du (OnApp), Luis Tomás (Red Hat)
Internal Reviewer(s)	Stefano Salsano (CNIT)

Abstract: This deliverable describes all the mechanisms developed within the project in order to implement the Superfluid platform, and, in particular, how they come together to provide unprecedented superfluid dynamics such as boot times in as little as a few milliseconds (comparable to process start-up times), fast migration times and tiny VM memory footprints (a few MBs).

Keyword List: Superfluid, virtual machines, virtualization, high performance, unikernels, performance optimizations, isolation, network functions, NFV, cloud, data centers. Service aggregation. Scalability.

INDEX

1 Introduction

One of the main objectives of the Superfluidity project was to design, implement and test a superfluid platform able to achieve performance KPIs such as boot times of a few milliseconds, memory footprints of a few MBs and massive consolidation, the ability to run thousands of instances concurrently on inexpensive, off-the-shelf hardware, all of it while providing a platform for running network functions in a strongly isolated manner. This deliverable describes all the research and technological advances achieved by the project in order to make this vision a reality.

We begin in section 2 by describing the core of the platform which we call LightVM. LightVM consists of the development of a number of high-performance, extremely lean *unikernels* along with a re-vamping of one of the major mainstream virtualization frameworks. The work, published at SOSP 2017 (the top operating systems conference in the world) is the first, to the best of our knowledge, to show that virtual machines can have performance properties equal to or better than containers *without* having to sacrifice any isolation or security the way containers do. This work has been released as open source and more information about it can be found at http://sysml.neclab.eu/projects/lightvm/.

Section 3 then describes further optimizations to the platform's hypervisor, scheduling and network sub-systems in order to make it an efficient platform for NFV, taking into account the high packet rates that the platform would be likely to encounter in operator networks; the work was published in SoCC 2017 (the top cloud computing conference in the world). This work is also open source; more information is at http://sysml.neclab.eu/projects/hypernf/.

Section 4 further optimizes the platform by looking at the issue of how to fairly share CPUs in the platform. The results show that sharing CPUs via traditional OS mechanisms is suboptimal, as it leads to inefficient and unfair resource allocations in many situations. Instead, the work proposes changes to the buffering discipline used by the network processes that remedy the problems identified in our experiments and shows that these changes achieve both good fairness and resource allocation. This work was published at SIGCOMM 2017's KBNets workshop.

In section 5 we face the issue of fast orchestration of unikernels, specifically targeting the Light VM solution that has been described in section 2 and section 3. We clearly identify the performance bottlenecks related to the orchestration of unikernels with state-of-the-art Virtual Infrastructure Managers (VIMs), like OpenStack and OpenVIM. We designed and implemented enhancements to

the VIMs to drastically reduce the time needed to orchestrate (deploy) unikernel VMs in OpenStack and OpenVIM. The implementation of this work has been released as open source (see https://github.com/superfluidity/vim-tuning-and-eval-tools) and contributed to the mainstream OpenVIM implementation. The first results of this work have already been published in the IEEE 2016 NFV-SDN Conference. At the time of writing, a journal submission to IEEE Transaction of Cloud Computing is being finalized.

Section 6 addresses the issue of optimizing superfluid platform deployments targeting containers, as far as memory usage is concerned. We focused on a concrete use case related to a Load Balancer VNF deployed using a Docker container. We demonstrated that with a set of enhancements it is possible to reduce the memory requirements for the VNF from 2GB to 1 GB.

Section 7 focuses on functionality for the platform to efficiently run parallel baseband signal processing slices in a RAN deployment. We discuss the design and implementation of a dataflow engine dedicated to such processing, based on two components called SliceManager and CoreManager. A performance analysis of the CoreManager subsystem is provided and application traces are presented that show the functionality of the proposed dataflow framework.

Section 8 then describes the modifications made to the OpenStack/Superfluidity platform's back-end software switch (Open vSwitch) in order to improve its performance by increasing its throughput and minimizing its latency, as well as making it more scalable.

This is followed by section 9, which considers the problem of protecting Virtual Network Function (VNF) resources against “Noisy Neighbour” effects, i.e. the degradation of performance due to the presence of other VNFs that compete for the resources. A solution to mitigate this issue is proposed, based on the recently released cache allocation technology from Intel.

Section 10 describes optimizations to the MicroVisor hypervisor platform to support network functions and RFBs at the density and scale needed by Superfluidity’s platform.

Finally, section 11 provides the conclusions and the appendix examines how to operate and manage Network Functions inside containers.

2 LightVM: The Superfluid Platform

The results reported in this section have been published in:

F. Manco, C. Lupu, F. Schmidt, J. Mendes, Simon Kuenzer, S. Sati, K. Yasukata, C. Raiciu, F. Huici, "My VM is Lighter (and Safer) than your Container", 26th Symposium on Operating Systems Principles. ACM SOSP 2017, October 28-31, 2017, Shanghai, China.

Available at: http://sysml.neclab.eu/projects/lightvm/lightvm.pdf

Large portions of text in the following sub-sections correspond to the text in the paper.

2.1 Introduction

Lightweight virtualization technologies such as Docker [6] and LXC [25] are gaining enormous traction. Google, for instance, is reported to run all of its services in containers [4], and Container as a Service (CaaS) products are available from a number of major players including Azure's Container Service [32], Amazon's EC2 Container Service and Lambda offerings [1, 2], and Google's Container Engine service [10]. Beyond these services, lightweight virtualization is crucial to a wide range of use cases, including just-in-time instantiation of services [23, 26] (e.g., filters against DDoS attacks, TCP acceleration proxies, content caches, etc.) and NFV [41, 51], all while providing significant cost reduction through consolidation and power minimization [46].

The reasons for containers to have taken the virtualization market by storm are clear. In contrast to heavyweight, hypervisor-based technologies such as VMWare, KVM or Xen, they provide extremely fast instantiation times, small per-instance memory footprints, and high density on a single host, among other features.

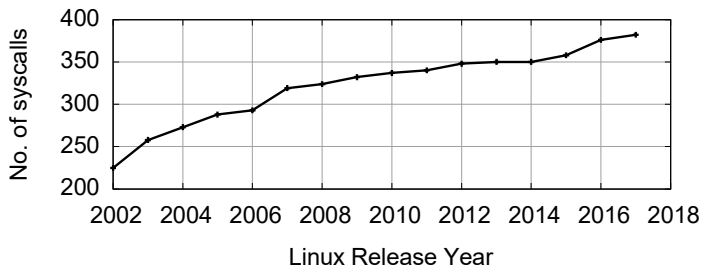


Figure 1: The unrelenting growth of the Linux syscall API over the years (x86_32) underlines the difficulty of securing containers.

However, no technology is perfect, and containers are no exception: security is a continuous thorn in their side. The main culprit is the hugely powerful kernel syscall API that containers use to interact

with the host OS. This API is very broad as it offers kernel support for process and thread management, memory, network, filesystems, IPC, and so forth: Linux, for instance, has 400 different system calls [37], most with multiple parameters and many with overlapping functionality; moreover, the number of syscalls is constantly in- creasing (see figure 1). The syscall API is fundamentally more difficult to secure than the relatively simple x86 ABI offered by virtual machines where memory isolation (with hardware support) and CPU protection rings are sufficient. Despite the many isolation mechanisms introduced in the past few years, such as process and network namespaces, root jails, seccomp, etc, containers are the target of an ever increasing number of exploits [11, 22]. To complicate matters, any container that can monopolize or exhaust system resources (e.g., memory, file descriptors, user IDs, forkbombs) will cause a DoS attack on all other containers on that host [14, 35].

At least for multi-tenant deployments, this leaves us with a difficult choice between (1) containers and the security issues surrounding them and (2) the burden coming from heavyweight, VM-based platforms. Securing containers in the context of an ever-expanding and powerful syscall API is elusive at best. Could we make virtualization faster and more nimble, much like containers? The explicit goal would be to achieve performance in the same ball-park as containers: instantiation in milliseconds, instance memory footprints of a few MBs or less, and the ability to concurrently run one thousand or more instances on a single host.

In this section we introduce LightVM, a lightweight virtualization system based on a type-1 hypervisor. LightVM retains the strong isolation virtual machines are well-known for while providing the performance characteristics that make containers such an attractive proposition. In particular, we make the following contributions:

- An analysis of the performance bottlenecks preventing traditional virtualization systems from achieving container-like dynamics (we focus our work on Xen).
- An overhaul of Xen's architecture, completely removing its back-end registry (a.k.a. the XenStore), which constitutes a performance bottleneck. We call this noxs (no XenStore), and its implementation results in significant improvements for boot and migration times, among other metrics.
- A revamp of Xen's toolstack, including a number of optimizations and the introduction of a split toolstack that separates functionality that can be run periodically, offline, from that which must be carried out when a command (e.g., VM creation) is issued.
- The development of Tinyx, an automated system for building minimalistic Linux-based VMs, as well as the development of a number of unikernels. These light- weight VMs are fundamental to achieving high performance numbers, but also for discovering performance bottlenecks in the underlying virtualization platform.

- A prototypical implementation along with an extensive performance evaluation showing that LightVM is able to boot a (unikernel) VM in as little as 2.3ms, reach same-host VM densities of up to 8000 VMs, migration and suspend/resume times of 60ms and 30ms/25ms respectively, and per-VM memory footprints of as little as 480KB (on disk) and 3.6MB (running).

To show its applicability, we use LightVM to implement four use cases: personalized firewalls, just-in-time service instantiation, high density TLS termination and a lightweight compute service akin to Amazon Lambda or Google's Cloud Functions but based on a Python unikernel. LightVM is available as open source at http://sysml.neclab.eu/projects/lightvm .

2.2 Requirements

The goal is to be able to provide lightweight virtualization on top of hypervisor technology. More specifically, as requirements, we are interested in a number of characteristics typical of containers:

- Fast Instantiation: Containers are well-known for their small start-up times, frequently in the range of hundreds of milliseconds or less. In contrast, virtual machines are infamous for boot times in the range of seconds or longer.
- High Instance Density: It is common to speak of running hundreds or even up to a thousand containers on a single host, with people even pushing this boundary up to 10,000 containers [17]. This is much higher than what VMs can typically achieve, which lies more in the range of tens or hundreds at most, and normally requires fairly powerful and expensive servers.
- Pause/unpause: Along with short instantiation times, containers can be paused and unpaused quickly. This can be used to achieve even higher density by pausing idle instances, and more generally to make better use of CPU resources. Amazon Lambda, for instance, "freezes" and "thaws" containers.

The single biggest factor limiting both the scalability and performance of virtualization is the size of the guest virtual machines: for instance, both the on-disk image size as well as the running memory footprint are on the order of hundreds of megabytes to several gigabytes for most Linux distributions. VM memory consumption imposes a hard up- per bound on the number of instances that can be run on the same server. Containers typically require much less memory than virtual machines (a few MBs or tens of MBs) because they share the kernel and have smaller root filesystems.

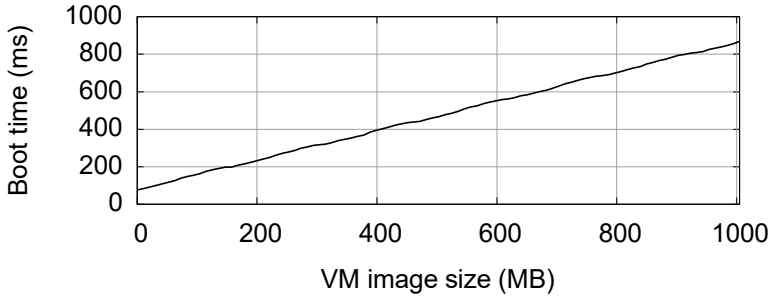


Figure 2: Boot times grow linearly with VM image size

Large VMs also slow down instantiation times: the time needed to read the image from storage, parse it and lay it out in memory grows linearly with image size. This effect is clearly shown in Figure 2 where we boot the same unikernel VM from images of different sizes, all stored in a ramdisk. We increase the size by injecting binary objects into the uncompressed image le. This ensures that the results are due to the effects that image size has on VM initialization.

2.3 Lightweight VMs

The first step towards achieving our goals is to reduce both the image size and the memory footprint of virtual machines. We observe, as others [27], that most containers and virtual machines run a single application; if we reduce the functionality of the VM to include only what is necessary for that application, we expect to reduce the memory usage dramatically. Concretely, we explore two avenues to optimize virtual machine images:

- Unikernels: tiny virtual machines where a minimalistic operating system (such as MiniOS [34]) is linked directly with the target application. The resulting VM is typically only a few megabytes in size and can only run the target application; examples include ClickOS [29] and Mirage [27].
- Tinyx: a tool that we have built to create a tiny Linux distribution around a specified application. This results in images that are a few tens of MBs in size and need around 30MBs of RAM to boot.

2.3.1 Unikernels

There is a lot of prior work showing that unikernels have very low memory footprint, and for specific applications there already exist images that one can re-use: ClickOS is one such example that can run custom Click modular router configurations composed of known elements. Mirage [27] is another example that takes applications written in OCaml and creates a minimalistic app+OS combo that is packed as a guest VM.

If one needs to create a new unikernel, the simplest is to rely on Mini-OS [34], a toy guest operating system distributed with Xen: its functionality is very limited, there is no user/kernel separation and no processes/fork. For instance, only 50 LoC are needed to implement a TCP server over Mini-OS that returns the current time whenever it receives a connection (we also linked the lwip networking stack). The resulting VM image, which we will refer to as the daytime unikernel, is only 480KB (uncompressed), and can run in as little as 3.6MB of RAM. We use the daytime unikernel as a lower bound of memory consumption for possible VMs.

We have also created unikernels for more interesting applications, including a TLS termination proxy and Minipython, a Micropython-based unikernel to be used by Amazon lambda-like services; both have images of around 1MB and can run with just 8MB of memory.

In general, though, linking existing applications that rely on the Linux syscall API to Mini-OS is fairly cumbersome and requires a lot of expert time. That is why we also explored another approach to creating lightweight VMs based on the Linux kernel, described next.

2.3.2 Tinyx

Tinyx is an automated build system that creates minimalistic Linux VM images targeted at running a single application (al- though the system supports having multiple ones). The tool builds, in essence, a VM consisting of a minimalistic, Linux- based distribution along with an optimized Linux kernel. It provides a middle point between a highly specialized unikernel, which has the best performance but requires porting of applications to a minimalistic OS, and a full-edged general- purpose OS VM that supports a large number of applications out of the box but incurs performance overheads.

The Tinyx build system takes two inputs: an application to build the image for (e.g., nginx) and the platform the image will be running on (e.g., a Xen VM). The system separately builds a filesystem/distribution and the kernel itself. For the distribution, Tinyx includes the application, its dependencies, and BusyBox (to support basic functionality). To derive dependencies, Tinyx uses (1) objdump to generate a list of libraries and (2) the Debian package manager.

To optimize the latter, Tinyx includes a blacklist of packages that are marked as required (mostly for installation, e.g., dpkg) but not strictly needed for running the application. In addition, we include a whitelist of packages that the user might want to include irrespective of dependency analysis. Tinyx does not directly create its images from the packages since the packages include installation scripts which expect utilities that might not be available in the minimalistic Tinyx distribution. Instead, Tinyx first mounts an empty OverlayFS directory over a Debian minimal debootstrap system. In this mounted directory we install the minimal set of packages discovered earlier as would be normally done in Debian. Since this is done on an overlay mounted system, unmounting this overlay gives us all the les which are properly configured as they would be on a Debian system. Before unmounting,


we remove all cache les, any dpkg/apt related files, and other unnecessary directories. Once this is done, we overlay this directory on top of a BusyBox image as an underlay and take the contents of the merged directory; this ensures a minimalistic, application-specific Tinyx "distribution". As a final step, the system adds a small glue to run the application from BusyBox's init.

To build the kernel, Tinyx begins with the "tinycong" Linux kernel build target as a baseline, and adds a set of built-in options depending on the target system (e.g., Xen or KVM support); this generates a working kernel image. By default, Tinyx disables module support as well as kernel options that are not necessary for virtualized systems (e.g., baremetal drivers). Optionally, the build system can take a set of user-provided kernel options, disable each one in turn, rebuild the kernel with the olddefconfig target, boot the Tinyx image, and run a user-provided test to see if the system still works (e.g., in the case of an nginx Tinyx image, the test includes attempting to wget a le from the server); if the test fails, the option is re-enabled, otherwise it is left out of the configuration. Combined, all these heuristics help Tinyx create kernel images that are half the size of typical Debian kernels and significantly smaller runtime memory usage (1.6MB for Tinyx vs. 8MB for the Debian we tested).

2.4 Virtualization Today

Armed with our tiny VM images, we are now ready to explore the performance of existing virtualization technologies. We base our analysis on Xen [3], which is a type-1 hypervisor widely used in production (e.g., in Amazon EC2). Xen has a small trusted computing base and its code is fairly mature, resulting in strong isolation (the ARM version of the hypervisor, for instance, consists of just 11.4K LoC [43], and dissagregation [5] can be used to keep the size of critical Dom0 code low). The competing hypervisor, KVM, is based on the Linux kernel and has a much larger trusted computing base. To better understand the following investigation, we start with a short introduction on Xen.

2.4.1 Short Xen Primer

The Xen hypervisor only manages basic resources such as CPUs and memory (see Figure 3). When it finishes booting, it automatically creates a special virtual machine called Dom0. Dom0 typically runs Linux and hosts the toolstack, which includes the xl command and the libxl and libxc libraries needed to carry out commands such as VM creation, migration and shutdown. Dom0 also hosts the XenStore, a proc-like central registry that keeps track of management information such as which VMs are running and information about their devices, along with the libxs library containing code to interact with it. The XenStore provides watches that can be associated with particular directories of the store and that will trigger callbacks whenever those directories are modified.

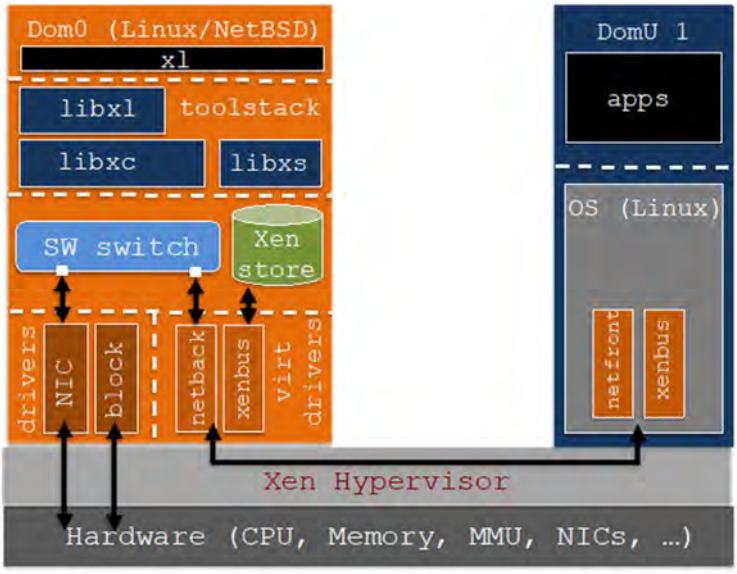


Figure 3: The Xen architecture including toolstack, the XenStore, software switch and split drivers between the driver domain (Dom0) and the guests (DomUs).

Typically, Dom0 also hosts a software switch (Open vSwitch is the default) to mux/demux packets between NICs and the VMs, as well as the (Linux) drivers for the physical devices. For communication between Dom0 and the other guests, Xen implements a split-driver model: a virtual back-end driver running in Dom0 (e.g., the netback driver for networking) communicates over shared memory with a front-end driver running in the guests (the netfront driver). So-called event channels, essentially software interrupts, are used to notify drivers about the availability of data

2.4.2 Overhead Investigation

As a testing environment, we use a machine with an Intel Xeon E5-1630 v3 CPU at 3.7 GHz and 128 GiB of DDR4 memory, and Xen and Linux versions 4.8. We then sequentially start 1000 virtual machines and measure the time it takes to create each VM (the time needed for the toolstack to prepare the VM), and the time it takes the VM to boot. We do this for three types of VMs that exemplify vastly different sizes: first, a VM running a minimal install of Debian jessie that we view as a typical VM used in practice; second, Tinyx, where the distribution is bundled into the kernel image as an initramfs; and finally, the daytime unikernel.

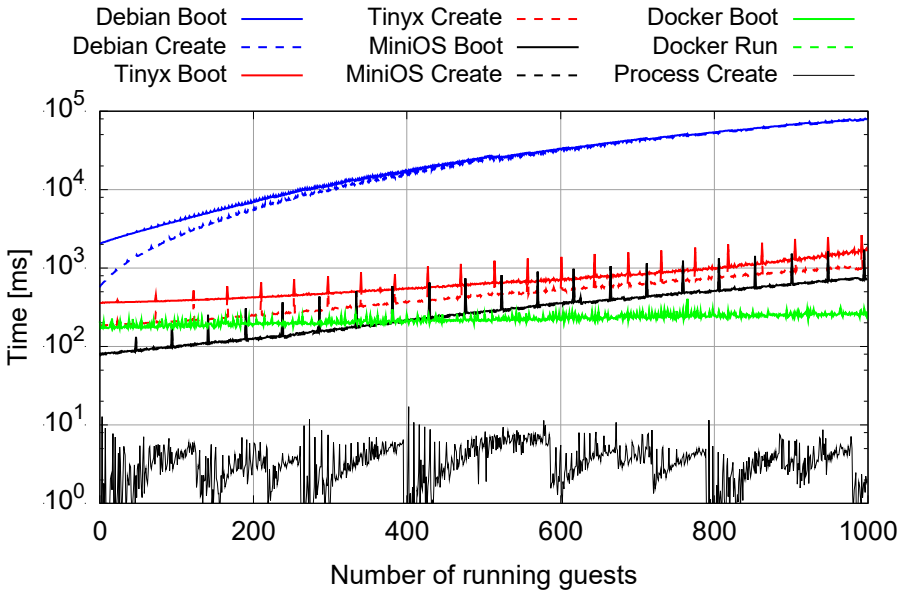


Figure 4: Comparison of domain instantiation and boot times for several guest types. With small guests, instantiation accounts for most of the delay when bringing up a new VM

Figure 4 shows creation and boot times for these three VM types, Docker containers, and basic Linux processes (as a baseline). All VM images are stored on a ramdisk to eliminate the effects that disk I/O would have on performance.

The Debian VM is 1.1GB in size; it takes Xen around 500ms to create the VM when there are no other VMs running, and it takes the VM 1.5 seconds to boot. The Tinyx VM (9.5MB image) is created in 360ms and needs a further 180ms to boot. The first unikernel (480KB image) is created in 80ms, and needs 3ms to boot.

Docker containers start in around 200ms, and a process is created and launched (using fork/exec) in 3.5ms on average (9ms at the 90% percentile). However, for both processes and containers creation time does not depend on the number of existing containers or processes.

As we keep creating VMs, however, the creation time increases noticeably (note the logarithmic scale): it takes 42s 10s and 700ms to create the thousandth Debian, Tinyx, and unikernel guest, respectively. These results are surprising, since all the VMs are idle after they boot, so the total system utilization should be low regardless of the number of VMs. Another result is also apparent from this test: as the size of the VM decreases, the creation time contributes a larger and larger fraction of the time that it takes from starting the VM creation to its availability: with lightweight VMs, the instantiation of new VMs becomes the main bottleneck.


To understand VM creation overheads, we instrumented Xen's xl command-line tool and its libxl library, and categorized the work done into several categories:

- Parsing the configuration le that describes the VM (kernel image, virtual network/block devices, etc.).
- Interacting with the hypervisor to, for example, reserve and prepare the memory for the new guest, create the virtual CPUs, and so on.
- Writing information about the new guest in the Xen- Store, such as its name.
- Creating and configuring the virtual devices.
- Parsing the kernel image and loading it into memory. Internal information and state keeping of the toolstack that do not t into any of the above categories.

Figure 5 shows the creation overhead categorized in this way. It is immediately apparent that there are two main contributors to the VM creation overhead: the XenStore interaction and the device creation, to the point of negligibility of all other categories.3 Device creation dominates the guest instantiation times when the number of currently running guests is low; its overhead stays roughly constant when we keep adding virtual machines

However, the time spent on XenStore interactions increases superlinearly, for several reasons. The protocol used by the XenStore is quite expensive, where each operation requires sending a message and receiving an acknowledgment, each triggering a software interrupt: a single read or write thus triggers at least two, and most often four, software interrupts and multiple domain changes between the guest, hypervisor and Dom0 kernel and userspace; as we increase the number of VMs, so does the load on this protocol. Secondly, writing certain types of information, such as unique guest names, incurs overhead linear with the number of machines because the XenStore compares the new entry against the names of all other already-running guests before accepting the new guest's name. Finally, some information, such as device initialization, requires writing data in multiple Xen- Store records where atomicity is ensured via transactions. As the load increases, XenStore interactions belonging to different transactions frequently overlap, resulting in failed transactions that need to be retried.

Finally, it is worth noting that the spikes on the graph are due to the fact that the XenStore logs every access to log les (20 of them), and rotates them when a certain maximum number of lines is reached (13,215 lines by default); the spikes happen when this rotation takes place. While disabling this logging would remove the spikes, it would not help in improving the overall creation times, as we verified in further experiments not shown here.

2.5 LightVM

Our target is to achieve VM boot times comparable to process startup times. Xen has not been engineered for this objective, as the results in the previous section show, and the root of these

problems is deeper than just inefficient code. For instance, one fundamental problem with the XenStore is its centralized, filesystem-like API which is simply too slow for use during VM creation and boot, requiring tens of interrupts and privilege domain crossings. Contrast this to the fork system call which requires a single software interrupt a single user-kernel crossing. To achieve millisecond boot times we need much more than simple optimizations to the existing Xen codebase.

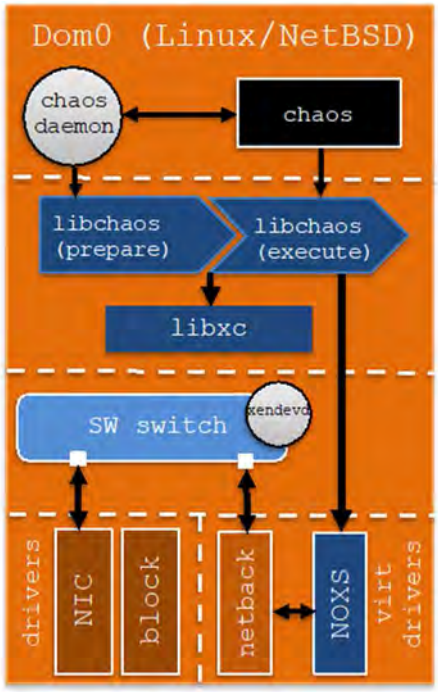


Figure 6: LightVM architecture showing noxs, xl's re- placement (chaos), the split toolstack and accompanying daemon, and xendevd in charge of quickly adding virtual interfaces to the software switch

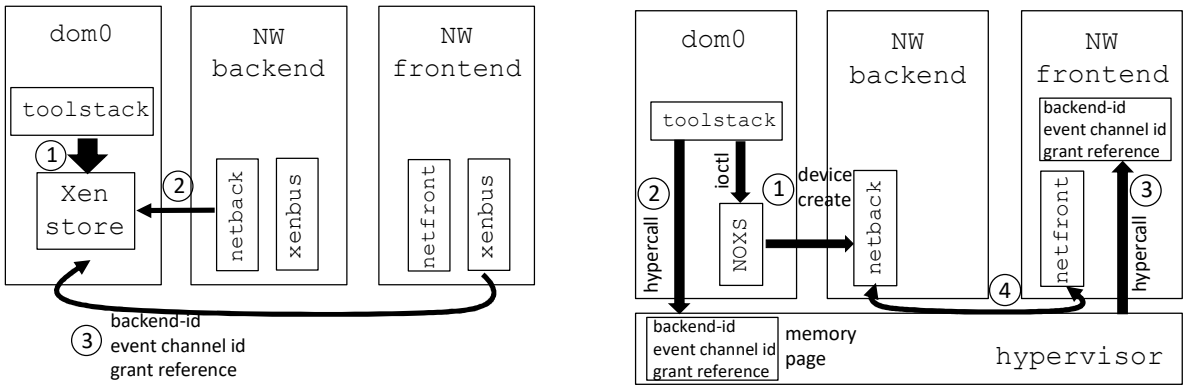
To this end, we introduce LightVM, a complete re-design of the basic Xen control plane optimized to provide light- weight virtualization. The architecture of LightVM is shown in Figure 6. LightVM does not use the XenStore for VM creation or boot anymore, using instead a lean driver called noxs that addresses the scalability problems of the XenStore by enabling direct communication between the frontend and backend drivers via shared memory instead of relaying messages through the XenStore. Because noxs does not rely on a message passing protocol but rather on shared pages mapped in the guest's address space, reducing the number of software interrupts and domain crossings needed for VM operations (create/save/resume/migrate/destroy).

LightVM provides a split toolstack that separates VM creation functionality into a prepare and an execute phase, reducing the amount of work to be done on VM creation. We have also implemented chaos/libchaos, a new virtualization toolstack that is much leaner than the standard xl/libxl, in addition to a small daemon called xendevd that quickly adds virtual interfaces to the software switch or handles block device images' setup. We cover each of these in detail in the next sections.

2.5.1 Noxs (no XenStore) and the Chaos Toolstack

The XenStore is crucial to the way Xen functions, with many xl commands making heavy use of it. By way of illustration, Figure 7a shows the process when creating a VM and its (virtual) network device. First, the toolstack writes an entry to the network back-end's directory, essentially announcing the existence of a new VM in need of a network device. Previous to that, the back-end placed a watch on that directory; the toolstack writing to this directory triggers the back-end to assign an event channel and other information (e.g., grant references, a mechanism for sharing memory be- tween guests) and to write it back to the XenStore (step 2 in the figure). Finally, when the VM boots up it contacts the XenStore to retrieve the information previously written by the network back-end (step 3). The above is a simplification: in actuality, the VM creation process alone can require interaction with over 30 XenStore entries, a problem that is exacerbated with increasing number of VMs and devices.

Is it possible to forego the use of the XenStore for operations such as creation, pause/unpause and migration? As it turns out, most of the necessary information about a VM is already kept by the hypervisor (e.g., the VM's id, but not the name, which is kept in the XenStore but is not needed during boot). The insight here is that the hypervisor already acts as a sort of centralized store, so we can extend its functionality to implement our noxs (no XenStore) mechanism.



(a) XenStore (b) noxs

Figure 7: Standard VM creation process in Xen using the XenStore versus our noxs implementation.

Specifically, we begin by replacing libxl and the corresponding xl command with a streamlined, thin library and command called libchaos and chaos, respectively (cf. Figure 6); these no longer make use of the XenStore and its accompanying libxs library.

In addition, we modify Xen's hypervisor to create a new, special device memory page for each new VM that we use to keep track of a VM's information about any devices, such as block and networking, that it may have. We also include a hypercall to write to and read from this memory page, and make


sure that, for security reasons, the page is shared read-only with guests, with only Dom0 allowed to request modifications.

When a chaos create command is issued, the toolstack first requests the creation of devices from the back-end(s) through an ioctl handled by the noxs Linux kernel module (step 1 in Figure 7b).4 The back-end then returns the details about the communication channel for the front-end. Second, the toolstack calls the new hypercall asking the hypervisor to add these details to the device page (step 2).

When the VM boots, instead of contacting the XenStore, it will ask the hypervisor for the address of the device page and will map the page into its address space using hyper- calls (step 3 in the figure); this requires modifications to the guest's operating system, which we have done for Linux and Mini-OS. The guest will then use the information in the page to initiate communication with the back-end(s) by mapping the grant and binding to the event channel (step 4). At this stage, the front and back-ends set up the device by exchanging information such as its state and its MAC address (for networking); this information was previously kept in the XenStore and is now stored in a device control page pointed to by the grant reference. Finally, front and back-ends notify each other of events through the event channel, which replaces the use of XenStore watches.

To support migration without a XenStore, we create a new pseudo-device called sysctl to handle power-related operations and implement it following Xen's split driver model, with a back-end driver (sysctlback) and a front-end (sysctlfront) one. These two drivers share a device page through which communication happens and an event channel.

With this in place, migration begins by chaos opening a TCP connection to a migration daemon running on the remote host and by sending the guest's configuration so that the daemon pre-creates the domain and creates the devices. Next, to suspend the guest, chaos issues an ioctl to the sysctl back-end, which will set a field in the shared page to denote that the shutdown reason is suspend, and triggers the event channel. The front-end will receive the request to shutdown, upon which the guest will save its internal state and unbind noxs-related event channels and device pages. Once the guest is suspended we rely on libxc code to send the guest data to the remote host.

2.5.2 Split Toolstack

In the previous section we showed that a large fraction of the overheads related to VM creation and other operations comes from the toolstack itself. Upon closer investigation, it turns out that a significant portion of the code that executes when, for instance, a VM create command is issued, does not actually need to run at VM creation time. This is because this code is common to all VMs, or at least common to all VMs with similar configurations. For example, the amount of memory may differ between VMs, but there will generally only be a small number of differing memory

configurations, similar to OpenStack's flavors. This means that VMs can be pre-executed and thus o-loaded from the creation process. To take advantage of this, we replace the standard Xen toolstack with the libchaos library and split it into two phases. The prepare phase (see Figure 8) is responsible for functionality common to all VMs such as having the hypervisor generate an ID and other management information and allocating CPU resources to the VM. We offload this functionality to the chaos daemon, which generates a number of VM shells and places them in a pool. The daemon ensures that there is always a certain (configurable) number of shells available in the system.

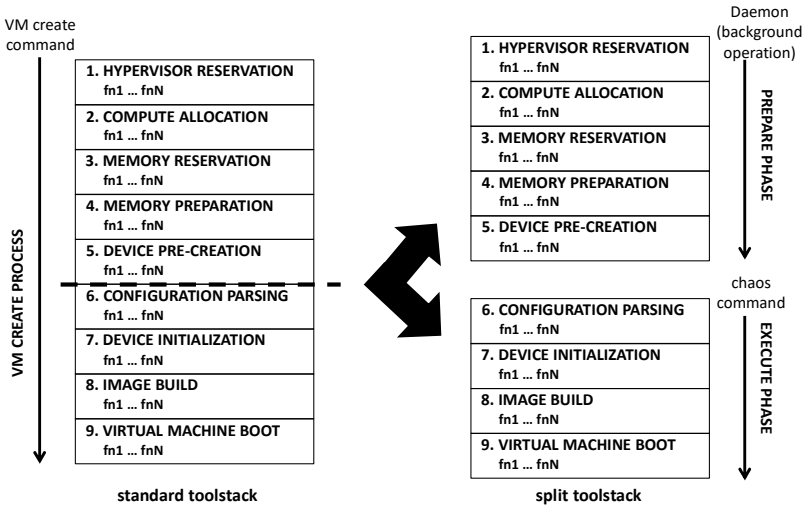


Figure 8: Toolstack split between functionality be- longing to the prepare phase, carried out periodically by the chaos daemon, and an execute phase, directly called by chaos when a command is issued.

The execute phase then begins when a VM creation command is issued. First, chaos parses the configuration le for the VM to be created. It then contacts the daemon and asks for a shell fitting the VM requirements, which is then removed from the pool. On this shell, the remaining VM- specific operations, such as loading the kernel image into memory and finalizing the device initialization, are executed to create the VM, which is then booted.

2.5.3 Replacing the Hotplug Script: xendevd

The creation of a virtual device by the driver domain usually requires some mechanism to setup the device in user-space (e.g., by adding a vif to the bridge). With standard Xen this process is done either by xl, calling bash scripts that take care of the necessary initialization or by udevd, calling the same scripts when the backend triggers the udev event. The script that is executed is usually user-configured, giving great flexibility to implement different scenarios. However launching and executing

bash scripts is a slow process taking tens of milliseconds, considerably slowing down the boot process. To work around this, we implemented this mechanism as a binary daemon called xendevd that listens for udev events from the backends and executes a pre-defined setup without forking or bash scripts, reducing setup time.

2.6 Evaluation

In this section we present a performance evaluation of LightVM, including comparisons to standard Xen and, where applicable, Docker containers. We use two x86 machines in our tests: one with an Intel Xeon E5-1630 v3 CPU at 3.7 GHz (4 cores) and 128GB of DDR4 RAM, and another one consisting of four AMD Opteron 6376 CPUs at 2.3 GHz (with 16 cores each) and 128GB of DDR3 RAM. Both servers run Xen 4.8. We use a number of different guests: (1) three Mini-OS-based unikernels, including noop, the daytime unikernel, and one based on Micropython [31] which we call Minipython; (2) a Tinyx noop image (no apps installed) and a Tinyx image with Micropython; and (3) a Debian VM. For container experiments we use Docker version 1.13.

2.6.1 Instantiation Times

We want to measure how long it takes to create and boot a virtual machine, how that scales as the number of running VMs on the system increases, and how both of these compare to containers. We further want to understand how the LightVM mechanisms affect these times.

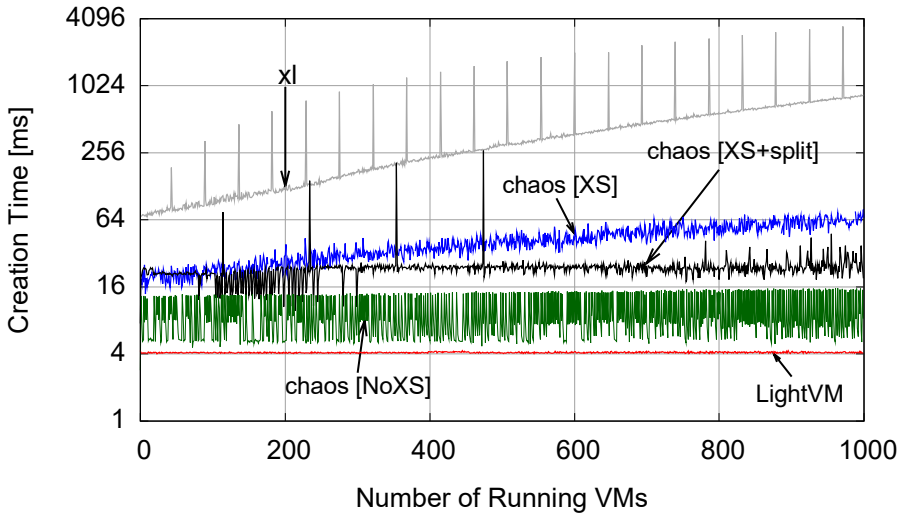


Figure 9: Creation times for up to 1,000 instances of the daytime unikernel for all combinations of LightVM's mechanisms. "xl" denotes standard Xen with no optimizations.

For the first test, we boot up to 1,000 daytime unikernels, and we measure the time it takes for the n'th unikernel to be created. We repeat this experiment with all combinations of the LightVM mechanisms: chaos + XenStore, chaos + noxs, chaos + split toolstack and chaos + noxs + split toolstack; we also include measurements when running out-of-the-box Xen (labeled "xl"). We run the tests on the 4-core machine, with one core assigned to Dom0 and the remaining three cores assigned to the VMs in a round-robin fashion.

The results are shown in Figure 9. Out of the box, Xen (the "xl" curve) has creation times of about 100ms for the first VM, scaling rather poorly up to a maximum of just under 1 second for the 1000th VM. In addition, the curve shows spikes at regular intervals as a result of the log rotation issue mentioned earlier in the paper. Replacing xl with chaos results in a noticeable improvement, with creation times ranging now from roughly 15 to 80ms. Adding the split toolstack mechanism to the equation improves scalability, showing a maximum of about 25ms for the last VMs. Removing the XenStore (chaos + noxs curve) provides great scalability, essentially yielding low creation times in the range of 8-15 ms for the last VMs. Finally, we obtain the best results with all of the optimizations (chaos + noxs + split toolstack) turned on: boot times as low as 4ms going up to just 4.1ms for the 1,000th VM. As a final point of reference, using a noop unikernel with no devices and all optimizations results in a minimum boot time of 2.3ms.

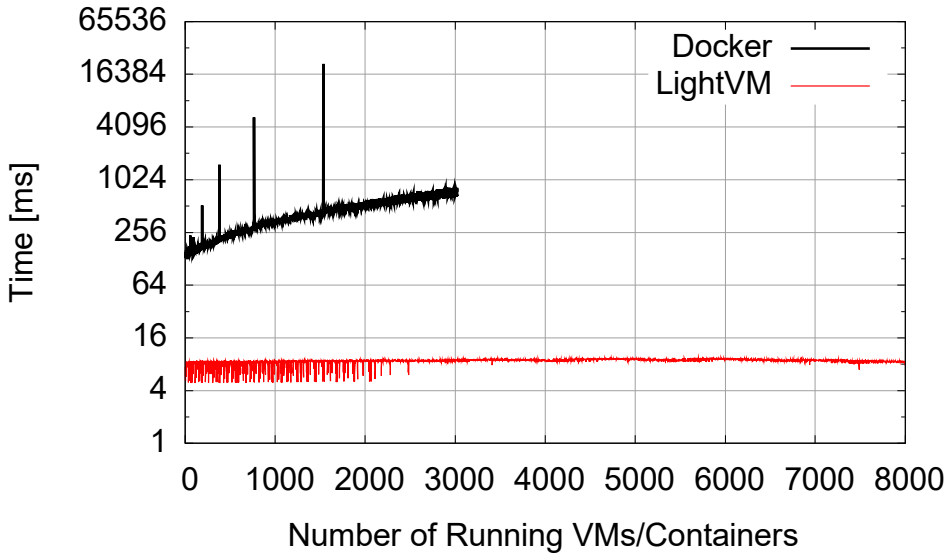


Figure 10: LightVM boot times on a 64-core machine versus Docker containers.

Next, we test LightVM against Docker containers when using even larger numbers of instances (see Figure 10). To measure this, we used the 64-core AMD machine, assigning 4 cores to Dom0 and the remaining 60 to the VMs in a round-robin fashion; we used the noop unikernel for the guests themselves. As before, the best results come when using chaos + noxs + split toolstack, which shows

good scalability with increasing number of VMs, up to 8,000 of them in this case. Docker containers start at about 150ms and ramp up to about 1 second for the 3,000th container. The spikes in that curve coincide with large jumps in memory consumption, and we stop at about 3,000 because after that the next large memory allocation consumes all available memory and the system becomes unresponsive.

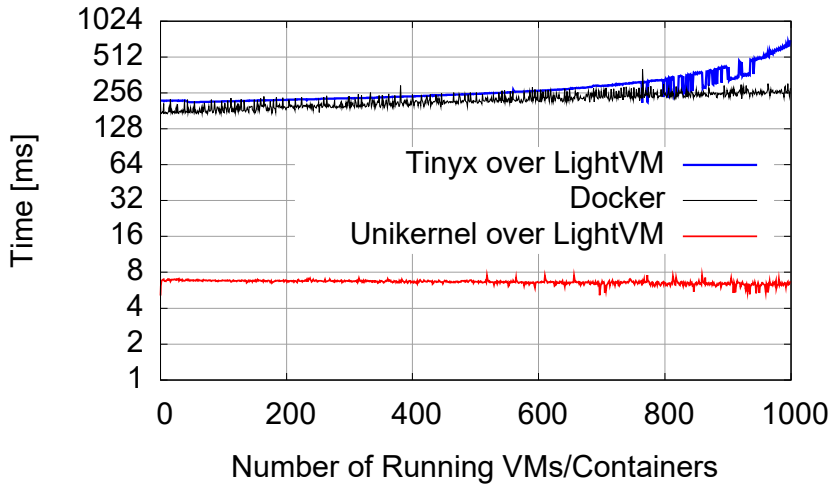


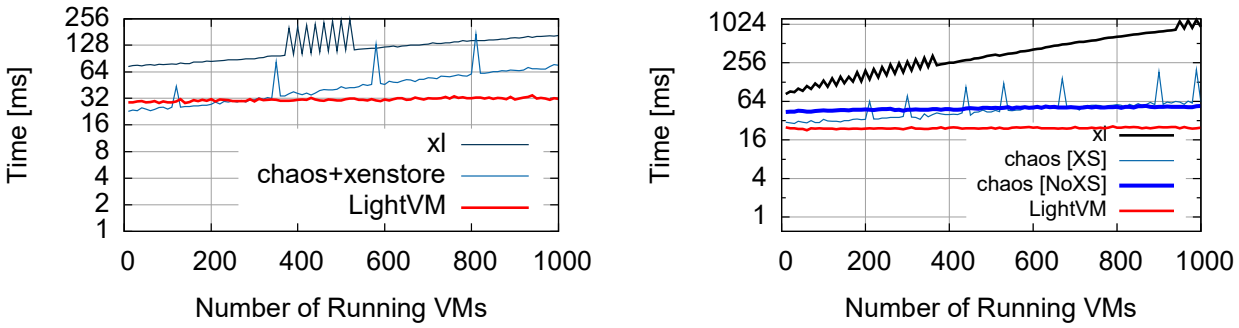
Figure 11: Boot times for unikernel and Tinyx guests versus Docker containers.

For the final test we show how the boot times of a Tinyx guest compare to those of a unikernel, and we further plot a Docker container curve for comparison (see Figure 11). As expected, the unikernel performs best, but worthy of note is the fact that Tinyx, a Linux-based VM, performs rather close to Docker containers up to roughly 750 VMs (250 per core on our test machine). The increase in boot times as the number of VMs per core increases is due to the fact that even an idle, minimal Linux distribution such as Tinyx runs occasional background tasks. As the number of VMs per core increases, contention for CPU time increases, increasing the boot time of each VM. In contrast, idling Docker containers or unikernels do not run such background tasks, leading to no noticeable increase in boot times.

2.6.2 Checkpointing and Migration

In the next set of experiments we use the 4-core machine and the daytime unikernel to test save/restore (i.e., checkpointing) and migration times. In all tests we use a RAM disk for the filesystem so that the results are not skewed by hard drive access speeds. We assign two cores to Dom0 and the remaining two to the VMs in a round-robin fashion. For checkpointing, the experimental procedure is as follows. At every run of the test we start 10 guests and randomly pick 10 guests to be checkpointed; for instance, in the first run all 10 guests created are checkpointed, in the second one

20 guests exist out of which 10 are checkpointed, and so on up to a total of 1,000 guests. This is to show how quick checkpointing is when the system is already running N numbers of guests.



(a) Save (b) Restore

Figure 12: Checkpointing times using the daytime unikernel

The results are shown in Figure 12a and Figure 12b, split between save and restore times, respectively. The figures show that LightVM can save a VM in around 30ms and restore it in 20ms, regardless of the number of running guests, while standard Xen needs 128ms and 550ms respectively. For migration tests we carry out the same procedure of starting 10 guests and randomly choosing 10 to migrate. Once the 10 guests are migrated, we replace the migrated guests with 10 new guests to make sure that the right number of guests are running on the source host for the next round of the test (e.g., if we have 100 guests, we migrate 10, leaving 90 on the source host; we then create 10, leaving 100 at the source host so that it is ready for the next round).

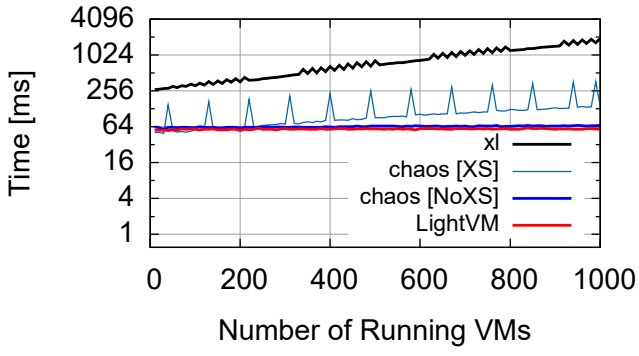


Figure 13: Migration times for the daytime unikernel.

We plot migration times for the daytime unikernel in Figure 13. As in previous graphs, turning all optimizations on (chaos, noxs and split toolstack) yields the best results, with migration times of about 60ms irrespective of how many VMs are currently running on the host. For low number of VMs

the chaos + XenStore slightly outperforms LightVM: this is due to device destruction times in noxs which we have not yet optimized and remain as future work.

2.6.3 Memory Footprint

One of the advantages of containers is that, since they use a common kernel, their memory usage stays low as their numbers increase. This is in contrast to VMs, where each instance is a full replica of the operating system and filesystem. In our next experiment we try to see if small VMs can get close to the memory scalability that containers provide. For the test we use the 4-core machine and allocate a single core to Dom0 and the remaining 3 to the VMs as before. We use three types of guests: a Minipython unikernel, a Tinyx VM with Micropython installed, and a Debian VM also with Micropython installed. For comparison purposes we also conduct tests with a Docker/Micropython container and a Micropython process.

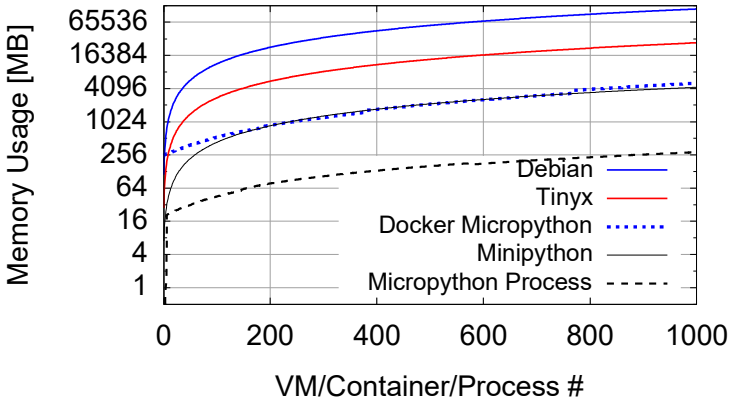


Figure 14: Scalability of VM memory usage for different VMs, for containers and for processes

Generally, the results (Figure 14) show that the unikernel's memory usage is fairly close to that of Docker containers. The Tinyx curve is higher, since multiple copies of the Linux kernel are running; however, the additional memory consumption is not dramatic: for 1,000 guests, the system uses about 27GB versus 5GB for Docker. This 22GB difference is small for current server memory capacity (100s of GBs or higher) and memory prices. Debian consumes about 114GB when running 1,000 VMs (assuming 111MB per VM, the minimum needed for them to run).

2.6.4 CPU Usage

For the final test of the section we take a look at CPU usage when using a noop unikernel, Tinyx and a Debian VM, and plot these against usage for Docker. For the VM measurements we use iostat to

get Dom0's CPU usage and xentop to get the guests' utilizations. As shown in Figure 15, Docker containers have the lowest utilization although the unikernel is only a fraction of a percentage point higher. Tinyx also fares relatively well, reaching a maximum utilization of about 1% when running 1,000 guests. The Debian VM scales more poorly, reaching about 25% for 1,000 VMs: this is because each Debian VM runs a number of services out of the box that, taken together, results in fairly high CPU utilization. In all, the graph shows that CPU utilizations for VMs can be roughly on par with that of containers, as long as the VMs are trimmed down to include only the functionality crucial for the target application.

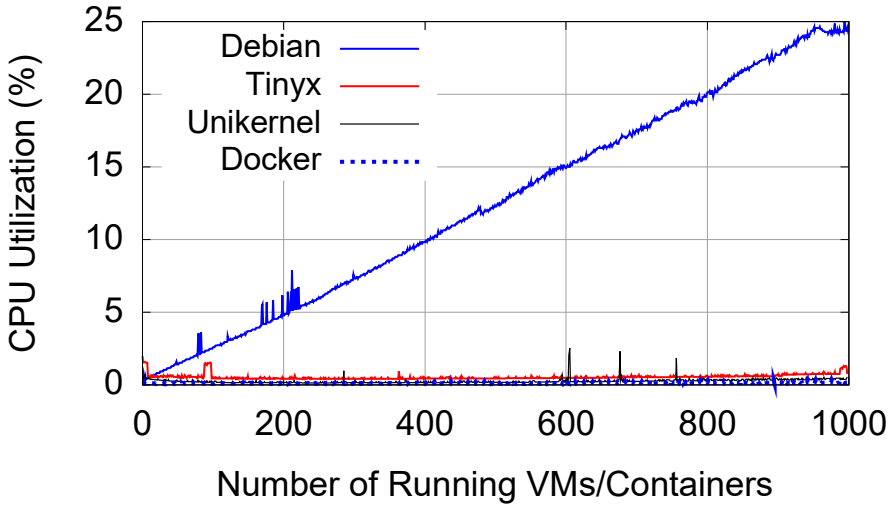


Figure 15: CPU usage for a unikernel, Tinyx, a Debian VM and Docker.

2.7 Use Cases

We now explore scenarios where lightweight virtualization can bring a tangible benefit over the status quo. In all the following scenarios, using containers would help performance but weaken isolation, while using full-blown VMs would pro- vide the same isolation as lightweight VMs, but with poorer performance.

2.7.1 Personal Firewalls

The number of attacks targeting mobile phones is increasing constantly [30]. Running up-to-date firewalls and intrusion detection/prevention on the mobile phone is difficult, so one option is to scrub the traffic in the cloud instead. Ideally, each mobile user should be able to run a personal firewall that is on-path of the traffic to avoid latency inflation.

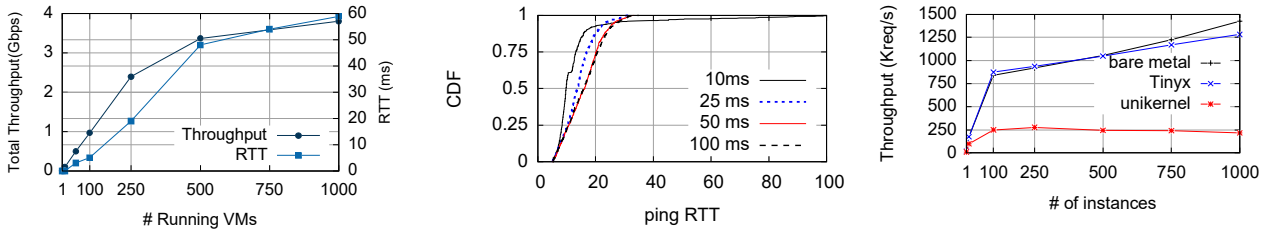
Recently, mobile operators have started working on mobile- edge computing (MEC) [16], which aims to run processing as close as possible to mobile users and, to this end, deploys servers co-located with mobile gateways situated at or near the cellular base stations (or cells).

The MEC is an ideal place to instantiate personal firewalls for mobile users. The difficulty is that the amount of deployed hardware at any single cell is very limited (one or a few machines), while the number of active users in the cell is on the order of a few thousand. Moreover, users enter and leave the cell continuously, so it is critical to be able to instantiate, terminate and migrate personal firewalls quickly and cheaply, following the user through the mobile network.

Using full-blown Linux VMs is not feasible because of their large boot times (a few seconds) and their large image sizes (GBs) which would severely increase migration duration and network utilization. Using containers, on the other hand, is tricky because malicious users could try to subvert the whole MEC machine and would be able to read and tamper with other users' traffic.

Instead, we rely on ClickOS, a unikernel specialized for network processing [29] running a simple firewall configuration we have created. The resulting VM image is 1.7MB in size and the VM needs just 8MB of memory to run; we can run as many as 8000 such firewalls on our 64-core AMD machine, and booting one instance takes about 10ms. Migrating a ClickOS VM over a 1Gbps, 10ms link takes just 150ms.

We want to see if the ClickOS VMs can actually do useful work when so many of them are active simultaneously. We start 1000 VMs running firewalls for 1000 emulated mobile clients, and then run an increasing number of iperf instances, each instance representing one client and being serviced by a dedicated VM. We limit each client's throughput to 10Mbps to mimic typical 4G speeds in busy cells. We measure total throughput as well added latency. For the latter, we have one client run ping instead of iperf.



(a) Running personal firewall for 1000 users using ClickOS

(b) Just-in-time instantiation of VMs to service mobile clients

(c) TLS termination throughput for up to 1000 end points

Figure 16: LightVM use cases.

The results, run on a server with an Intel Xeon E5-2690 v4 2.6 GHz processor (14 cores) and 64GB of RAM, are shown in Figure 16a. The cumulative throughput grows linearly until 2.5Gbps (250 clients), each client getting 10Mbps. After that, heavy CPU contention curbs the throughput increase: the


average per-user throughput is 6.5Mbps when there are 500 active users, and 4Mbps with 1000 active users. The per-packet added latency is negligible with few active users (tens), but increases to 60ms when 1000 users are active; this is to be expected since the Xen scheduler will effectively round-robin through the VMs. We note that VMs servicing long flows care less about latency, and a better scheduler could prioritize VMs servicing few packets (like our ping VM); this subject is worth future investigation.

To put the performance in perspective, we note that the maximum theoretical download throughput of LTE-advanced is just 3.3Gbps (per cell sector), implying that a single ma- chine running LightVM would be able to run personalized firewalls for all the users in the cell without becoming a performance bottleneck.

2.7.2 Just-in-Time Instantiation

Our second use-case also targets mobile edge computing: we are interested in dynamically offloading work from the mobile to the edge as proposed by [28]; the most important metrics are responsiveness and the ability to offer the service to as many mobile devices as possible. We implemented a dummy service that boots a VM whenever it receives a packet from a new client, and keeps the VM running as long as the client is actively sending packets; after 2s of inactivity we tear down the VM. To measure the worst-case client perceived latency, we have each client send a single ping request, and have the newly booted VM reply to pings.

We use open-loop client arrivals with different intensities and plot the CDFs of ping times in Figure 16b. The different curves correspond to varying client inter-arrival rates: with one new client every 25 ms, the client-measured latency is 13ms in the median and 20ms at the 90%. With one new client every 10 ms, the RTTs improve up to the point that our Linux bridge is overloaded and starts dropping packets (mostly ARP packets), hence some pings time out and there is a long tail for the client-perceived latency.

2.7.3 High Density TLS Termination

The Snowden leaks have revealed the full extent of state-level surveillance, and this has pushed most content providers to switch to TLS (i.e., HTTPS) to the point where 70% of Internet traffic is now encrypted [39]. TLS, however, requires at least one additional round-trip time to setup, increasing page load times significantly if the path RTT is large. The preferred solution to reduce this effect is to terminate TLS as close to the client as possible using a content-distribution network and then serve the content from the local cache or fetch it from the server over a long-term encrypted tunnel.

Even small content providers have started relying on CDNs, which now must serve a larger number of customers on the same geographically-distributed server deployments. TLS termination needs the


long term secret key of the content provider, requiring strong isolation between different content-providers' HTTPS proxies; simply running these in containers is unacceptable, while running full Linux VMs is too heavyweight.

We have built two lightweight VMs for TLS termination: one is based on Minipython and the other one is based on Tinyx. Our target is to support as many TLS proxies as possible on a single machine, so we rely on axtls [12], a TLS library for embedded systems optimized for size. The unikernel relies on the lwip networking stack, boots in 6ms and uses 16MB of RAM at runtime. The Tinyx machine uses 40MB of RAM and boots in 190ms.

To understand whether the CDN provider can efficiently support many simultaneous customers on the same box, we have N apache bench clients continuously requesting the same empty le over HTTPS from N virtual machines. We measure the aggregate throughput on the 14-core machine and plot it in Figure 16c (bare metal means a Linux process, i.e., no hypervisor). The graph shows that adding more VMs increases total throughput; this is because more VMs fully utilize all CPUs to perform public-key operations, masking protocol overheads. Tinyx's performance is very similar to that of running processes on a bare-metal Linux distribution: around 1400 requests per second are serviced. This number is low because we use 1024-bit RSA keys instead of more efficient variants such as ECDHE. Finally, note that the unikernel only achieves a fifth of the throughput of Tinyx; this is mostly due to the inefficient lwip stack. With appropriate engineering, the stack can be fixed, however reaching the maturity of the Linux kernel TCP stack is a tall order.

These results highlight the tradeoffs one must navigate when aiming for lightweight virtualization: either use Tinyx or other minimalistic Linux-based VMs that are inherently slower to boot (hundreds of ms) and more memory hungry, or invest considerable engineering effort to develop minimal unikernels that achieve millisecond-level boot times and allow for massive consolidation.

2.7.4 Lightweight Compute Service

On-demand compute services such as Amazon's Lambda or blockchain-backed distributed ledgers are increasing in popularity. Such services include data, image, or video aggregations and conversions, or cryptographic hash computations, and perform calculations that often do not last more than a few seconds. Further, there is no need to keep state between independent calculations which means that the service can be simply destroyed after finishing the calculation. Nevertheless, compute services for different tenants need strong isolation to reduce sensitive information leaks.

Lightweight VMs are a perfect match for such lightweight computation services. For this use case we rely on the Mini- python unikernel we created which runs computations written in Python, similar to Amazon's Lambda. The unikernel uses the lightweight MicroPython interpreter and also links a networking stack. In addition, we have implemented a daemon in Dom0 that receives compute

service requests (in the form of python programs) and spawns a VM to run the program. When the program finishes the VM shuts down. We ran experiments on our four-core machine. All compute services calculated an approximation of e that takes approximately 0.8 seconds. The domains were spawned on three of the four cores (with the fourth exclusively used by Dom0).

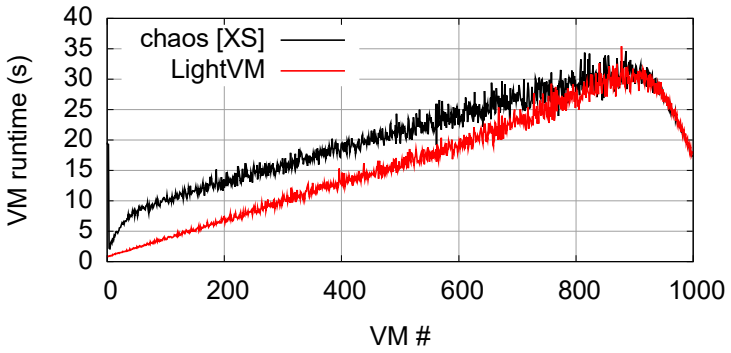


Figure 17: Lightweight compute function service time on an overloaded machine (Minipython unikernels)

We generate one thousand compute requests in an open loop with inter-arrival times of 250ms. This is faster than our machine can cope (266ms inter-arrivals lead to full utilization), slowly increasing load on the system. Creation times are not strongly affected by the increasing load, since Dom0 had its own dedicated core. Creation times for noxs slowly increase from approximately 2.8 ms to approximately 3.5 ms. Using the split toolstack and its pre-created domains takes a nearly constant 1.3 ms regardless of the number of already- created domains. Figure 17 shows the time it takes for the nth compute request to be serviced in this overloaded sys- tem, and Figure 18 shows the number of active VMs as a function of time. Notice how our optimizations, in particular not using the XenStore, improve the completion times by a factor of 5 when the system is slightly overloaded (100-200 backlogged VMs); here the work reduction provided by noxs allows other VMs to do useful work instead, reducing the number of backlogged VMs.

2.8 Related Work

A number of OS-level virtualization technologies exist and are widely deployed, including Docker, LXC, FreeBSD jails and Linux-VServer, among others [6, 13, 25, 42]. In terms of high density, the work in [17] shows how to run 10,000 Docker containers on a single server. Zhang et al. implement network functions using Docker containers and can boot up to 80K of them [51]. In our work, we make the case for lightweight virtualization on top of a hypervisor, providing strong isolation while retaining the attractive properties commonly found in containers.


A number of works have looked into optimizing hypervisors such as Xen, KVM and VMWare [3, 20, 47] to reduce their overheads in terms of boot times and other metrics. For example, Intel Clear Containers [45] (ICC) has similarities to our work but a different goal. ICC tries to run containers within VMs with the explicit aim of keeping compatibility with existing frameworks (Docker, rkt); this compatibility results in overheads. LightVM optimizes both the virtualization system and guests, achieving performance similar to or better than containers without sacrificing isolation. We have also showed that it is possible to make automated build tools to minimize the effort needed to make such specialized guests (i.e., with Tinyx). ICC also optimizes parts of the virtualization system (e.g., the toolstack), but does not provide other features such as the split toolstack and uses larger guests: an ICC guest is 70MB and boots in 500ms [19] as opposed to a Tinyx one which is about 10MB and boots in about 300ms. ukvm [50] implements a specialized unikernel monitor on top of KVM and uses MirageOS unikernels to achieve 10 ms boot times (the main metric the work focuses on). Jitsu [26] optimizes parts of Xen to implement just-in-time instantiation of network services by accelerating connection start-up times. Earlier work [48] implemented JIT instantiation of honeypots through the use of image cloning; unlike the work there, we do not require the VMs on the system to run the same application in order to achieve scalability. The work in [36] optimizes xl toolstack overheads, but their use of Linux VMs results in boot times in the hundreds of milliseconds or seconds range. LightVM aims to provide container- like dynamics; as far as we know, this is the first proposal to simultaneously provide small boot, suspend/resume and migration times (sometimes an order of magnitude smaller than previous work), high density and low per-VM memory footprints.

Beyond containers and virtual machines, other works have proposed the use of minimalistic kernels or hypervisors to provide lightweight virtualization. Exokernel [8] is a minimalistic operating system kernel that provides applications with the ability to directly manage physical resources. NOVA [44] is a microhypervisor consisting of a thin virtualization layer and thus aimed at reducing the attack surface (NOVA's TCB is about 36K LoC, compared to for instance 11.4K for Xen's ARM port). The Denali isolation kernel is able to boot 10K VMs but does not support legacy OSes and has limited device support [49]. The work in [40] proposes the implementation of cloudlets to quickly offload services from mobile devices to virtual machines running in a cluster or data center, although the paper reports VM boot times in the 60-90 seconds range. The Embassies project [7, 15] and Drawbridge [33] make a similar case than us by providing strong isolation through picoprocesses that have a narrow interface (the way VMs do), though the target there was running isolated user-space applications or web client apps. Finally, unikernels [27] have seen significant research interest; prior work includes Mirage [27], ClickOS [29], Erlang on Xen [9] and OSv [21] to name just a few. Our work does not focus on unikernels, but rather leverages them to be able to separate the effects coming from the virtual machine from those of the underlying virtualization platform. For example, they allow us to reach high density numbers without having to resort to overly expensive servers.

2.9 Discussion and Open Issues

Memory sharing: LightVM does not use page sharing be- tween VMs, assuming the worst-case scenario where all pages are different. One avenue of optimization is to use memory de-duplication (as proposed by SnowFlock [24]) to reduce the overall memory footprint; unfortunately this requires non-negligible changes to the virtualization system.

Generality: While LightVM is based on Xen, most of its components can be extended to other virtualization plat- forms such as KVM. This includes (1) the optimized toolstack, where work such as ukvm [50] provides a lean toolstack for KVM (among other things); (2) the pre-creation of guests, which is independent of the underlying hypervisor technology; and (3) the use of specialized OSes and unikernels, several of which already exist for non-Xen hypervisors (e.g., the Solo5 unikernel [18], rump kernels [38], OSv [21]). The one feature that is Xen-specific is the XenStore, though KVM keeps similar information in the Linux kernel (process information) and in the QEMU process (device information).

Usability and portability: Despite its compelling performance, LightVM is still not as easy to use as containers. Container users can rely on a large ecosystem of tools and support to run unmodified existing applications. LightVM exposes a clear trade-off between performance and portability/usability. Unikernels provide the best performance, but require non-negligible development time and manual tweaking to get an image to compile against a target application. Further, debugging and extracting the best performance out of them is not always trivial since they do not come with the rich set of tools that OSes such as Linux have. At the other extreme, VMs based on general-purpose OSes such as Linux require no porting and can make use of existing management tools, but their large memory footprints and high boot times, among other issues, have at least partly resulted in the widespread adoption of containers (and their security problems).

In designing and implementing the Tinyx build system we tried to take a first step towards solving the problem: Tinyx provides better performance than a standard Debian distribution without requiring any application porting. However, Tinyx is still a compromise: we sacrifice some performance with respect to unikernels in order to keep the ecosystem and existing application support.

The ultimate goal is to be able to automatically build custom-OSes targeting a single application. For instance, the Rump Kernels project [38] builds “unikernels”, relying on large portions of NetBSD to support existing applications. This is not quite what we would need since the performance and size of the resulting images are not in the same order of magnitude as LightVM. Part of the solution would have to decompose an existing OS into ne-granularity modules, automatically analyze an application’s dependencies, and choose which is the minimum set of modules needed for the unikernel to compile. This area of research is future work.

2.10 Conclusions

We have presented LightVM, a complete redesign of Xen's toolstack optimized for performance that can boot a minimalistic VM in as little as 2.3ms, comparable to the fork/exec implementation in Linux (1ms). Moreover, LightVM has almost constant creation and boot times regardless of the number of running VMs; this is in contrast to the current toolstack that can take as much as 1s to create a VM when the system is loaded. To achieve such performance LightVM foregoes Xen's centralized toolstack architecture based on the Xen-Store in favor of a distributed implementation we call noxs, along with a reimplementation of the toolstack.

The use cases we presented show that there is a real need for lightweight virtualization, and that it is possible to simultaneously achieve both good isolation and performance on par or better than containers. However, there is a development price to be paid: unikernels offer best performance but require significant engineering effort which is useful for highly popular apps (such as TLS termination) but likely too much for many other applications. Instead, we have pro- posed Tinyx as a midway point: creating Tinyx images is streamlined and (almost) as simple as creating containers, and performance is on par with that of Docker containers.

2.11 References for section 2

[1] Amazon Web Services. Amazon EC2 Container Service. https://aws.amazon.com/ecs/.

[2] Amazon Web Services. AWS Lambda - Serverless Compute. https://aws.amazon.com/lambda.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Wareld. 2003. Xen and the Art of Virtualization. SIGOPS Oper. Syst. Rev. 37, 5 (Oct. 2003), 164–177. https://doi.org/10.1145/1165389.945462

[4] J. Clark. Google: "EVERYTHING at Google runs in a container". http://www.theregister.co.uk/2014/05/23/google_ containerization_two_billion/.

[5] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Wareld. 2011. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervi- sor. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11). ACM, New York, NY, USA, 189–202. https://doi.org/10.1145/2043556.2043575

[6] Docker, The Docker Containerization Platform. https://www. docker.com/.

[7] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. 2008. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In Proceedings of the 8th USENIX Conference on

Operating Systems Design and Implementation (OSDI'08). USENIX Association, Berkeley, CA, USA, 339–354. http://dl.acm.org/citation.cfm?id=1855741.1855765

[8] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Man- agement. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95). ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[9] Erlang on Xen 2012. Erlang on Xen. http://erlangonxen.org/. (July 2012).

[10] Google Cloud Platform The Google Cloud Platform Container Engine. https://cloud.google.com/container-engine.

[11] A. Grattaori. Understanding and Hardening Linux Contain-ers. https://www.nccgroup.trust/us/our-research/understanding-and- hardening-linux-containers/.

[12] Cameron Hamilton-Rich. axTLS Embedded SSL. http://axtls. sourceforge.net.

[13] Poul henning Kamp and Robert N. M. Watson. 2000. Jails: Conning the omnipotent root. In In Proc. 2nd Intl. SANE Conference.

[14] J. Hertz. Abusing Privileged and Unprivileged Linux Contain- ers. https://www.nccgroup.trust/uk/our-research/abusing-privileged- and-unprivileged-linux-containers/.

[15] Jon Howell, Bryan Parno, and John R. Douceur. 2013. Embassies: Radically Refactoring the Web. In Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). USENIX, Lombard, IL, 529–545. https://www.usenix.org/conference/ nsdi13/technical-sessions/presentation/howell

[16] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. 2015. Mobile Edge Computing - A key technology towards 5G. ETSI White Paper No. 11, First edition (2015).

[17] IBM. [n. d.]. Docker at insane scale on IBM Power Sys- tems. https://www.ibm.com/blogs/bluemix/2015/11/docker-insane- scale-on-ibm-power-systems.

[18] IBM developerWorks Open Solo5 Unikernel. https://developer. ibm.com/open/openprojects/solo5-unikernel/.

[19] Intel. Intel Clear Containers: A Breakthrough Combination of Speed and Workload Isolation. https://clearlinux.org/sites/default/les/vmscontainers_wp_v5.pdf.

[20] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In In Proc. 2007 Ottawa Linux Symposium (OLS '07).

[21] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating Sys- tem for Virtual Machines. In Proceedings of

the 2014 USENIX Annual Technical Conference (USENIX ATC ’14). USENIX Association, Philadel- phia, PA, 61–72. https://www.usenix.org/conference/atc14/technical- sessions/presentation/kivity

[22] E. Kovacs. Docker Fixes Vulnerabilities, Shares Plans For Making Platform Safer. http://www.securityweek.com/docker-xes- vulnerabilities-shares-plans-making-platform-safer.

[23] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Fe- lipe Huici. 2017. Unikernels Everywhere: The Case for Elastic CDNs. In Proceedings of the 13th ACM SIGPLAN/SIGOPS International Confer- ence on Virtual Execution Environments (VEE ’17). ACM, New York, NY,USA, 15–29. https://doi.org/10.1145/3050748.3050757

[24] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. 2009. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys ’09). ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1519065.1519067

[25] LinuxContainers.org LinuxContainers.org. https:// linuxcontainers.org.

[26] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. 2015. Jitsu: Just-In-Time Summoning of Unikernels. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’15). USENIX Association, Oakland, CA, 559–573. https://www.usenix.org/conference/ nsdi15/technical-sessions/presentation/madhavapeddy

[27] Anil Madhavapeddy and David J. Scott. 2013. Unikernels: Rise of the Virtual Library Operating System. Queue 11, 11, Article 30 (Dec. 2013), 15 pages. https://doi.org/10.1145/2557963.2566628

[28] Y. Mao, J. Zhang, and K. B. Letaief. 2016. Dynamic Computation Offloading for Mobile-Edge Computing With Energy Harvesting Devices. IEEE Journal on Selected Areas in Communications 34, 12 (Dec 2016), 3590–3605. https://doi.org/10.1109/JSAC.2016.2611964

[29] 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 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’14). USENIX As- sociation, Seattle, WA, 459–473. https://www.usenix.org/conference/ nsdi14/technical-sessions/presentation/martins

[30] McAee. 2016. Mobile Threat Report. https://www.mcafee.com/us/ resources/reports/rp-mobile-threat-report-2016.pdf. (2016).

[31] MicroPython MicroPython. https://micropython.org/.

[32] Microsoft. Azure Container Service. https://azure.microsoft. com/en-us/services/container-service/.

[33] Microsoft Research. Drawbridge. https://www.microsoft.com/ en-us/research/project/drawbridge/.

[34] minios Mini-OS. https://wiki.xenproject.org/wiki/Mini-OS.

[35] A. Mourat. [n. d.]. 5 security concerns when using Docker. https://www.oreilly.com/ideas/ve-security-concerns-when-using- docker.

[36] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Bar- balace, Daniel Hagimont, and Binoy Ravindran. 2017. Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor. In Proceedings of the 13th ACM SIGPLAN/SIGOPS Inter- national Conference on Virtual Execution Environments (VEE '17). ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3050748.3050758

[37] MAN page. Linux system calls list. http://man7.org/linux/man- pages/man2/syscalls.2.html.

[38] Rumpkernel.org Rump Kernels. http://rumpkernel.org/.

[39] Sandvine. Internet trac encryption. https://www.sandvine. com/trends/encryption.html.

[40] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. IEEE Pervasive Computing 8, 4 (Oct. 2009), 14–23. https://doi.org/10. 1109/MPRV.2009.82

[41] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making Middleboxes Someone Else's Problem: Network Processing As a Cloud Service. In Proceedings of the ACM SIGCOMM 2012 Conference on Computer Communication (SIGCOMM '12). ACM, New York, NY, USA, 13–24. https://doi.org/10. 1145/2342356.2342359

[42] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based Operating System Vir- tualization: A Scalable, High-performance Alternative to Hypervi- sors. SIGOPS Oper. Syst. Rev. 41, 3 (March 2007), 275–287. https://doi.org/10.1145/1272998.1273025

[43] S. Stabellini. Xen on ARM. http://www.slideshare.net/xen_ com_mgr/alsf13-stabellini.

[44] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A Microhypervisor- based Secure Virtualization Architecture. In Proceedings of the 5th European Conference on Computer Systems (EuroSys '10). ACM, New York, NY, USA, 209–222. https://doi.org/10.1145/1755913.1755935

[45] A. van de Ven. An introduction to Clear Containers. https://lwn.net/Articles/644675/.

[46] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. 2009. Server Workload Analysis for Power Minimization Using Consolidation. In Proceedings of the 2009

USENIX Annual Techni- cal Conference (USENIX ATC ’09). USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=1855807.1855835

[47] VMWare. vSphere ESXi Bare-Metal Hypervisor. http://www. vmware.com/products/esxi-and-esx.html.

[48] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Georey M. Voelker, and Stefan Savage. 2005. Scal- ability, Fidelity, and Containment in the Potemkin Virtual Honey- farm. SIGOPS Oper. Syst. Rev. 39, 5 (Oct. 2005), 148–162. https://doi.org/10.1145/1095809.1095825

[49] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. 2002. Scale and Performance in the Denali Isolation Kernel. SIGOPS Oper. Syst. Rev. 36, SI (Dec. 2002), 195–209. https://doi.org/10.1145/844128.844147

[50] Dan Williams and Ricardo Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud ’16). USENIX Association, Den- ver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams

[51] 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

3 HyperNF: Superfluidity Platform Optimizations for NFV

The results reported in this section have been published in:

K. Yasukata, F. Huici, V. Maffione, G. Lettieri, M. Honda, “HyperNF: Building a High Performance, High Utilization and Fair NFV Platform”, ACM Symposium on Cloud Computing 2017 (SoCC '17), September 25-27, 2017, Santa Clara, California,

Available at: http://sysml.neclab.eu/projects/hypernf/hypernf.pdf

Large portions of text in the following sub-sections correspond to the text in the paper.

3.1 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.

3.2 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. In particular, in order to reduce costs and to be able to provide accurate resource allocation and meet SLAs we would need:

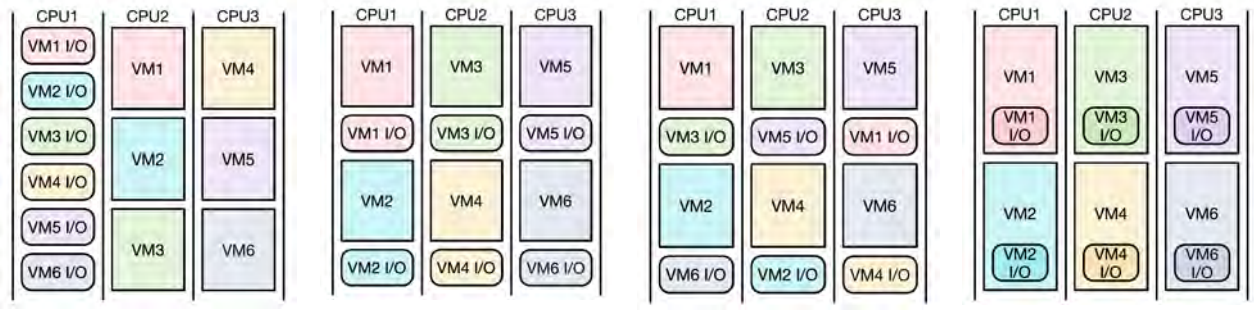
- High Utilization: Ensuring that CPU cycles are neither wasted (e.g., by busy polling when no packets are available) nor idle (e.g., by dedicating cores to I/O when they could be used by VMs).
- Adaptability: The platform should be able to dynamically assign idle resources in order to cope with peaks 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.

3.2.1 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).



(a) Split (b) Merge (c) Mix (d) HyperNF

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.

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.

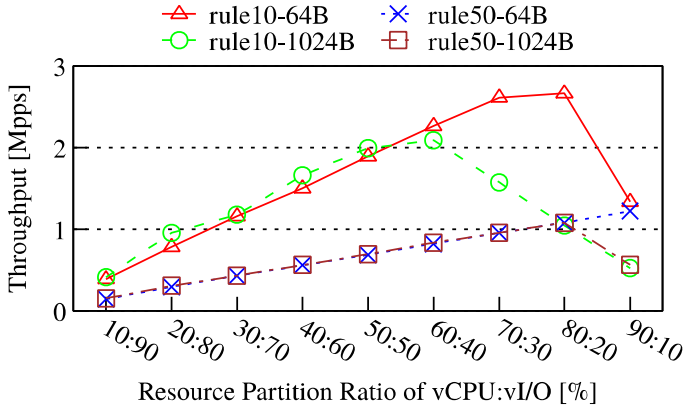


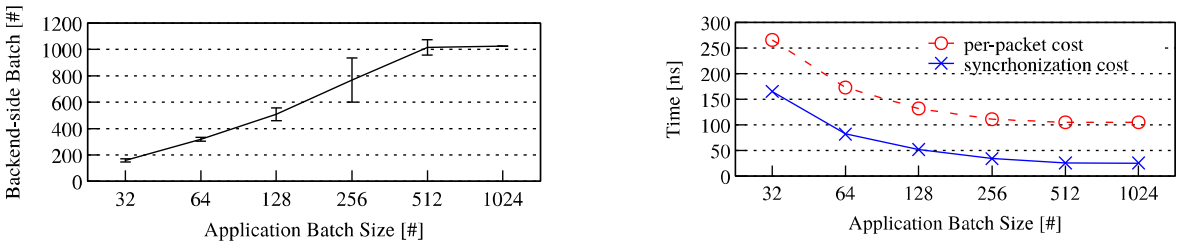
Figure 2: Throughput when forwarding traffic through a firewall VM for different static CPU assignments to the VM versus the I/O threads, different packet sizes, and different number of firewall rules. No assignment yields the highest throughput for all cases.

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.

3.2.2 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 VMEXIT instruction, along with thread handoff 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 "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.



(a) Specified application batch size versus actual measured batch size in the bridge. Error bar is the standard deviation.

(b) Per-packet cost for different application batch sizes.

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

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.

3.2.3 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.

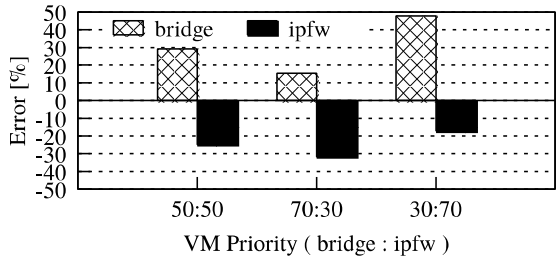


Figure 4: Error when comparing expected throughput from two VMs (as set by Xen's prioritization mechanism) versus actual forwarded throughput.

We then measure how much the forwarded traffic deviates from the split we specified and calculate: $error$ = ($goodput$− $expectedt\square roug\square put$)/$expectedt\square roug\square put$×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 Xenspecific: 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.

3.3 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.

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

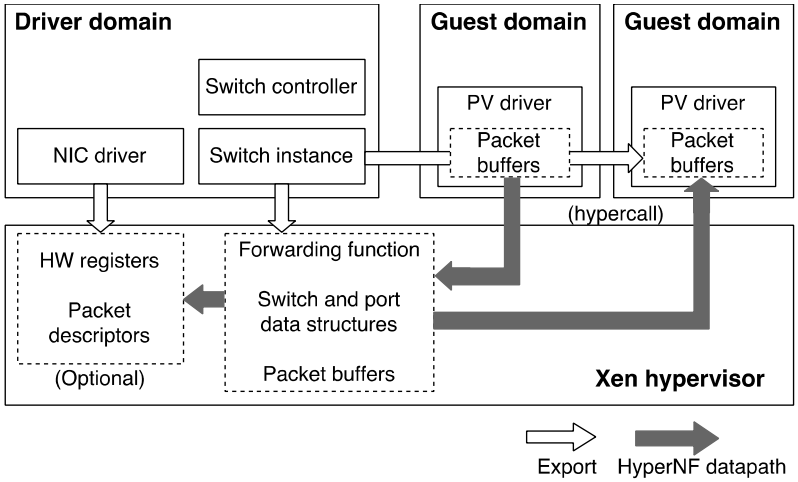


Figure 5: HyperNF architecture. Packet switching and forwarding are done through a hypercall by exporting the back-end software switch's data path to the hypervisor and, optionally, the fast path of NIC drivers. Tx/Rx operations issued from a VM are carried by the hypercall and do not require VM switches.

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.

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.

3.3.2 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 deviceindependent 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.

3.3.3 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.

3.3.4 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.

3.3.5 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].

3.4 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.

3.4.1 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 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).

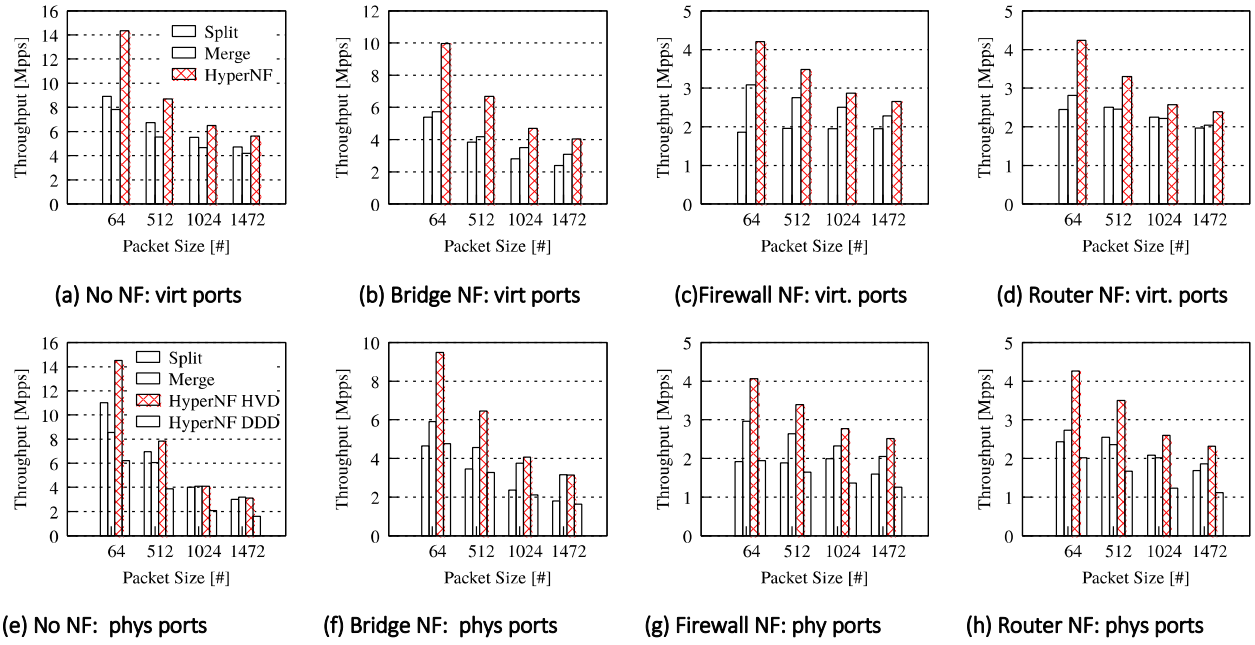


Figure 6: Throughput when running different NFs one at a time using the split model, the merge model and HyperNF with virtual (top graphs) and physical ports (bottom graphs).

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. 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.

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.

3.4.2 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.

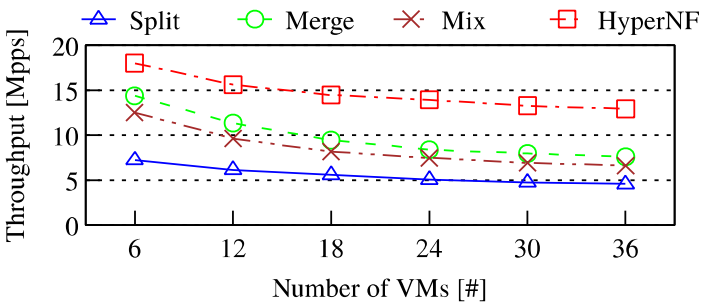


Figure 7: Cumulative throughput on the NFV platform when multiple NFs share cores. In this experiment 6 cores are dedicated to the VMs and assigned in a round-robin fashion. Packet size is 256B.

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

3.4.3 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.

	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.

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 $error = (goodput - expectedt\square roug\square put)/expectedt\square roug\square put \times 100$ where $expectedt\square roug\square put = baseline \times VM\ priority/100$ (the baseline is the performance when a VM has 100% of a CPU core for running the NF).

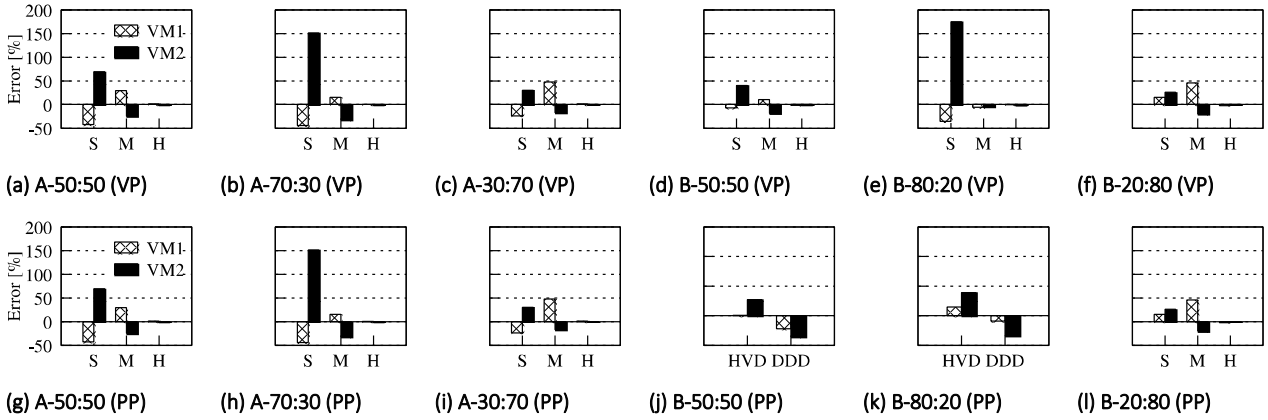


Figure 8: Accuracy of resource allocation when running on virtual ports (VP) and physical ports (PP) with two VMs and different priority ratios. In VP graphs, S, M and H represent split, merge and HyperNF, respectively. For PP, the generator runs on the same host as the VMs and the receiver on an external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

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%.

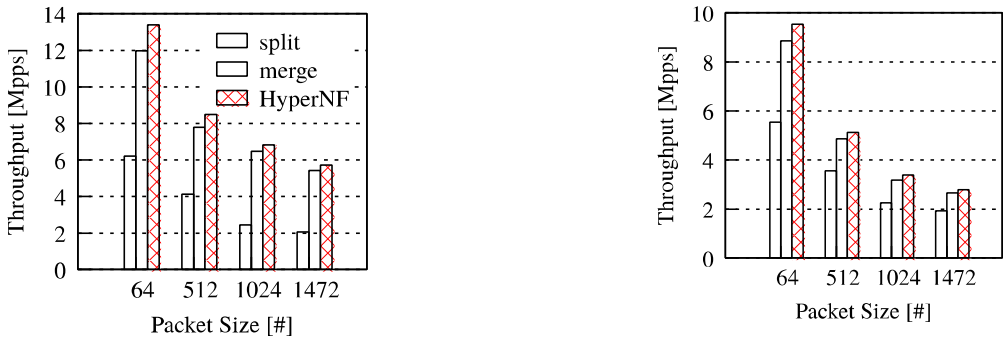
For the test with physical ports we use the same set up as above but move the pkt-gen receivers to an external host (one of our 4-core servers), and, as usual, run two experiments, one with the

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.

3.4.4 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.



(a) No NF, VM to VM

(b) Forwarding NF

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.

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.

3.5 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).

3.5.1 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.

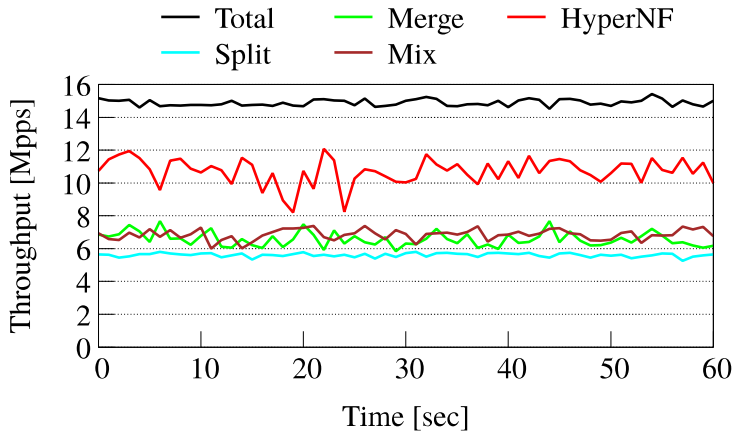
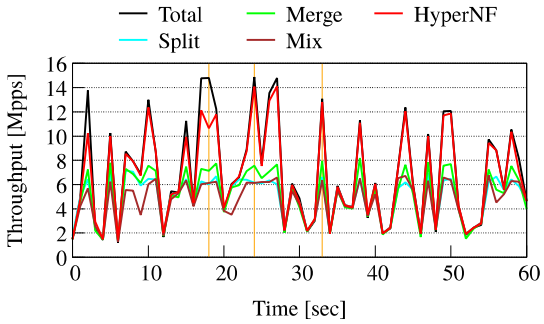
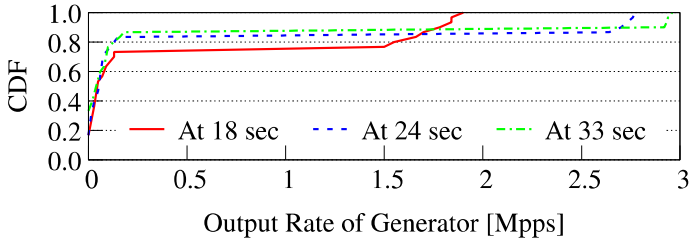


Figure 10: Cumulative forwarded throughput for the dynamic per-VM traffic scenario. Packet size is 256B and the VMs all run a 10-rule firewall. The "Total" curve is the offered throughput.

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).



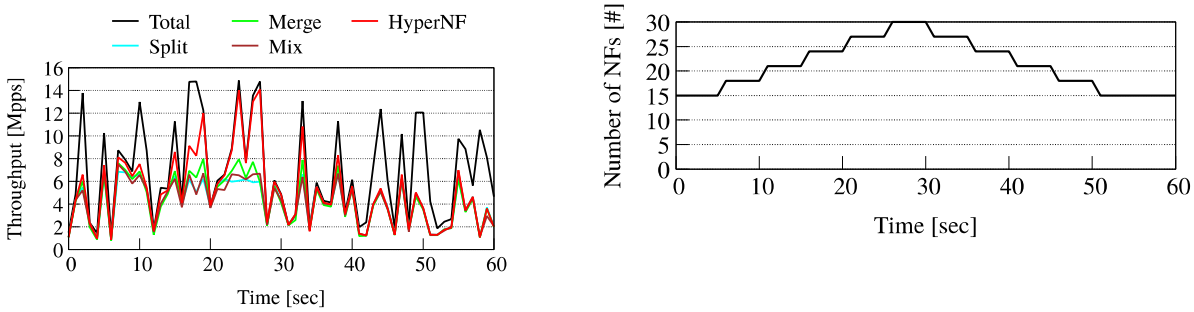
(a) Throughput



(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.

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.



(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.

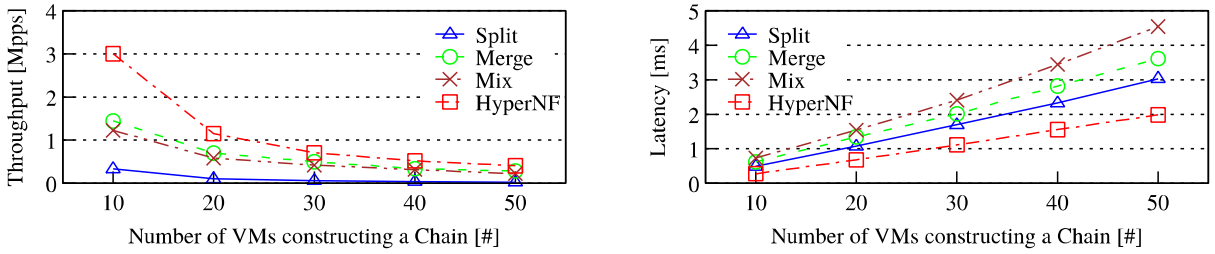
In the final scenario (dynamic traffic, dynamic NFs) we use the same setup as in the previous scenario, except this time we begin with only 15 VMs at t = 0 and bring three new VMs up every 5 seconds (in actuality we unpause existing VMs so that they become immediately active), up to a total of 30 VMs. 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.

3.5.2 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.

3.6 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.

3.7 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.

3.8 References for section 3

[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: Baremetal 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. Universit`a 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,

4 Efficient CPU Sharing in the Superfluidity Platform

The results reported in this section have been published in:

L. Vasilescu, V. Olteanu, C. Raiciu, “Sharing CPUs via Endpoint Congestion Control”, ACM SIGCOMM Workshop on Kernel-Bypass Networks, KBNets '17, August 21 - 21, 2017, Los Angeles, USA.

Available at: https://dl.acm.org/citation.cfm?id=3098589

Large portions of text in the following sub-sections correspond to the text in the paper.

4.1 Introduction

Modern networks are more than just bit pipes; they also offer processing and sometimes even storage. Traffic no longer just flows through, it is also processed in the network by “middleboxes” including firewalls, WAN optimizers, tunnel endpoints or application level caches. As network processing in ASICs or bespoke hardware is difficult to upgrade and scale, a recent push towards running network processing as software on commodity hardware, called NFV, has already moved from research into deployment: major operators are running virtualized network functions on small data centers deployed throughout their networks. This trend is only set to continue, with datacenters being pushed close to the clients at the mobile edge (a trend called Mobile Edge Computing). In data centers, operators rely on commodity processing to implement various functionality (e.g. firewalling, NATs, load balancing).

The shift towards in-network processing stems, to some extent, from the popularity of cloud computing, and it comes as no wonder that the current mindset towards optimizing new networks is similar to optimizing clouds: the general formulation is that a user wanting service will specify the amount of bandwidth and processing it needs for its traffic, and a centralized controller will solve an optimization problem to decide where to place the computing and how to route the traffic [2]. We observe that this approach has a few drawbacks: a) it is very difficult to characterize traffic demand and processing demand ahead of time, b) provisioning decisions are made on subsets of the path (e.g. the ISP network), and thus do not take into account the end-to-end properties of traffic, and finally c) one needs to reserve resources for peak load, leading to wasted resources. Putting this in context, we can conclude that modern multi-resource networks are allocated using the equivalent of circuit switching in the old telephone networks. While this type of allocation offers great isolation properties, it is also very inefficient.

The starting point of this paper is the observation that all resources, including processors, main memory and storage, are used, most often, to produce, consume or to transfer data, and can be viewed as special types of network “links” with different speeds.

We want to explore what would happen if we do away with cloud-style reservations for in-network processing: is it possible to adopt the best-effort per-packet service in the context of sharing other resources beyond bandwidth? In other words, we would like to rely on endpoint congestion control to share network processing resources in the same way it shares network capacity.

We rely on experimentation to see whether traditional OS process scheduling is sufficient to share network processing. Our results show that sharing CPUs via traditional OS mechanisms is suboptimal, as it leads to inefficient and unfair resource allocations in many situations. Next, we propose changes to the buffering discipline used by the network processes that remedy the problems identified in our experiments. We evaluate these changes in simulation, showing that they achieve both good fairness and resource allocation.

4.2 Background and Approach

A wide body of existing research has focused on achieving high packet processing performance on commodity hardware, typically by bypassing the kernel: packets move from the NIC directly to the process and back out on the NIC [7, 12] as shown in Fig. 1.a. This is accomplished by mapping the send and receive packet rings in application memory; the kernel is only used as a control plane to set-up the direct datapath between apps and the NIC. To check for available packets apps either rely on polling (with DPDK [7]) or use syscalls (netmap [12]) which effectively signal NIC interrupts to apps. By default, a single process gets exclusive access to the whole NIC, but it is also possible to have many applications share the NIC by using hardware multiplexing: modern NICs have tens/hundreds of send/receive queue pairs, each of which can be exclusively assigned to a different process. Incoming traffic can be directed to the various queues by using hashing (called Receive Side Scaling, or RSS), or via explicit filters (e.g. Flow Director in Intel NICs). In virtualized environments, kernel bypass is used in conjunction with hypervisor bypass: the NIC rings are mapped directly in the guests' memory by using hardware virtualization (called SR-IOV).

Existing work has shown that it is possible to process packets for multiple 10Gbps NICs on commodity hardware for packet forwarding [5, 12], load balancing [3, 9], NATs [10], caches [8], etc. However, all these works consider processing in isolation, on a dedicated core or a machine. The main assumption is that the same processing is applied to all traffic coming from an interface and, in this context, existing solutions are up to the job.

Almost all current work assumes CPU resources are reserved for network processing, and wide-area provisioning even reserves bandwidth between the different processing sites. Network processing, however, must be able to cope with smaller traffic volumes belonging to multiple tenants. Consider a mobile edge cloud deployment, close to a cellular base station: every mobile user or even application could have a personalized firewall and proxy wanting to run on the edge cloud. As the number of users is on the order of a few thousand per cell, how should one provision the computing resources at the mobile edge? Today's approach would be to reserve processing for every single

user's firewall, but that is obviously wasteful, since most users are silent most of the time and have intermittent traffic bursts where the peak speed could be in the tens or hundreds of Mbps.

Packet-switching, coupled with endpoint congestion control, is a much better way to share Internet links. Packet switching enables resource pooling [13]: the total capacity on a path can be used by any flow traversing that path, and endpoint congestion control adapts the rates to network conditions, ensuring that flows get their fair share when they cross bottleneck links.

Would it be possible to resource pool CPUs for flows that require network processing in the same way that we pool bandwidth? To enable resource pooling, we can simply schedule multiple processing functions on the same CPU core, and have the flows share the CPU in the same way they share bandwidth: on a best effort basis. This simple idea is shown in Figure 1.b: the three processes are scheduled on the same CPU, each being given a receive ring by the NIC and taking turns at processing packets. When a process is not running, its packets will be buffered in the NIC ring buffers; once the process is scheduled, it will process the packets from its ring until either its scheduling quantum expires or there are no more packets to process. If the process cannot keep up with the incoming packet rate, the ring buffers will fill up, leading to packet drops.

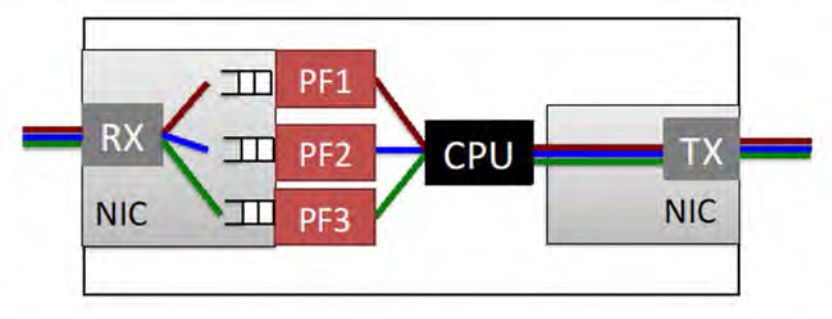


Figure 1: Network processing on commodity hardware: moving from reserved cores to resource pooling.

Running multiple processing functions on the same CPU is easy enough to do, and it will enable memory isolation between the processing functions via the OS or hypervisor mechanisms. The interesting remaining question is whether it will enable the efficient type of bandwidth resource pooling prevalent in the Internet today. Network links have the following properties that enable efficient sharing via endpoint congestion control:

- Constant throughput: regardless of the number of flows, a (wired) link's total throughput is constant.
- Proportional end-to-end congestion notification - all flows sharing a bottleneck link perceive the same average loss rate and this enables endpoint congestion control to ensure high utilization and fairness at the same time.


If the properties hold for CPU sharing, then sharing will be as simple as Fig. 1. In the next section, we set out to discover the answer experimentally.

4.3 Processor Pooling in Networks

In our experiments we rely on FastClick [1], a version of the Click modular router [6]. Click is a well known packet processing framework that has hundreds of pre-compiled elements that can be combined in directed packet-processing graphs called configurations to implement a wide range of network processing functionality [6]. FastClick [1] solves the performance problems of Click and enables simple, user-mode deployment: it runs on top of netmap [12], automatically assigns queue-pairs to Click processes, sets CPU affinity and has batching support to improve performance.

We use a simple Click configuration that simply forwards packets from input to output and changes the MAC addresses. We run multiple copies of this configuration as separate FastClick processes that are assigned to run on the same CPU core, and each process is assigned a separate NIC queue-pair. To simulate more complex processing functionality, we also add an element to the forwarding configuration that executes a fixed number of cycles per packet. We generate packets from a sender using the pkt-gen tool from netmap, ensuring that load is split equally among the Click forwarding processes. We measure per-process and total throughput.

4.3.1 Throughput

In all our experiments, we generate 128B packets at 10Gbps (8.22 million packets per second) and forward them using a variable number of processes, all sharing the same CPU core. We use fairly large buffer rings (1024 packets) per process, and similar FastClick batches. In our initial experiments, we found that FastClick is able to sustain line rate for 128B packets when a single process is used, but the throughput dropped quickly when we added more processes: with 5 processes, total throughput was a mere 5.1Mpps. We expected a throughput reduction due to context switching overheads, but this was rather extreme. To reduce the effects of context switching, we first used the round-robin scheduler and modified the scheduling quantum, trying values of 1ms, 10ms and 100ms. Surprisingly, performance was the same regardless of the quantum, and a closer look at scheduler statistics shows that the number of context switches was more or less the same for all quanta, instead of being inversely correlated. This implies that our processes were never preempted by the OS scheduler. To understand why, we show the main loop executed by each Click process below:

```
while(1){
       poll(...);//read packet batch from NIC
       process_batch(...);//process packets
       ioctl(TXSYNC,...);//send packets out
}
```

A process will be scheduled when poll has packets ready; at that point the process will read and process all the packets available in its receive ring. The process_batch call can handle at least 8200 packets per ms (as shown by the single process results), and we expect that a batch of 1024 packets will take around 100µs; after this, the packets are placed on the (most likely empty) send ring and the OS is notified.

Next, the process will invoke poll again; this call is unlikely to block since in the meantime some more packets have arrived in the process's ring; these packets will be further processed, another smaller batch retrieved, and so on, until there are no packets in the receive ring. At this point the process will block. This effect is clearly shown in Figure 2 where a CDF of actual batch sizes is shown for an experiment with 5 processes sharing the same core: more than 90% of batches contain ten or fewer packets, while the remaining ones have 1000 packets or more.

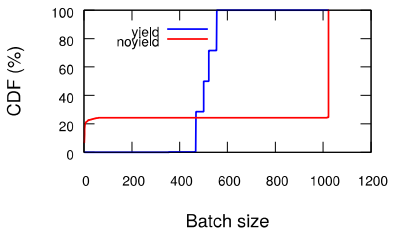


Figure 2: Batch sizes used by Click (5 processes per core, 1024 packets per receive ring).

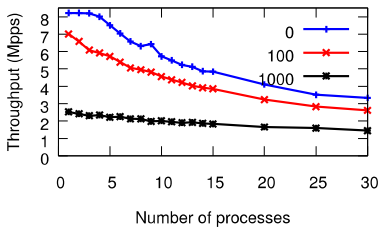


Figure 3: Total throughput vs. # Click processes per core (forwarding, batch = 1024 pkts)

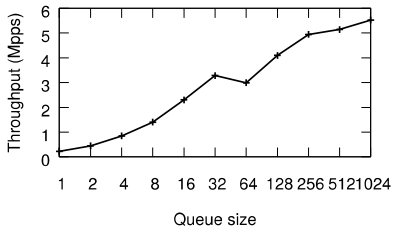


Figure 4: Total throughput as we vary the per-NF queue size (forwarding, ten processes)

A simple fix is to have each process yield the processor after it processes the first batch. We implemented it by adding a sched_yield call before poll and reran the experiment in Figure 2. The batch size distribution is a Gaussian distribution centered around 500 packets. The throughput achieved using yield is around 7.5 Mpps, while the throughput without using yield is 5.1 Mpps. With yield, we reran the throughput experiments, varying the number of processes used and the cost per packet. The results are shown in Figure 3, for the case where processing is plain forwarding (line 0), or when an additional number of cycles is spent per packet (100 or 1000). With plain forwarding, when we have 30 processes the total throughput is 3.5Mpps (or 4.3Gbps), a 60% reduction. This is because a large number of processes fighting for the cache means there is little or no data in the cache when a process is re-scheduled. Note that the maximum decrease is reached at 25 contending processes, and thus adding further processes does not further decrease total throughput. The overheads strongly depend on how costly the packets are to process. As packets become more

expensive, it takes more time to process a batch of packets, reducing the number of context switches and their relative overheads. When 1000 cycles are spent per packet, the overhead of running with 30 processes instead of a single one is 40% (from 2.5Mpps to 1.5Mpps). Even higher per-packet costs further reduce overheads: with 10000 cycles per packet, throughput drops by 16% from 288Kpps (1 process) to 239Kpps (30 processes).

Effect of buffer sizing. Batching is one of the main techniques used to amortize the costs of system calls: the netmap paper [12] shows that batches of at least 10 packets per syscall are needed to amortize syscall overheads for the packet generator. In our case, we control batches by setting the appropriate number of packets in the NIC rings and FastClick. To understand the effect of buffer sizing, in our next experiment we ran plain forwarding (no extra cycles per packet) with different sized ring buffers, varying the number of FastClick processes as above. The results show that higher queue sizes and, implicitly, FastClick batches result in better total throughput: using ten processes and 128 packet buffers results in a throughput of 3.7Mpps; with 1024 buffers the throughput is 5.5Mpps, a 50% improvement.

The results so far show that CPU sharing is fairly efficient for network processing: while total throughput is reduced by 50% in the worst case we have tested, in practice we expect the reduction to be around 10%-20%. Even with a 50% drop in throughput, the potential savings due to pooling are massive: we only need two CPU cores to guarantee 10Gbps forwarding performance for 30 tenants; with reservations we would need 30 cores.

4.3.2 Sharing Behaviour

Our experiments so far use non-responsive flows where the packet rates are exactly the same. These flows are then mapped to different receive rings by the NIC and processed by separate processes, where the per-packet cost could vary between processes. If the flows were sharing a normal link (whether drop-tail or RED-based) instead of a CPU, they would experience the same loss rate and achieve similar rates. We experimentally measured how the CPU is shared in practice. Our experiments, shown in Fig. 5, highlight three CPU sharing outcomes that depend on how costly the per-packet processing is. All of these outcomes are ill-suited for resource pooling.

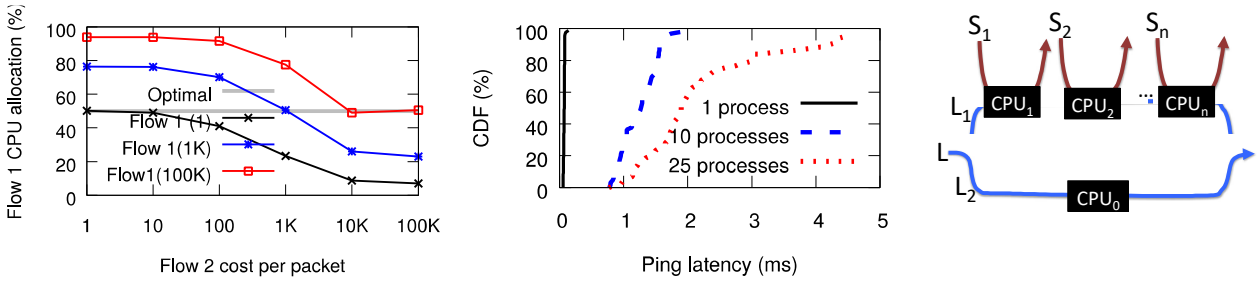


Figure 5: Throughput distribution for two competing processes processing flows with different per-packet costs.

Figure 6: Ping latency measurements with different numbers of processes contending for the CPU.

Figure 7: Line topology: a multipath flow competes with many short flows on one of its paths, and is alone on the other path.

Packet-level fairness for cheap packets. When context switches are triggered by blocking on I/O rather than the OS scheduler, the emerging behaviour is that of packet-level fairness among the different processes contending for the same core; this is the case for all experiments discussed previously. This is intuitive, since each process will run through a similarly sized batch (1024 packets or less) and then lose the CPU.

Packet-rate fairness does not imply fair sharing of the CPU. When the processes spend different number of cycles per packet, the CPU allocation is unfair: for instance, a process that forwards 1000 cycle packets competes with another process that does simple forwarding, each process forwards 1.25Mpps and the expensive process gets around 80% of CPU. This is clearly shown in Figure 5 when there is a difference between the costs of the different flows.

CPU fairness for expensive packets. When the time taken to process a batch is more than 1ms (the scheduler quantum), the sharing is dictated by the OS scheduler and each process gets roughly the same share of the CPU. In Figure 5, note how a flow spending 10K cycles per packet gets the same CPU time as one spending 100K cycles per packet.

Unfair allocation for mixed scenarios. When some processes exceed their quantum and some don't, the CPU is given preferentially to the more expensive processes. For instance, when a plain forwarding process competes with one that fully uses its quantum, the forwarding will only get 10% of the CPU; the fair outcome would be 50% of the CPU.

In summary, the OS scheduler does a poor job of sharing the CPU among network-processing tasks.

4.3.3 Latency

The next question is to what extent resource pooling the CPU increases packet-level latency. The scenario we examine is when we have P CPU-hungry network processes (such as the ones we have examined in our experiments until now) and one idle process that must simply forward ping traffic. In the worst case where each process uses its 1ms scheduling quantum, the ping latency could be inflated by upto P ms—is this what happens in practice?

We ran one process that forwards only ICMP ping packets, measuring the end-to-end latency as reported by ping in three experiments: when the ping forwarder runs alone, and when it shares the core with other 10 or 25 forwarding processes (as above). The results, shown in Figure 6, show that

ping latency is around 120 µs on an idle core. When competing with 10 forwarding processes the median latency jumps to 1.3ms, and the tail to 2ms. With 25 processes, the median is 1.8ms and tail is 4.5ms. This is significantly better than the worst case 25ms, but still significant. To reduce latencies further, we could run smaller queues, which in turn would hurt total throughput. Within a single machine, however, it should be possible to schedule lighter processes on one core and busy processes on other cores, and using differently sized queues for the two categories.

4.3.4 Multiple Bottlenecks

Real networks are much more complex than the dumbbell topology we've used so far (with the CPU as the bottleneck). In such networks, per-resource fair-sharing can lead to inefficient use of network capacity; this is what we want to explore in this section. We resort to simulation in the high-speed htsim simulator from [11]. An interesting yet simple topology to study is the one in Figure 7, where a multipath flow called L has two subflows taking different paths. The first subflow, L1, traverses n bottlenecks (where n is configurable), sharing each bottleneck with a single other short flow (Si). The second subflow, L2, traverses a single bottleneck that it uses exclusively.

For simplicity, we assume all flows have packets that are equally costly for each bottleneck, and that one CPU can process a flow at 100Mbps. We are interested in how throughput is distributed among the short flows and the multipath flow, and what the total network throughput is. To emulate OS scheduling, we have implemented in htsim a fair queueing mechanism that can process a configurable number of packets per batch; when we set the maximum batch size to one, we have a perfect fair queue implementation; when we set it to 1024 we mimick the behaviour seen in our experiments above.

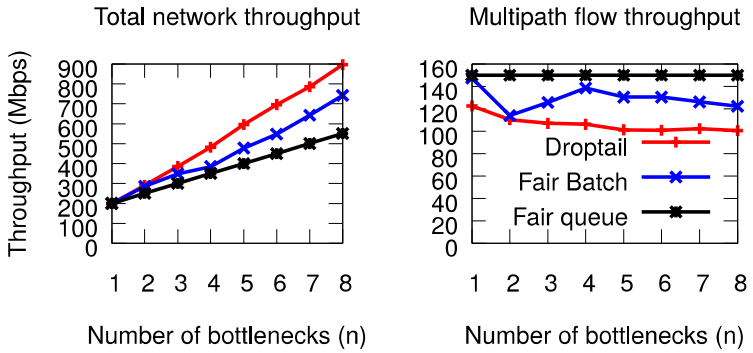


Figure 8: Fair queueing hurts total network throughput and overall fairness.


We vary n, the number of CPUs and their associated short flows, and measure total throughput obtained by the Multipath TCP long flow, as well as the throughput of the short flows. Multipath TCP pushes traffic away from congested links: in our example, a perfect MPTCP congestion controller will measure a strictly higher loss rate on its L1 subflow, and will therefore move traffic away from the top link and fill the bottom link instead, regardless of n [14].

When each CPU is simulated using a standard drop-tail link, the results shown in Figure 8 confirm our expectations (with small deviations from theory when n is small): the multipath flow pushes most of its traffic on the bottom path, and total network throughput is within 2% of the theoretical optimal. However, when we simulate the CPU as a simple fair-queueing link, subflow L1 gets 50Mbps regardless of n, reducing the speed of the n short flows to 50Mbps each and driving total network throughput to half the optimal. In other words, giving 150Mbps to the MPTCP connection is done by wasting 50Mbps · n capacity; the larger n is, the larger the wasted throughput. This happens because the MPTCP subflow will only get signals from the first fair queue (when it overflows), and not the rest. Finally, we ran an experiment where we use batching in the fair queue, mimicking OS packet processing. The outcome is slightly better than per-packet fair queueing, but the top subflow still gets around 25Mbps regardless of n, with the appropriate drop in total network throughput.

To summarize, fair queueing is inadequate for wide area network sharing because it can severely decrease total network throughput; drop-tail behaviour is desirable for this particular topology

4.4 Towards a Solution

We now explore potential solutions to enable appropriate sharing of CPUs. A good solution must behave like a droptail queue, sending congestion signals proportionally to all participating flows, but it must send more signals to more expensive flows to enable fair sharing of the bottleneck resource.

When sharing a bottleneck link, all TCP senders will converge to the same CWND as they measure the same loss rate: on average, CWND=$\sqrt{\frac{2}{p}}$ where p is the loss rate. Consider now two flows F1 and F2 sharing a bottleneck link and having the same round trip time, and F2's packets are twice the size of F1's. The two flows will get the same congestion signals and will have the same average congestion window, giving F2 twice the throughput of F1.

To enable fair-sharing of the bottleneck link, F2's window must be half of F1's, so F2 must see a loss rate four times higher than F2. A variant of RED called RED_4 does exactly this [4]: it sends more congestion signals to flows that have higher packet sizes. RED_4 computes a mark rate p for its cheapest flow based on the buffer usage, and marks other flows packets with probability $\frac{MSS_i^2}{MSS_{min}^2} p$.

We can use RED_4 in our case with a slight change: instead of using the packet size in the formula above, we use the average time it takes to process a packet instead. We have implemented RED_4 in htsim and measured how much throughput F1 achieves as a function of F2's packet size, showing results in Fig. 9. First, notice how per-packet droptail queues allocate the bottleneck unfairly; our experiments in the previous section have shown that fair queueing behaves equally bad, giving the expensive flow 2 a larger share. RED_4, on the other hand, behaves really well, being within 10% of the optimal allocation. Finally, notice that F1 starts to get slightly more capacity when F2 is more costly: this is because our simulator does not implement ECN, and our congestion signals are packet drops. When F2 has really large packets, it starts experiencing high loss rates that trigger timeouts, further reducing its rate.

The sharing behaviour of RED_4 is really encouraging; RED_4 also behaves correctly in the line topology from Figure 8. However, RED_4 assumes a single queue is shared by all flows, and retrofitting this architecture to the OS would be very inefficient. Instead, we propose RED_4 MQ, a variation of RED_4 that uses multiple per-flow queues instead of a single queue, and that can be easily implemented on top of the architecture in Fig. 1. RED_4 MQ applies the same RED_4 algorithm on packet enqueue, but uses the sum of all queue sizes to decide the drop probability instead of the size of the single queue in the original algorithm. We have implemented it in htsim and use a default batch size of 1024, as used in our testbed experiments in section 3.

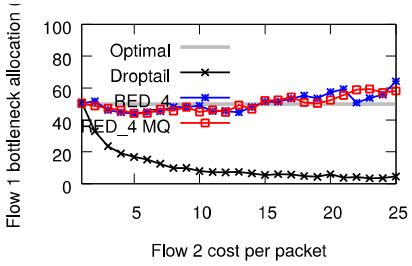


Figure 9: Throughput distribution among two competing flows with different packet sizes

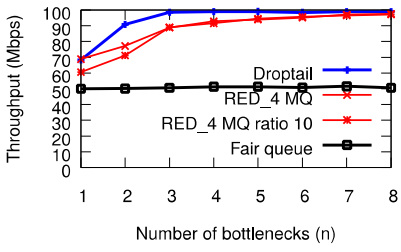


Figure 10: RED_4 MQ performs close to optimal in the line topology, while fair-queueing hurts total throughput.

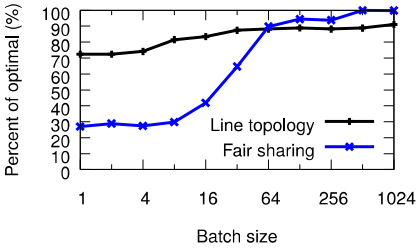


Figure 11: Effect of batch size on the performance of RED_4 MQ: larger batches give better performance.

Fig. 9 shows how RED_4 MQ shares a bottleneck among two competing flows with different per-packet costs: RED_4 MQ performs very well, similar to the single queue RED_4 algorithm. In the line topology (see Fig. 8), RED_4 MQ performs within 10% of droptail queues, which is nearly optimal.

Finally, to understand why RED_4 MQ behaves so well, we studied its sensitivity to the batch size in the fairness and line topologies. Figure 11 shows the performance of RED_4 MQ compared to optimal

as a function of batch size. The results show that batching is crucial to achieving the correct behaviour; intuitively, when we have small batches, the emergent behaviour of the algorithm is dominated by fair-queueing, as all flows will be serviced roundrobin and slowly converge to the same window. When batching is used, on the other hand, the RTT in the line topology increases significantly for expensive flows, reducing their effective rate, and burst losses are much more likely when n is large.

Practical considerations. Implementing RED_4 MQ is our future work: it requires some minor modifications to netmap (kernel code) to measure the total number of packets in all receive rings and average packet processing times per process. Understanding in more depth the emergent behaviour of RED_4 MQ requires careful theoretical analysis (e.g. using fluid models). Using these insights, we intend to look beyond RED_4 MQ for alternatives that do not rely on RED, an AQM notoriously difficult to configure correctly.

Our work has so far assumed that traffic passes through a middlebox (e.g. a firewall); however certain middleboxes terminate TCP connections (e.g. a proxy). In such cases, when the proxy is overloaded TCP flow control will throttle the senders. Sharing CPUs correctly in this case requires further thought

4.5 Implications

Our work shows it should be possible for multiple network processing functions to share a CPU efficiently. Furthermore, our RED_4 MQ active queue mechanism provides appropriate congestion signals to the endpoints such that CPUs can be shared via endpoint congestion control, just like networks. CPU resource allocation becomes now a part of network routing: the network decides the links and the CPUs a certain flow should take, and the flow modifies its rate according to the congestion signals it receives. This will enable operators to load balance traffic across its network links and network processing load across its processors.

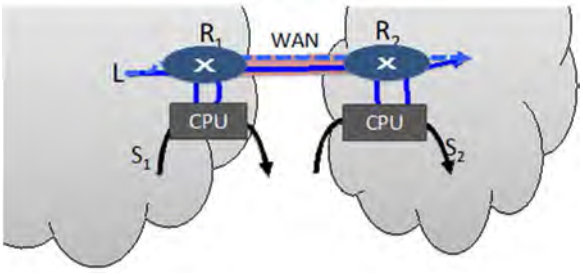


Figure 12: To compress or not? Using CPU pooling and Multipath TCP to dynamically select the most efficient way to transport data between two datacenters

The benefits of CPU processing are however much higher than pooling bandwidth or processing in isolation: they also allow cross resource pooling, i.e. trading off between bandwidth and CPU utilization. As an example, consider the task of sending web crawl data between the two datacenters shown in Figure 12: one can send the data as is on the busy WAN link, or compress the data at the source, transfer and decompress it at the destination datacenter. It is almost impossible to decide ahead of time what the best strategy is: if the CPUs are idle and the data is compressible (e.g. text), compression is obviously superior; if the CPUs are busy or the data compresses poorly, it is better to send directly without compressing. Ideally, one should dynamically load balance across the two paths as done by the Multipath TCP flow L: one subflow sends directly over the WAN link, while the other crosses via the CPUs to compress and decompress the data. The loss rate on the two subflows will tell the Multipath TCP controller how to send most traffic: when the CPU is congested, the loss rate on the bottom subflow will be higher and most traffic will move to the top subflow if the WAN link is uncongested; conversely, when the WAN link is congested, more traffic will be sent via the bottom subflow and thus be compressed.

4.6 References for section 4

[1] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In ANCS 2015.

[2] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz. Near optimal placement of virtual network functions. In INFOCOM 2015.

[3] Daniel E. Eisenbud et al. Maglev: A fast and reliable software network load balancer. In NSDI 2016.

[4] S. de Cnodder, O. Elloumi, and K. Pauwels. Red behavior with different packet sizes. In ISCC 2000.

[5] M. Dobrescu, N. Egi, K. Argyraki, B. G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In SOSP, 2009.

[6] E. Kohler et al. The Click modular router. ACM Trans. Computer Systems, 18(1), 2000.

[7] Intel Corporation. DPDK: Data Plane Development Kit. http://dpdk.org/.

[8] S. Kuenzer, A. Ivanov, F. Manco, J. Mendes, Y. Volchkov, F. Schmidt, K. Yasukata, M. Honda, and F. Huici. Unikernels Everywhere: The Case for Elastic CDNs. In VEE 2017.

[9] V. Olteanu and C. Raiciu. Datacenter scale load balancing for multipath transport. In HotMiddlebox 2016.

[10] V. A. Olteanu, F. Huici, and C. Raiciu. Lost in network address translation: Lessons from scaling the world's simplest middlebox. In HotMiddlebox 2015.

[11] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In SIGCOMM 2011.

[12] L. Rizzo. netmap: A novel framework for fast packet i/o. In Proc. USENIX Annual Technical Conference, 2012.

[13] D. Wischik, M. Handley, and M. B. Braun. The resource pooling principle. SIGCOMM CCR October 2008.

[14] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In NSDI 2011.

5 Performance evaluation and tuning of Virtual Infrastructure Managers (VIMs) for Unikernel orchestration

Some of the results reported in this section have been published in:

P. L. Ventre, C. Pisa, S. Salsano, G. Siracusano, F. Schmidt, P. Lungaroni, N. Blefari-Melazzi: "Performance Evaluation and Tuning of Virtual Infrastructure Managers for (Micro) Virtual Network Functions", 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), 7-9 November 2016, Palo Alto, California, USA.

Available at: http://netgroup.uniroma2.it/Stefano_Salsano/papers/salsano-ieee-nfv-sdn-2016-vim-performance-for-unikernels.pdf

Portions of text in the following sub-sections correspond to the text in the paper.

A submission to a Journal (IEEE Transaction on Cloud Computing) including the remaining results of this section is under preparation.

The proposed general model of the VM instantiation process is shown in Figure 5-1. We decompose the operations among the VIM core components, the VIM local components and the Compute resource/hypervisor. The VIM core components are responsible for receiving the VNF instantiation requests (i.e. the initial request in Figure 5-1) and for choosing the resources to use, i.e. the scheduling. This decision is translated into a set of requests which are sent to the VIM local components. These are located near the resources and are responsible for enforcing the decisions of the VIM core components by mapping the received requests to the corresponding hypervisor technology API calls. These APIs are typically wrapped by a driver, which is responsible for instructing the Compute resource to instantiate and boot the requested VNFs.

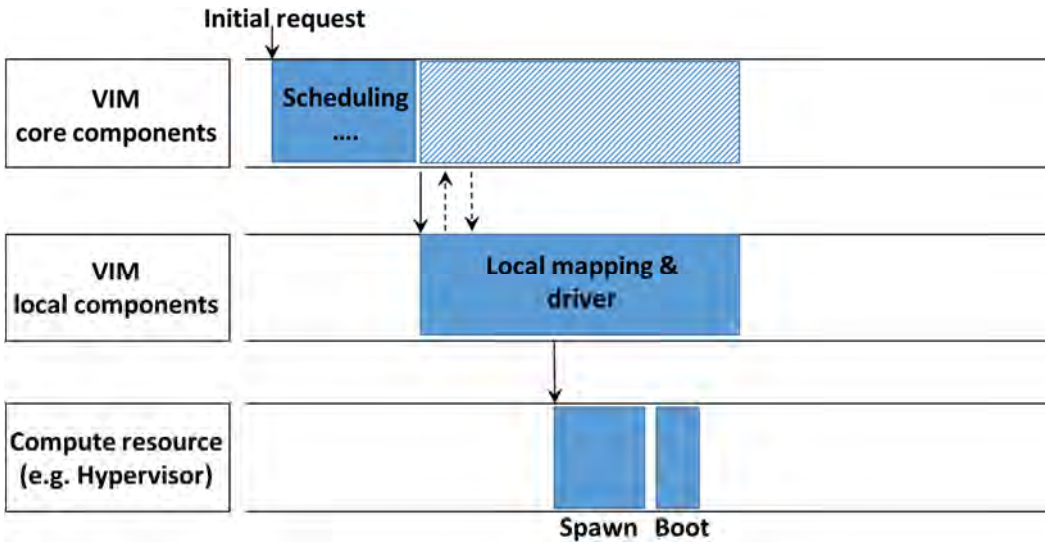


Figure 5-1: VIM instantiation general model

Figure 5-2 shows how the OpenStack, Nomad and OpenVIM components can be mapped on the proposed reference model. In the next subsections, we provide a detailed analysis of the operations of the three VIMs. As for the Compute resource/hypervisor, in this work we focus on Xen [1], an open-source hypervisor commonly used in production clouds, as the resource manager. Xen can be configured to use different toolstacks (tools to manage guest creation, destruction and configuration). A toolstack can interact with the hypervisor directly or using a toolstack API. The toolstack API provided by Xen is called libxl and is adopted by the majority of Xen compatible toolstacks.

Virtual Infrastructure Managers (VIMs)	Nomad	OpenStack *	OpenVIM
VIM core components	Nomad Server	NOVA APIs, Keystone, Glance, NOVA Scheduler, Cinder, **Neutron Server, Neutron Plugins**	OpenVIM Core MySQL
VIM local components	Nomad Client	Nova Compute, Nova Network, **Neutron Agents**	OpenVIM Local (SSH daemon, 3rd party software)
Compute resources	Xen (xl)	Xen (libvirt)	Xen (libvirt)

* Bold items have been introduced by Neutron

Figure 5-2: Mapping the reference model onto the considered orchestrators

5.1 Modelling OpenStack

We have considered two models for OpenStack, which differ especially in their approach to networking: we refer as “OpenStack Legacy” to the model, which relies on the Nova Network component, employed by pre-Newton OpenStack releases, while we refer simply as “OpenStack” to the model based on the Neutron networking component. Indeed, Nova Network, has been now deprecated, but we believe the results can still be interesting for legacy OpenStack deployments still in operation.

5.1.1 OpenStack Legacy

Figure 5-3 shows the scheduling and instantiation process for OpenStack Legacy. The requests are submitted to the Nova API using the HTTP protocol (REST API). The Nova API component manages the Initial requests and stores them in the Queue Server. At this point, an authentication phase towards Keystone is required. The next step is the retrieval of the image from Glance, which is required for the creation of virtual resources. At the completion of this step, the Nova Scheduler is involved: this component performs scheduling tasks by taking the requests from the Queue Server, deciding the Compute nodes where the guests should be deployed and sending back its decision to the Nova API (passing through the Queue Server). The components described so far are mapped to the VIM core components, in our model. After receiving the scheduling decision from the Nova Scheduler, the Nova API contacts the Nova Compute node. This component manages the interaction with the specific hypervisors using the proper toolstack and can be mapped (along with Nova Network) to the VIM local components. The VM instantiation phase can be divided in two sub-steps: Network creation and Spawning. Once this task is finished, Nova Compute sends all the necessary information to Libvirt, which manages the spawning process instructing the Xen hypervisor to boot the virtual machine. When the completion of the boot process is confirmed, Nova Compute sends a notification and the Nova API confirms the availability of the new VM. At this point, the machine is ready and started. The above description, reflected in Figure 5-3, is a simplified view of the actual process: for sake of clarity many details have been omitted. For example, the messages exchanged between the components traverse the messaging system (Nova Queue Server, which is not shown), and at each step the system state is serialized in the Nova DB (not shown).

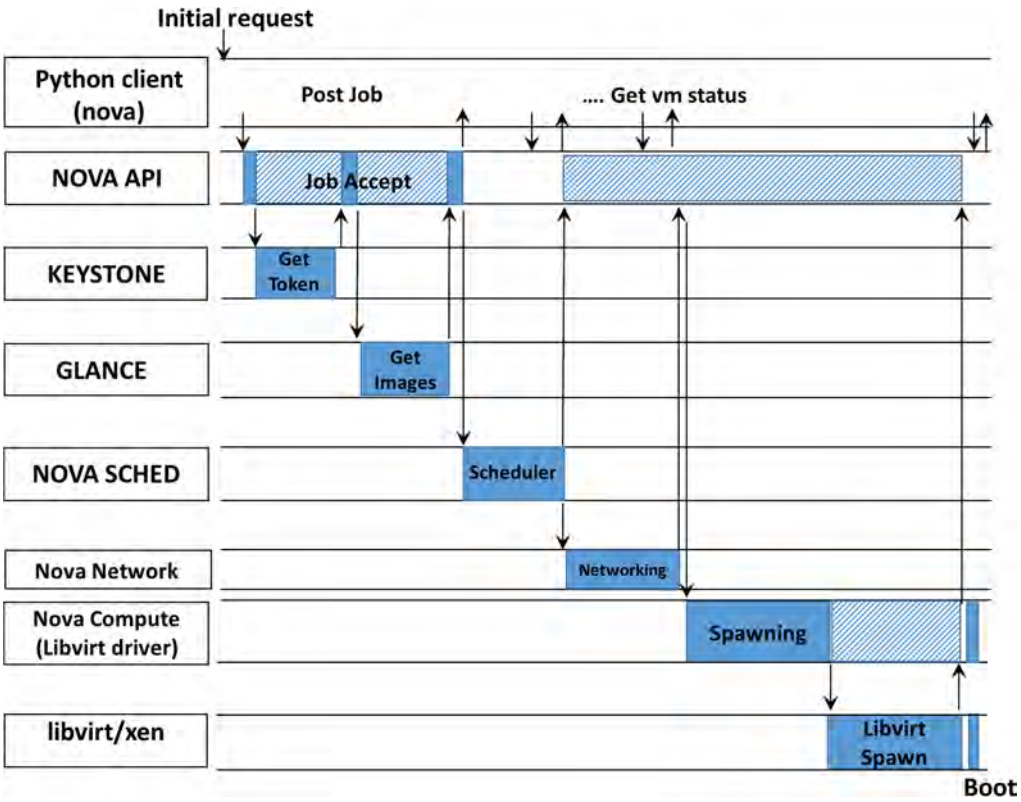


Figure 5-3: VIM instantiation model for OpenStack Legacy

5.1.2 OpenStack

The VM instantiation process in OpenStack is similar to the one described above for OpenStack legacy, but relies on Neutron for the network management. Provisioning a new VM instance involves the interaction between multiple components inside OpenStack. The main components involved in the instantiation process are: Keystone, for authentication and authorization; Nova (with its subcomponents Nova API, Nova Scheduler and Nova Compute), for VM provisioning; Glance, the image store component; Cinder, to provide persistent storage volumes for VM instances (and not employed in the Unikernel scenario); Neutron, which provides virtual networking, and which is split in the two subcomponents Neutron Controller (core component) and Neutron Agent (local component).

Figure 5-4 shows the main phases of the scheduling and instantiation process for OpenStack Nova:

1. The instantiation request is submitted to Nova API using the HTTP protocol (REST API);
2. Keystone authenticates the credentials and generates an auth-token which will be used for sending requests to other components;
3. Nova API sends a request by passing the auth-token to the Glance API to get the VM image details. Glance returns the image URI and its metadata;

4. Nova API sends a request to Nova Scheduler, which finds an appropriate host via filtering and weighting (filter scheduler). Nova Scheduler returns the chosen host ID;
5. Nova API requests a volume for the instance to Cinder, which returns the block storage information;
6. Nova API requests Neutron to configure the network for the instance;
7. Neutron server validates the request and instructs the Neutron Agent to create the network;
8. Nova Compute receives the network information;
9. Nova Compute requests the hypervisor via libvirt to start the virtual machine.

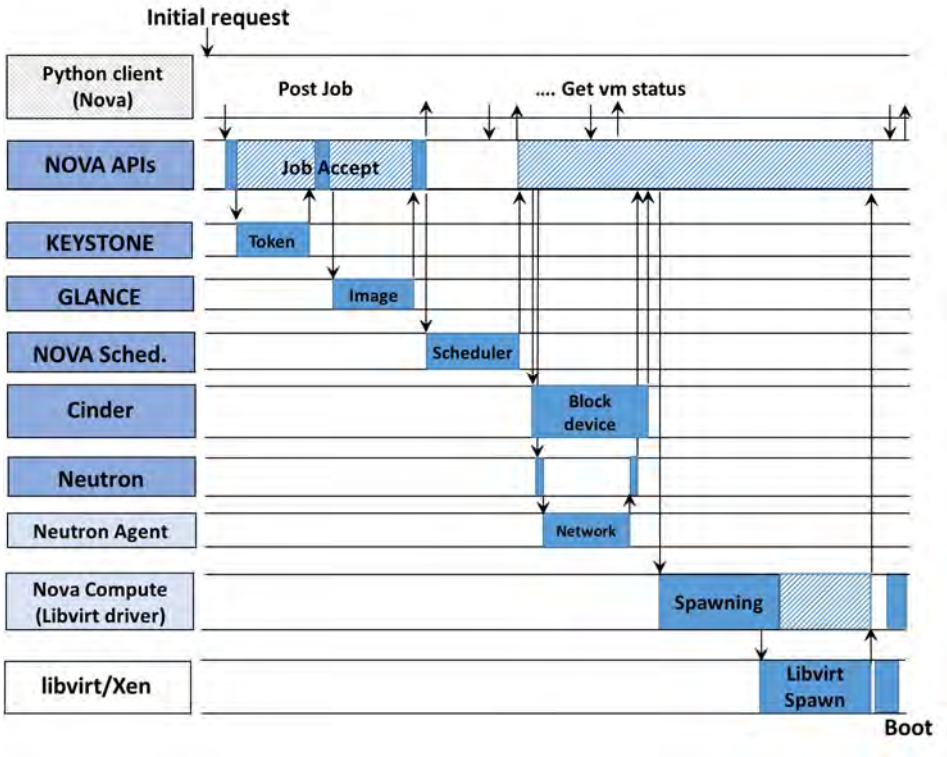


Figure 5-4: VIM instantiation model for OpenStack (with Neutron)

5.2 Modelling Nomad

The scheduling and instantiation process for Nomad is shown in Figure 5-5. According to our model, the Nomad Server is mapped into the VIM core components. It receives the requests for the instantiation of VMs (jobs) through the REST API. Once the job has been accepted and validated, the Server takes the scheduling decision and selects in which Nomad Client node to run the VM. The Server contacts the Client sending an array of job IDs. As a response, the Client provides a subset of IDs which are the ones that will likely be executed in this transaction. The Server acknowledges the

IDs and the Client executes these jobs. The Nomad Client is mapped to the VIM local components and interacts with compute resources/hypervisors. We used XL, the default Xen toolstack, to interface with the local Xen hypervisor. XL is built using libxl and provides a command line interface for guest creation and management. The Client executes these jobs loading the Nomad Xen driver, which takes care of the job execution interacting with the XL toolstack. The instantiation process takes place and once completed the Client notifies the Server about its conclusion. Meanwhile the boot process of the VM starts and continues asynchronously with respect to the Nomad Client.

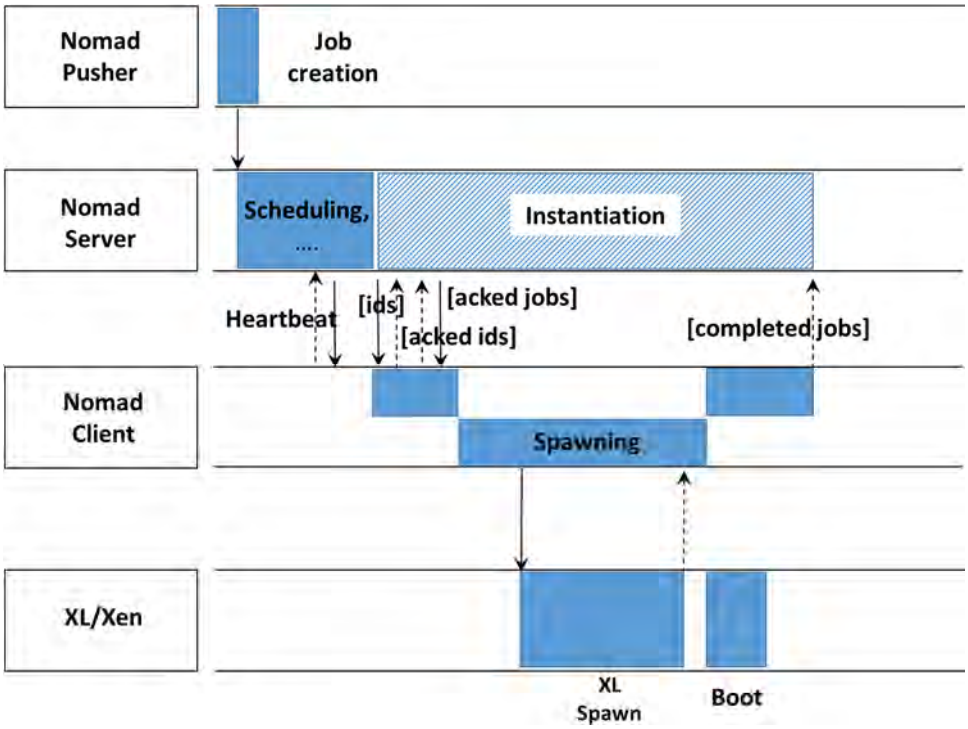


Figure 5-5: VIM instantiation model for Nomad

5.3 Modelling OpenVIM

Borrowing the terminology we have defined in Figure 5-2, OpenVIM core and OpenVIM DB compose the VIM core components. The submission of tasks (i.e. commands and associated descriptors) can be performed through the OpenVIM Client. The requests from the client are forwarded to the core leveraging REST APIs. Each task request is then mapped to a specific image flavor and image metadata. These, together with the status of the Compute nodes (retrieved from the OpenVIM DB) are given in input to the scheduling process which decides the Compute nodes that will handle these requests. The taken decision is written to the database and the OpenVIM Client is notified returning a HTTP 200 OK message. At this point the task is split in a network request and an instantiation request. If the networking part is managed through a compatible SDN controller, the network request is forwarded to a Network thread which takes care of interacting with SDN controller of the

infrastructure in order to perform all the networking related tasks. Otherwise, network request is directly forwarded to libvirt. Instead, the instantiation is performed partially on the controller node and the second part on the local node: OpenVIM core is a multi-thread application where each Compute node is associated to a thread and a work-queue. According to the scheduling decision, the jobs are submitted to the proper work queue and subsequently the spawning process can start. The thread starts this phase by generating the XML description of the instance and submitting it to the OpenVIM locals libvirt daemon running on the chosen compute node. The requests, data and what is needed for the spawning is forwarded to destination leveraging SSH connections. If needed, 3rd party software can be invoked for further configurations (e.g. Open vSwitch). At this point the job is completely offloaded to the Compute resources component. The libvirt daemon creates the instance based on the received XML. After this step, the associated thread on the OpenVIM core calls again the libvirt daemon to instruct it to start the previously-created instance. Finally, the libvirt daemon boots the instance through the Xen Hypervisor and the information on the started instance is updated in the OpenVIM database. The OpenVIM client can obtain up to date information during the VM lifetime by polling its status. Figure 5-6 shows the instantiation process in OpenVIM. For sake of clarity, networking details have been omitted.

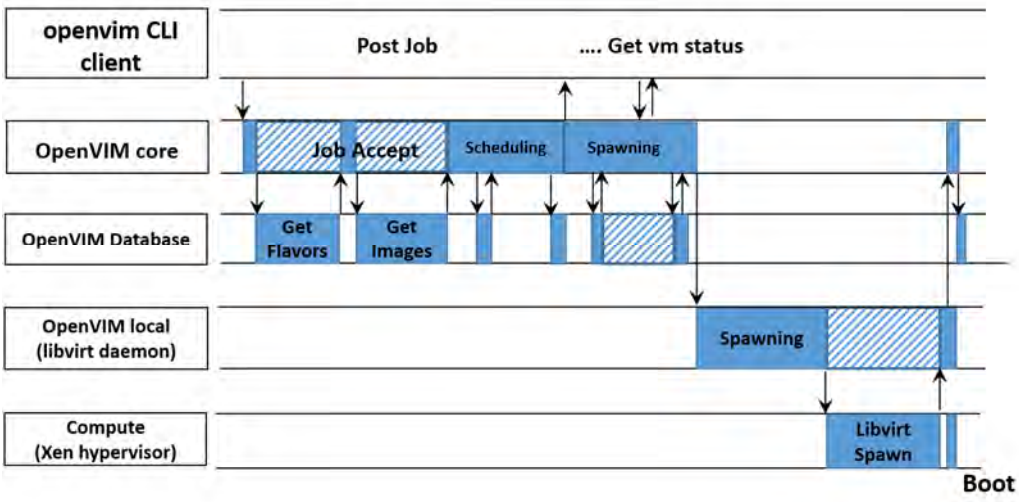


Figure 5-6: VIM Instantiation model for OpenVIM

5.4 VIM Modifications to boot Micro-VNFs

We designed and implemented some modifications in the considered VIMs in order to be able to instantiate Micro-VNFs based on ClickOS using OpenStack, Nomad and OpenVIM. The main reasons for these adaptations lie in the peculiarities of the ClickOS Unikernel Virtual Machines compared to regular VMs. A regular VM can boot its OS from an image or a disk snapshot that can be read from an associated *block device* (disk). The host hypervisor instructs the VM to run the boot loader that


reads the kernel image from the block device. On the other hand, we are interested in ClickOS based Micro-VNFs, provided as a tiny self-contained kernel, without a block device. These VMs need to boot from a so-called *diskless image*. When instantiating such VMs, the host hypervisor reads the kernel image from a file or a repository and directly injects it in the VM memory, meaning that the modifications inside the VIM can be relatively small and affecting only the component interacting with the virtualization platform to boot *diskless images*. We define as *stock* the VIM versions that only include the modifications required to boot the ClickOS VMs. We analysed the time spent by the *stock* VIMs during the different phases of the driver operations. Then, taking into account the results, we proceeded with an optimization trying to remove those functions/operations not directly needed to instantiate Micro-VNFs. We define as *tuned* these optimized versions.

The amount of work to implement the modifications for each VIM is heavily dependent on the specific project: for example for Nomad and OpenVIM it was relatively easy to introduce the support for booting from *diskless images* thanks to their simple architectures and to the flexibility of their *southbound* framework. Instead, it was more difficult to understand how to modify OpenStack, due to its larger codebase, supporting several types of hypervisors, each through more than one toolstack. Our patches to OpenStack, Nomad and OpenVIM are available at [3].

5.5 Experimental results

In order to evaluate the VIM performances in the VM scheduling and instantiation phase, we combined different sources of information. We analysed the message exchanges in order to obtain a coarse information about the beginning and the end of the different phases of the VM instantiation process. The analysis of the messages is a convenient approach as it does not require to understand and modify the source code. We have developed a VIM Message Analyser tool with a Python script and the Scapy library [2]. The VIM Message Analyzer (available at [3]) is capable of analyzing Nova and Nomad message exchanges. For a more detailed breakdown of the timings for specific components or phases, we inserted timestamp logging instructions inside the code of the Nomad Client, Nova Compute nodes as well as the OpenVIM core. We have generated the workload for OpenStack using Rally [4], a well known benchmarking tool. For the generation of the Nomad workload, instead, we have developed the Nomad Pusher tool. It is a utility written in the GO language, which can be employed to programmatically submit jobs to the Nomad Server. For OpenVIM we have simply scripted the OpenVIM command line client operations.

We executed experiments to evaluate the performance of the considered VIMs. We present here two main results: i) the total time needed to instantiate a ClickOS [5] VM (representing a Micro-VNF); ii) the timing breakdown of the Spawning process in Nova and of the Driver execution in Nomad. The first result is based on the VIM Message Analyzer we have developed. The timing breakdown is


obtained with the approach of inserting timestamp loggers in the source code of the VIMs. All results have been obtained by executing a test of 100 replicated runs, in unloaded conditions. Error bars in the figures denote the 95% confidence intervals of the results. In each run, we requested the VIM to instantiate a new ClickOS VM. The VM is deleted before the start of the next run. Our experimental set-up is composed by two hosts with an Intel Xeon 3.40GHz quad-core CPU and 16GB of RAM. One host (hereafter referred to as HostC) is used for the VIM core components, the other host (HostL) for the VIM Local components and the Compute resource. We are using Debian 8.3 operating systems with Xen-enabled v3.16.7 Linux kernels. Both hosts are equipped with two network interfaces at 10 Gb/s: one interface is used as the management interface and the other one for the direct interconnection of the host (data plane network). In order to emulate a third host running OpenStack Rally and Nomad Pusher, we created a separated network namespace using the iproute2 suite in the HostC, then we interconnected this namespace to the data plane network.

5.5.1 OpenStack Legacy Experimental results

For what concerns OpenStack Legacy we run Keystone, Glance, Nova orchestrator, Horizon, and the other components in HostC, while we run Nova Compute and Nova Network in HostL.

With reference to Figure 5-3, we report in Figure 5-7 the measurements of the instantiation process in OpenStack Legacy, separated for each component. The topmost horizontal bar (Stock) refers to the OpenStack version that only includes the modifications to boot the ClickOS VMs. The experiment reports a total time exceeding two seconds which is not adequate for highly dynamic NFV scenarios. Analysing the timing of the execution of the single components, we note that most of the time is spent during the spawning phase while the other components account for around 0.5 seconds. In Figure 5-8 (topmost horizontal bar), we show the details of the spawning phase which is split in three phases: 1) Create image, 2) Generate XML and 3) Libvirt spawn. The first two are executed by the Nova Libvirt driver and the last one is executed by the underlying Libvirt layer. The Nova Libvirt driver also executes some network configuration steps which are not shown, as they happen in parallel with the Create image phase, which always terminates after the network configuration has finished. By analyzing the timing of the stock version, we can see that the Create image is the slowest step with a duration of about 1 second. This step includes operations like the creation of log files and the creation of the folders to store Glance images. The original OS image (the one retrieved from Glance) is re-sized in order to meet user requirements (the so called flavors in OpenStack's jargon). Moreover, if it is required, the swap memory or the ephemeral storage are created and finally some configuration parameters are injected into the image (e.g. SSH key-pair, network interface configuration).

The "Generate XML" and "Libvirt spawn" steps introduce a total delay of 0.4 seconds, but optimizing these steps is non-trivial as all the performed operations cannot be skipped or downsized. Indeed,

during the Generate XML step, the configuration options of the guest domain are retrieved and then used to build the guest domain description (the XML file given in input to Libvirt). For instance, options like the number of CPUs and CPU pinning are inserted in the XML file. Once this step is over, the libxl API is invoked in order to boot the VM. When the API returns, the VM is considered spawned, terminating the instantiation process. In this test we are not considering the whole boot time of the VM, as this is independent from the VIM operations, and thus out of the scope of this work. The considered spawning time measures only the time needed to create the Xen guest domain.

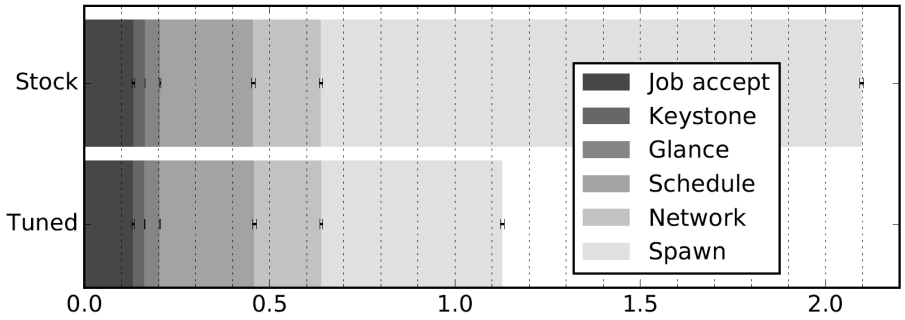


Figure 5-7: ClickOS (micro-NFV) instantiation time breakdown on OpenStack legacy

The performance measurements for Nova results reported above show that most of the instantiation time is spent on the *Create image* operation. The ``tuned'' horizontal bar in Figure 5-7 reports the obtained result. The reduction of the instantiation time highlights that the advantages of using Unikernels with respect to a full fledged OS are not only the shorter boot times: specific VIM optimizations can be performed due to their characteristics. We are using a tiny diskless VM, which means we can skip most of the actions performed during the image creation step. For example, we do not need to resize the image in order to match the flavour disk size. Moreover, unikernels provide only the functionality needed to implement a specific VNF, so it is unlikely to have an SSH server running on it, hence SSH key-pairs configuration is not needed.

Furthermore, if a Micro-VNF does not require a full IP stack, the injection of the network configuration in the image is useless. Indeed, some Micro-VNFs need only to have configured the bridge on which it has to be attached. Implementing these optimizations, we are able to reduce the spawning time down to 0.4 s (Figure 5-8, bottom bar) obtaining a reduction of about 70% compared to the stock version of OpenStack. Looking at the overall instantiation time, the relative reduction (Figure 5-8, bottom bar) is about 45%, down to 1.15 s from 2.1 s for the stock OpenStack.

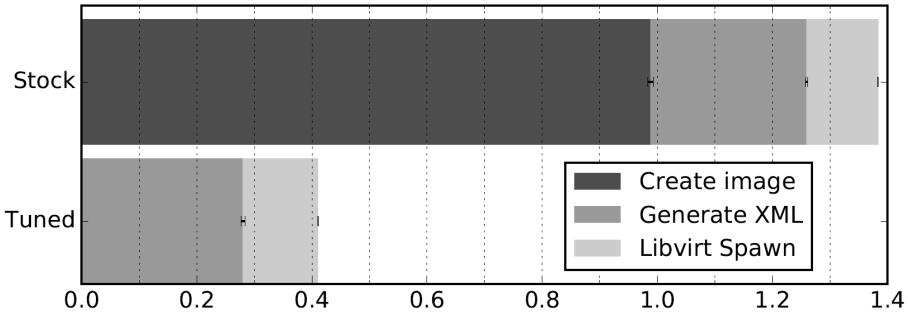


Figure 5-8: ClickOS spawning time breakdown on Nova-compute (OpenStack legacy)

5.5.2 OpenStack Experimental results

Concerning Openstack, the setup is similar to the one of Openstack Legacy described above, but we run Neutron instead of Nova network. With reference to Figure 5-4, we report in Figure 5-9 and Figure 5-10 the measurements of the instantiation process and the spawning time breakdown. The block device and neutron operations are started at different times, but run in parallel.

The tuned version does not include the operations related to block device mapping, due to ClickOS machines being diskless. However, we could not completely skip the “create image” phase, as this is needed to create the ephemeral disk which contains the root of the operating system. We keep a local copy of the image for each instance, so that VMs which are running are not affected by changes in the original image. The removal of the block device operations has also an impact on the spawning phase, removing the need to wait for block device information. The “generate xml” phase is also affected, as less operations have to be performed.

We did not implement optimizations on neutron, as it is already optimized and makes extensively use of caching. Caching is also employed in other operations: image download, block device creation/reuse, network creation.

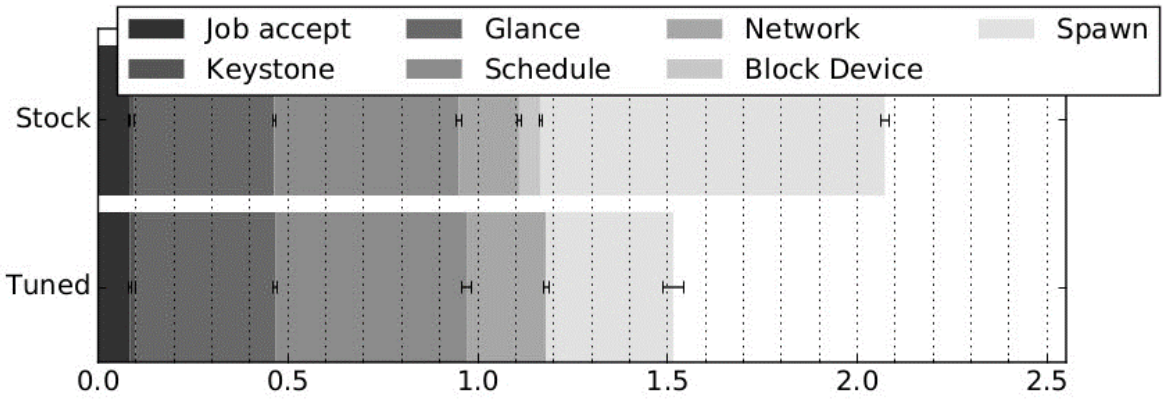


Figure 5-9: ClickOS instantiation time breakdown on OpenStack

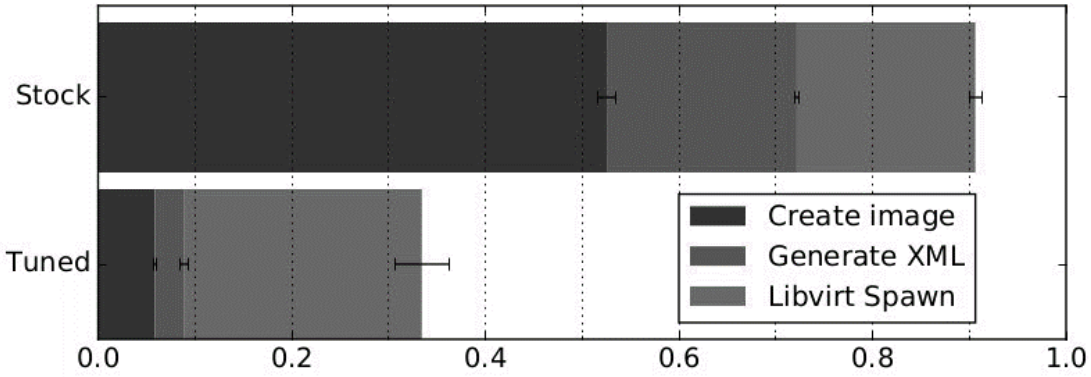


Figure 5-10: ClickOS spawning time breakdown on Nova-compute

5.5.3 Nomad Experimental results

According to the Nomad architecture, the minimal deployment includes two nodes. Therefore, we have deployed a Nomad Server, which performs the scheduling tasks, in HostC and a Nomad Client, which is responsible for the instantiation of virtual machines, in HostL. The Nomad Pusher runs in HostC but in a different network namespace. We identified two major steps in the breakdown of the instantiation process: Scheduling and Instantiation. The topmost horizontal bar in Figure 5-11 reports the results of the performance evaluation for the stock Nomad. The total instantiation time is much lower than the one obtained for OpenStack. This result is not surprising: Nomad is a minimalistic VIM providing only what is strictly needed to schedule the execution of virtual resources and to instantiate them. Looking at the details, the Scheduling process is very light-weight, with a total run time of about 50 ms. The biggest component in the instantiation time is the spawning process which is executed by the XenDriver. Diving in the driver operations, we identified 4 major steps: Download artifact, Init Environment, Spawn, and Clean, as reported in Figure 5-12. In the first step, Nomad tries to download the artifacts specified by the job. For a Xen job, the Client is required to download the configuration file describing the guest and the image to load. This part adds a delay of about 40 ms and can be optimized or entirely skipped. Init Environment and Clean introduce a low delay (around 20 ms) and are interrelated: the former initializes data structures, creates log files and folders for the Command executor, the latter cleans up the data structures once the command execution is finished. The XL spawn is the step which takes longer but by studying the source code we found no room to implement further optimizations: indeed, the total spawning measured time is around 100 ms. Considering a light overhead introduced by the Command executor the time is very similar to the what we obtain by running directly the XL toolstack. The overall duration of the spawning phase is 160 ms, lower that the 280 ms for the instantiation phase reported in Figure 5-11. This is due to the notification mechanism from the client towards the server. It uses a lazy approach for communicating the end of

the scheduled jobs: when the message is ready to be sent, the client waits for a timer expiration to attempt to aggregate more notifications in a single message. This means that the instantiation time with Nomad is actually shorter than the one shown in Figure 5-11.

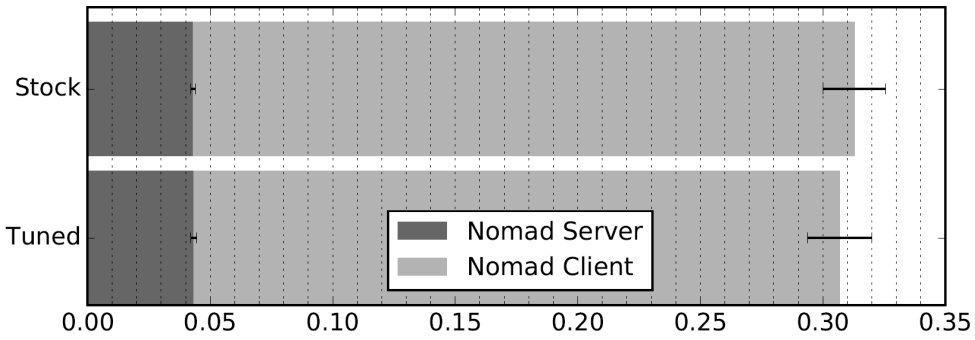


Figure 5-11: ClickOS: (micro-NFV) instantiation time breakdown on Nomad

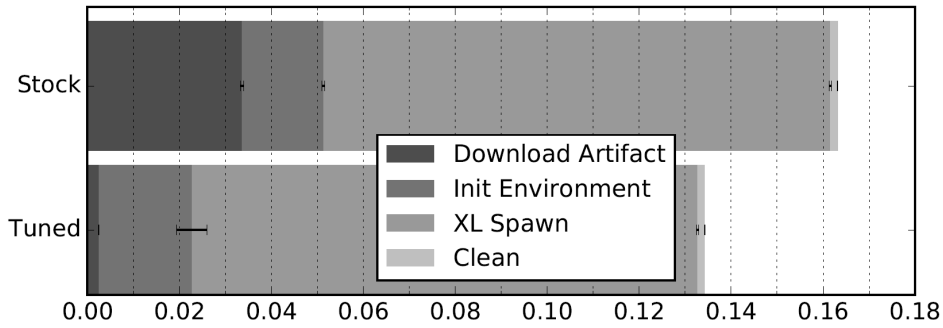


Figure 5-12: ClickOS spawning time breakdown on Nomad

5.5.4 OpenVIM Experimental Results

We deploy *OpenVIM Core* components and *OpenVIM DB* on HostC, while HostL has been employed as compute node. We use the *development mode* for OpenVIM in our experiments, which requires less resources and disables some functionalities, like EPA features, passthrough or SR-IOV interfaces support which are not required for the scope of our work. We also configured HostC as compute node noticing no performance degradation of the system.

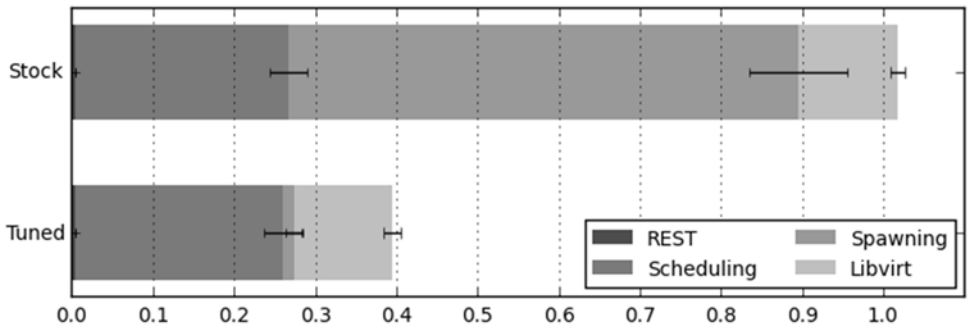


Figure 5-13 OpenVIM instantiation time breakdown

Figure 5-13 (top bar) shows the measurements related to the instantiation process for OpenVIM *stock* divided in *REST*, *Scheduling*, *Spawning* and *libvirt* phases. With reference to the model depicted in Figure 5-2, *REST*, *Scheduling* and part of the *Spawning* are mapped with *OpenVIM Core* components. The second part of the *Spawning* phase is spent on the compute node. Finally, for the *libvirt* phase there is a one-to-one mapping. Summing up all the components we have obtained a total instantiation time of about 1 s.

We have started the optimization process by excluding from the instantiation phase all the operations which are not required for the ClickOS boot but performed in any case by OpenVIM. During the spawning process, the VM interface is reset sending *ifdown/ifup* via SSH. This operation is totally irrelevant for ClickOS and it can be very time consuming since the spawning is blocked waiting for the interfaces coming down/up. Another operation we have optimized, it is the full-copy of the VM image: by default a copy is performed to move the image in libvirt folder of the Compute node. We have removed totally this overhead maintaining a local copy of the image and avoiding the copy each time. Another source of overhead is *libvirt*, which by default operates using SSH. Even if all its operations are small, SSH can introduce a certain amount of overhead. Thus, as further optimization, we have enabled TCP as transport layer for the *libvirt* communication channel instead of SSH. Finally, we have optimized also the generation of the XML because for the *stock* version we use the default behavior of *libvirt* with Xen. We have removed all unnecessary parts like the graphics, serial ports and we have avoided also the allocation of the CD-ROM virtual device and of the hard disk.

Then, we have focused our attention on the *OpenVIM Core* components. In particular, the management of work queues has attracted our attention: for each Compute node there is a worker thread which manages a working queue where the instantiation requests are enqueued as output of the scheduling. By default, the worker checks the queue each second. In a deployment with a few numbers of Compute nodes, it results to be too conservative. Setting to zero this refresh time has led to a considerable reduction of the instantiation process, but at the same time has increased the load on the CPU. With our deployment we have found a good compromise setting up the polling interval at 1 ms (in this configuration, we have measured a load of 2% in the HostC where OpenVIM Core and

OpenVIM DB run). Results of our optimizations are shown in Figure 5-13. With the above modifications we are able to measure times of around 400 msec. Looking at the overall instantiation time, the relative reduction is about 65%. The *Spawning* phase becomes negligible, while the other phases are mostly unchanged. Figure 5-14 shows the breakdown of the spawning phase for OpenVIM tuned.

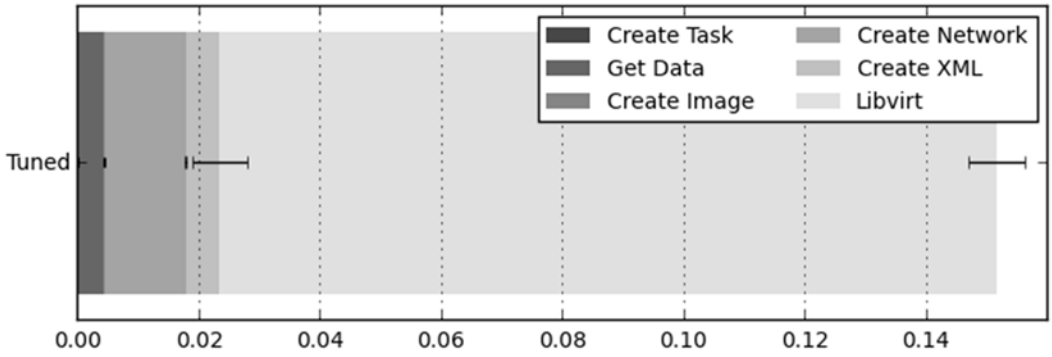


Figure 5-14 OpenVIM tuned spawning time breakdown

5.6 References for section 5

[1] (2004) Xen Project. [Online]. http://www.xenproject.org

[2] Scapy project. [Online]. http://www.secdev.org/projects/scapy/

[3] VIM tuning and evaluation tools. [Online]. https://github.com/superfluidity/vim-tuning-and-eval-tools

[4] Openstack rally. [Online]. https://wiki.openstack.org/wiki/Rally

[5] Joao Martins et al., "ClickOS and the art of network function virtualization," in 11th USENIX Conference on Networked Systems Design and Implementation, 2014, pp. 459-473.

6 Reducing Memory Use of Docker-based VNFs: NetScaler CPX Case Study

6.1 Introduction

One key use case of Superfluidity is to move data-plane network functions from the Mobile Core and S/Gi-LAN domains of the 5G architecture to the Multi-access Edge Computing (MEC) domain. In the previous sections of this deliverable we have explored how lightweight VMs (LightVM) and respective network architectures (HyperNF) deliver this vision. We also explored how CPUs can be efficiently shared, in the scenario of multiple network functions competing for packet processing.

Based on our experience, stateful network functions, such as TCP optimization, video delivery, video optimization, etc. involve maintaining IP flow or TCP session state and storing network packets in application buffers. As we have seen, modern hypervisors, but also container engines, are able to share CPU resources with very low overhead. Moreover, accelerated networking, as also evolved further by Superfluidity (HyperNF, FastClick), optimizes network I/O.

As a consequence, stateful network functions, such as the above, tend to become memory bound, in the sense that their ability to support high total throughput and large numbers of concurrent network connections or flows is limited by the available memory. This will be further exacerbated by the significantly increased bandwidth each broadband 5G connection will be able to consume.

These are the reasons why, in this section, we focus on the need of minimizing memory utilisation, which is vitally important for maximising the number of network function instances that the potentially small memory (RAM) resources of MEC hardware will be able to accommodate. This also maximizes the aggregate throughput and number of concurrent users the will be able to support.

As a case study, we have picked NetScaler CPX [1], a variation of NetScaler ADC that is available as a Docker container. NetScaler CPX supports the majority of functions of NetScaler ADC, including stateful network functions, such as split-TCP Performance Enhancing Proxy (PEP).

6.2 NetScaler CPX

Citrix NetScaler CPX is a container-based application delivery controller that can be provisioned on a Docker host.

NetScaler CPX allows to:

- Leverage Docker engine capabilities and NetScaler load balancing and traffic management features for container-based applications.
- Deploy one or more NetScaler CPX instances as standalone instances on a Docker host. A NetScaler CPX instance provides throughput of up to 1 Gbps.

NetScaler CPX Express is the free developer version that supports up to 20 Mbps and 250 SSL connections. NetScaler CPX Express supports most of the CPX feature set, except TCP optimization and L7 DDoS.

The features and capabilities of NetScaler CPX that differentiate it from alternative options are:

High concurrency: NetScaler CPX is optimized for high TCP session count and high HTTP response-per-second performance.

Application-savvy load balancing: Unlike Round Robin and hash-based load balancing, NetScaler CPX algorithms take into account the actual load and send traffic to the least-loaded containerized app. You can integrate Layer 7 policies required for deploying a service, and you can apply policies to both ingress and egress traffic. These policies can redirect a connection based on HTTP data, rewrite an HTTP request or response, and switch a connection to a specific app based on HTTP payload, HTTP heater, or IP address range. You can combine load balancing with DNS service groups or use NetScaler CPX as a DNS service.

Service discovery and auto-configuration: NetScaler has built-in support for service discovery. NetScaler Management and Analytics System (MAS) ties into Mesos, Marathon, and Kubernetes, and acts as a CPX controller. It auto-configures NetScaler CPX to changes in the app topology. You can automate the configuration of NetScaler CPX to load-balance any type of app via Stylebooks—declarative templates that reside in NetScaler MAS.

Integrated management and analytics: NetScaler Management and Analytics System (MAS) acts a controller, orchestration point, and analytics platform for NetScaler CPX. You can manage multiple NetScaler CPXs at scale and get immediate visibility to the health of apps and microservices. NetScaler CPX streams its internal counters and transaction logs to NetScaler MAS. Several built-in insights provide detailed reporting on user experience, security, and SSL. You can manage NetScaler CPXs as a fleet through NetScaler MAS, with automated per-instance configuration and SSL certificates management. With NetScaler MAS as a controller, you can tie in to orchestration systems, SDN controllers, or overlay network solutions to insert NetScaler CPX into routed or host-network topologies.

DDoS mitigation: NetScaler CPX has proven algorithms to manage attacks that exploit weaknesses in TCP, HTTP, and DNS. By using NetScaler CPX for all app-to-app traffic, you can protect apps from DDoS attacks from the outside and inside.

SSL offload: Deploy SSL/TLS with the most secure ciphers, such as elliptical curve cryptography. Mandate specific ciphers across all load balancers by using NetScaler Management and Analytics System SSL certificate management tools. NetScaler CPX can aggregate connections to backend servers to optimize performance and secure the connection to the backend apps by re-encrypting connections. You can push your apps to the cloud or separate production from pre-production traffic with confidence.

More information on the capabilities of NetScaler CPX are available in the respective data sheet [2].

6.3 Enhancements

Towards reducing the memory requirements of NetScaler CPX, we implemented the enhancements below:

- We changed the build process of the NetScaler CPX Docker container to use a specialized Linux base image. We optimized the base image, to only contain the Linux modules, libraries and packages required for NetScaler CPX. The result was a significant reduction of the NetScaler CPX Docker container image size (currently to 367MB), as well as of the base memory requirements.

- We improved how NetScaler CPX accounts for available memory, when it is allocating data structures for packet processing. In the case of NetScaler MPX and VPX, using total available memory for memory accounting is perfectly appropriate, since memory size depends on the hardware specs (MPX) or the resources reserved by the hypervisor (VPX). However, in the case of NetScaler CPX, using total memory is not appropriate, since memory available depends on the memory allocated by the Docker engine, as configured by the customer. We thus enhanced NetScaler CPX to use allocated memory, instead of total memory, for memory accounting.

- We noticed that memory may be allocated for features and capabilities that NetScaler CPX does not support. Not allocating memory for those was, apparently, a straightforward improvement.

- We identified additional improvements, when NetScaler CPX operates under conditions of low memory. Specifically, we noticed that operational tools, such as viewing performance statistics, generating connection logging output for troubleshooting, etc. didn't work very well under such

conditions. Even though these were not exactly relevant to our work herein, we have added these findings as potential enhancements on the NetScaler CPX backlog.

6.4 Results

As a result of the combination of enhancements above, the recommended minimum memory requirement of NetScaler CPX was decreased from 2GB to 1GB. This effectively doubles the density of container instances that can be created on any given hardware configuration. Also, they enable applying more complex service configurations to a NetScaler CPX instance, which weren't possible before. Finally, they allow for allocating more memory for packet/flow processing, which increases the throughput of memory-bound stateful network functions, as we outlined in paragraph 6.1.

6.5 References for section 6

[1] NetScaler CPX and CPX Express, https://www.citrix.com/products/netscaler-adc/cpx-express.html, Citrix Systems Inc.

[2] NetScaper CPX Data Sheet, https://www.citrix.com/products/netscaler-adc/resources/netscaler-cpx-data-sheet.html, Citrix Systems Inc.

7 Dataflow Engine for C-RAN Baseband Processing

The results reported in this section have not yet been published, but a paper is in preparation and will contain portions of the text below.

7.1 Introduction

The previous sections have covered most aspects of the Superfluid platform, mostly targeting fix wire and higher layer packet processing issues. In this section we now turn to the issue of efficiently running a set of parallel baseband signal processing slices by defining a C-RAN dataflow abstraction and associated runtime system. The framework includes i) flexible dataflow baseband applications (T4.3) ii) dataflow API and iii) dataflow engine DFE implementing the methods for efficient mapping of dataflow applications to computation resources (T5.1/T5.2). In order to address various targets and diverse performance requirements, the dataflow framework supports COTS x86 servers, ARMv8 micro-servers as well as application-specific baseband MPSoCs (Tomahawk).

The next generation cellular standards are at the verge of deployment. Again, the maximum achievable data rate increases along “the wireless roadmap” [1], now to approach 1Gb/s at its market introduction, and growing to 10Gb/s over the coming years. Key part of 5G is the support for applications demanding high data rates so-called Enhanced Mobile Broadband (eMBB) [2][3]. However, 5G not only requires a linear-scale increase in required data rates and therefore processing power over 4G. Also new very low data-rate modes as NB-IoT (narrow-band internet-of-things) [2] have been standardized. For specific applications, the minimum data rate has been lowered to 100kb/s and below.

As it is very inefficient to run these low-end modem versions on a platform customized for 1-10Gb/s, a new scalability challenge arises: one must address 5 orders of magnitude in data rate scalability. As the required processing per bit at high data rates is about 10x larger due to MIMO than at low data rates, this translates into a scalability requirement of 6 orders of magnitude in signal processing power.

In addition, 5G will have a URLLC mode (ultra-reliable low latency communication) which requires very short latencies of processing in the order of 1ms and below, whereas NB-IoT requires in the order of 1s latency only. Hence, the signal processing platform must be energy efficient for very different corner cases of latency as well. Moreover, as URLLC requires reliability, i.e. the connection to the cellular network must not be lost, the platform must also run reliably in this mode of operation as well.

Finally, 5G introduces a new concept named network slicing. As the communication requirements defined by the application differ dramatically in data rate, reliability, and latency, each application may negotiate its performance parameter-set to adhere to its specific requirements [3][4]. The impact for a modem is that every application running in parallel can require opening a new slice of its own. 5G challenges current thinking of modem design for cellular in a big way. We need a new level of adaptability and have to cope with a plurality of different slices with very differing signal processing requirements. Not only must the platform be adaptable to scale, but also must manage to serve all slices running in parallel and manage the system of signal processing for very differing parallel communications links.

7.2 Baseband System Architecture

Depending on the application scenario, 5G baseband subsystem have to cope with the huge range of throughput and latency requirements. However, this poses the challenge on system management to dynamically estimate the required performance level and based on this, to decide on optimum system arrangement in terms of required processing elements (PE), communication channels and task and buffer placement. The principle of dynamic management of baseband unit is illustrated in Figure . Based on the application requirements and the channel quality signaled to the base station (BS), the optimum transmission mode is selected for user application slice. Based on this, baseband algorithm structure is generated at BS and system component i.e. processing elements, memory, communication channels are allocated and configured to specific performance level. Algorithm tasks are mapped and scheduled on PEs according to the application-specific TTI deadline.

Figure 7-2 sketches the principle of baseband function configuration and deployment introduced within the Superfluidity project. Based on application slice specifications the baseband model is generated as dataflow graph, reflecting application functionality and structure. The next step is to perform a graph transformation for model optimization and complexity reduction. Task clustering reduces the structural graph complexity that decreases the task management overhead. Optimized dataflow graphs are analyzed by CoreManager for critical paths, parallelism profile, deadlines, and based on this a task schedule is computed in conjunction with the allocation and configuration of required system resources (i.e. PEs, buffers, dataflow channels).

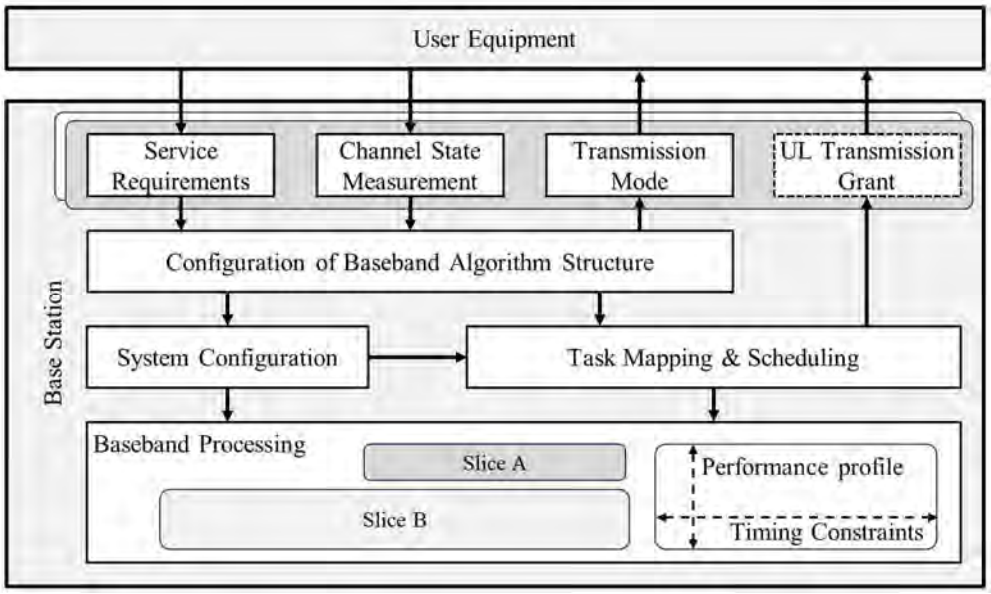


Figure 7-1: The principle of dynamic baseband slice configuration according to application requirements and channel characteristics.

CoreManger enables implementing different strategies for resource sharing, i.e. processing resources can be shared by all slices, or for each slice an exclusive access to specific resources is granted. Moreover, CoreManager can virtually create HW slices i.e. the set of resources and configurations allocated to process specific application slice. In general it supports spatial, temporal and spatio-temporal resource allocation and slice separation. The principle of HW slicing is illustrated in Figure 7-3.

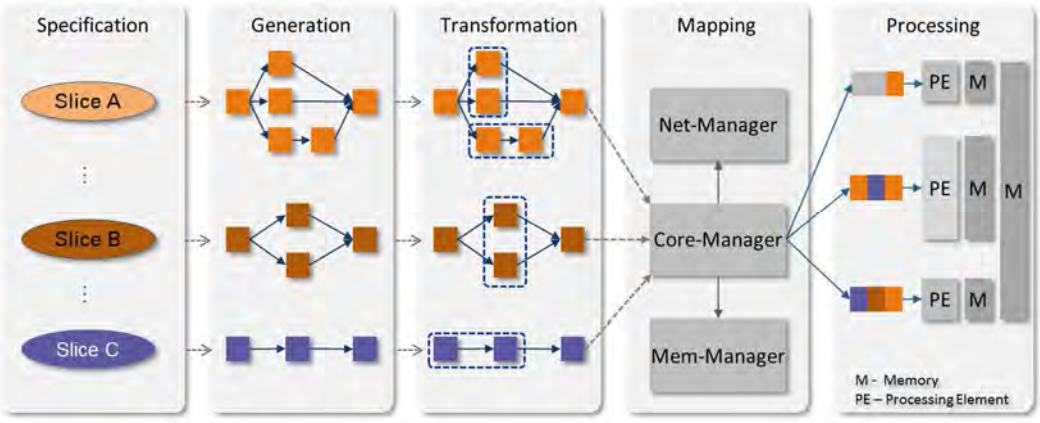


Figure 7-2: The principle of slicing of baseband applications and the design stages for application deployment.

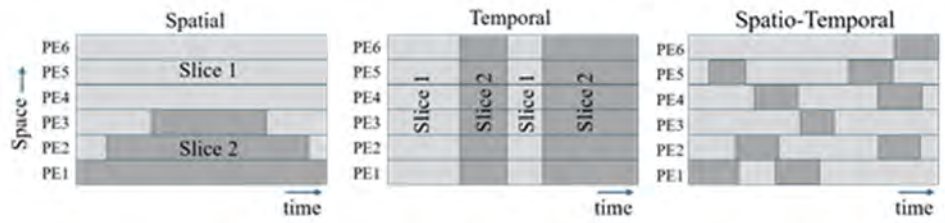


Figure 7-3: Principle of HW slicing using CoreManager. CoreManager (re)allocates dynamically resources according to performance profile and deadline requirements of specific application slice following specific allocation strategy: Spatial-allocation (left), temporal-allocation (middle) and combined spatio-temporal-allocation (right).

The dataflow framework with mentioned capabilities has been developed within Superfluidity project. However, while the research regarding design and transformation of flexible baseband application is part of T4.3 and deliverable D4.3 [7], this work package and deliverable addresses the dataflow engine runtime, particularly, the management unit CoreManager solving the problem of resource allocation and task mapping to computation resources. Various dataflow engine architectures have been investigated during the project enabling so a trade-off in terms of performance and flexibility:

- Multi-threaded architecture that make use of POSIX threads for implementing engine components within single process;
- Multi-process architecture implementing the components of dataflow engine as a set of communicating processes. This architecture allows seamless virtualization of system components by instantiating processes within dedicated VM.
- Multi-processor architecture – that implements components of dataflow engine using dedicated processors. Moreover, dedicated instruction set has been implemented in order to accelerate the resource allocation and scheduling algorithms of CoreManager [8].

While pure software solutions (first two items) are suited for low to moderate throughput base station, the HW accelerated approach is inevitable for cutting edge 5G high bandwidth, high throughput scenarios. It is worth to mention that all proposed approaches provide almost same capability and functionality, moreover, the same dataflow API is adopted in order to allow seamless application portability and adaptation to target platform. Superfluidity primarily focused on SW approaches on COTS server platforms, however, we also ported the Superfluidity dataflow software stack to baseband multi-processor system Tomahawk and demonstrated 5G capability [8].

7.3 Dataflow Engine

The Superfluidity dataflow application model defines (i) slices i.e. the components of application and (ii) tasks i.e. the algorithmic signal processing kernels. Each slice represents dataflow graph that is the composition of tasks. The Dataflow engine (DFE) adopts the Khan Process Network model i.e. multiple parallel processes communicate using message-queues (Figure 7-4). DFE comprises three control components: Application controller, SliceManager and CoreManager. SliceManager schedules the slices to available slice processors (SP). Each slice generates dynamically dataflow graph of associated baseband function and dispatches it to the CoreManager. CoreManager schedules the dataflow graphs of multiple slices to shared processing elements (PE) according to specific slicing strategy (Figure 7-3). The principle of DFE pipelined processing is illustrated in Figure 7-5.

The key performance indicator of dataflow engine is the throughput/latency of message-queues. In order to identify the DFE limitations, we developed specific benchmarks implementing queue with locking and lock-less synchronization mechanisms and analyzed the queue performance. For Xeon server and single-producer single-consumer scenario, the blocking queue approaches throughput of 3mil. 64-bit messages/s and latency below 400ns/message. In contrast to this, the performance of lock-less queue improved by factor of two. These figures indicate the limitations of dataflow engine in terms of maximum task throughput as well as task management overhead. E.g. assuming for simplicity LTE 1ms TTI deadline constraints for slice processing, the maximum 6000 task can by processed by CoreManager. However, as documented in report D4.3 [7], the number of tasks may exceed 10000. Due to this limitation, we introduced in the Superfluidity the concept of task-clusters i.e. the logical aggregation of multiple simple tasks representing a super-task. Using this method, the structural complexity of task graphs can be reasonably reduced.

In order to analyze CoreManager control loop overhead i.e. the latency of task scheduling, dispatching to PE and receiving completion information, we developed simple benchmark comprising a chain of dependent tasks and assume CoreManager subsystem according to Figure 7-6 with blocking queues. The measured latency as time series and histogram is illustrated in Figure 7-7. As expected, the control loop latency approaches in average 700ns that is about twice the delay of the queue.

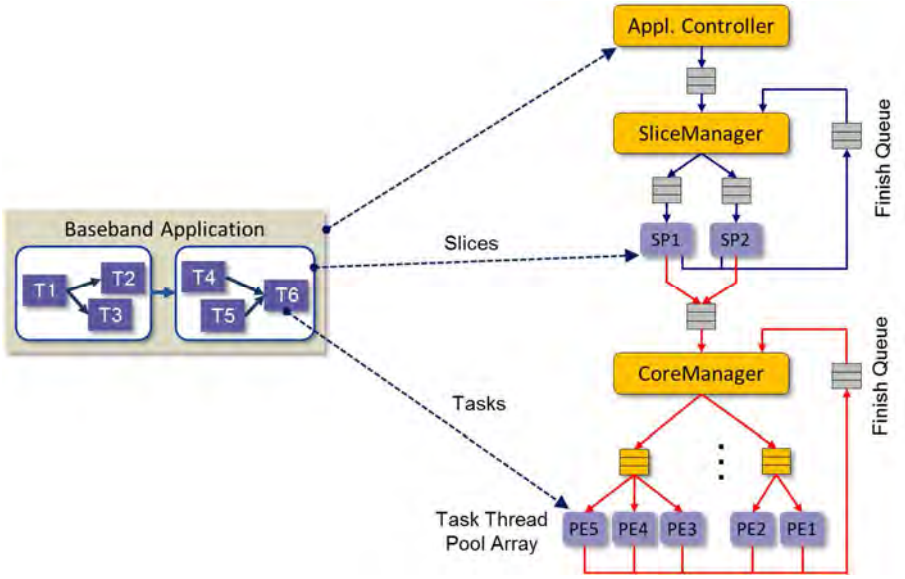


Figure 7-4: The principle of dataflow engine comprising Application controller, SliceManager and CoreManager components. Slice Processors (SP) and processing elements (PE) are dedicated to slice and task processing, repectively.

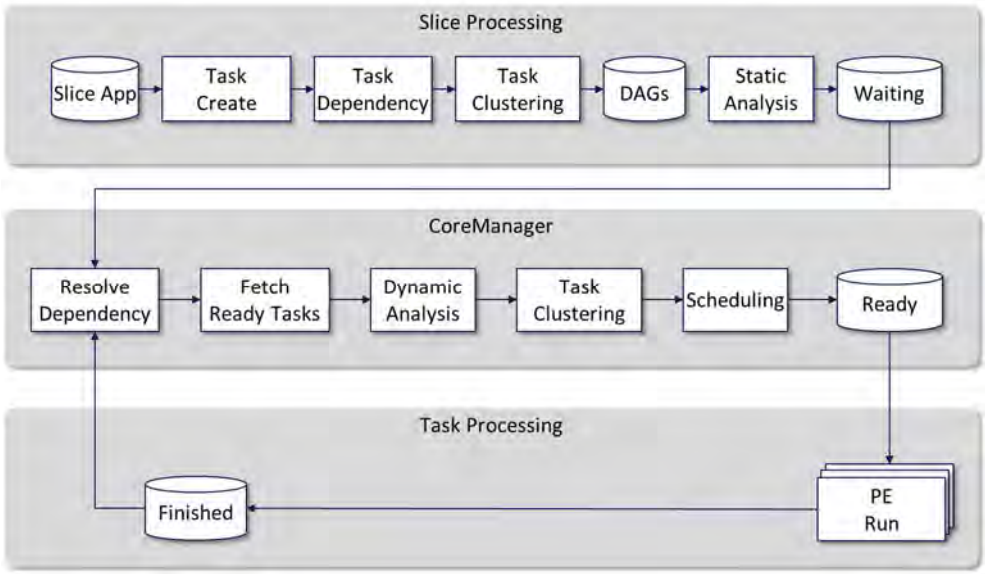


Figure 7-5: Principle of dataflow engine pipelined processing comprising slice processing, task scheduling (CoreManager) and Task processing steps.

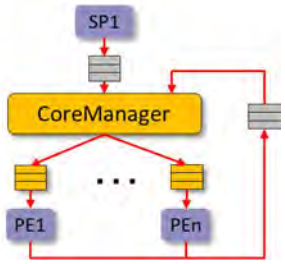


Figure 7-6: CoreManager subsystem used for the measurement of control loop latency.

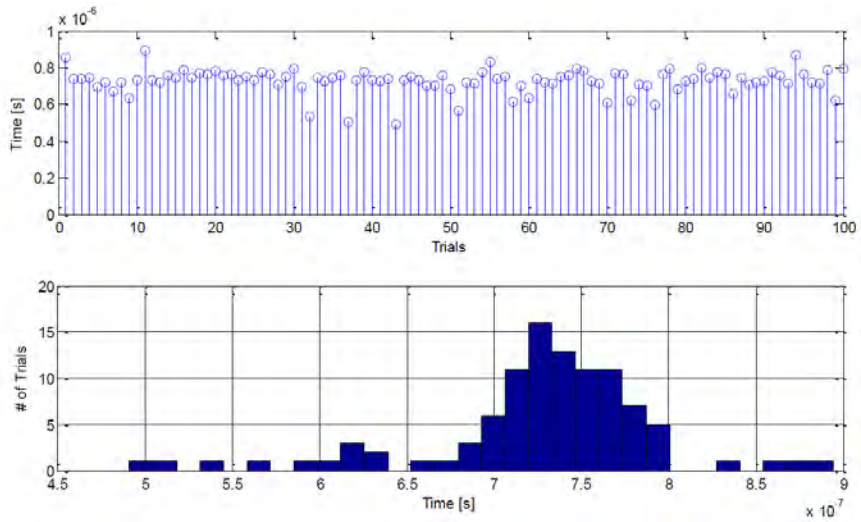


Figure 7-7: The latency measurement of CoreManager control loop.

7.3.1 DFE Performance Analysis

For the execution of dataflow applications, the different performance indicators are considered. One is the overall execution time of a program that runs on a given platform. If the program complexity (e.g. number of task executions, processed data per task) is changing, the execution time of the program should scale accordingly. Moreover, the jitter should be minimized i.e. the execution time of the same program should be almost the same for multiple runs. In other words, the variance of the execution time should be minimal. Especially in the area of signal processing a minimum variance is vital because most applications have to fulfil critical timing constrains.

Apart from the scalability and the variance in the program execution time a third important factor is associated with the task life cycle within the program. The life cycle is comprised of the task creation, the scheduling within the dataflow engine and the task execution. The variance of this time span should be as minimal as possible for equal tasks within a program and moreover, between multiple

program executions. Again, this requirement is vital for software and algorithmic engineers. In order to design a signal processing system with deterministic timing behaviour, it is important to know the worst case (maximum) execution time of each task.

For the purpose of testing the aforementioned determinism, a data flow benchmark was written comprising two tasks i.e. a producer and a consumer. The producer task generates the data which are consumed by the second task. The amount of data D_t produced by a single task can vary. Furthermore, the workload within each task is directly proportional to D_t. A program run is then comprised of a given number N_t of these producer-consumer pairs (Figure 7-8). Thus the total number of executed tasks in a program is given by $N_t \times 2$. Furthermore the producer task is restricted to run on one processing element (PE1) and the consumer can only run on another processing element (PE2).

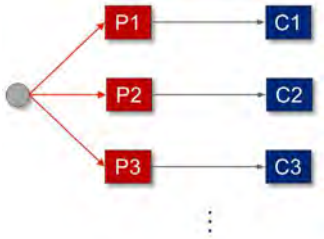


Figure 7-8: Dataflow benchmark for dataflow engine performance analysis.

Figure 7-9 shows the execution time over the workload D_t for a varying number of tasks N_t. Hereby, each data point depicts the mean value for the execution time of 50 consecutive program runs. It can be observed that there is a linear dependency of N_t and the execution time.

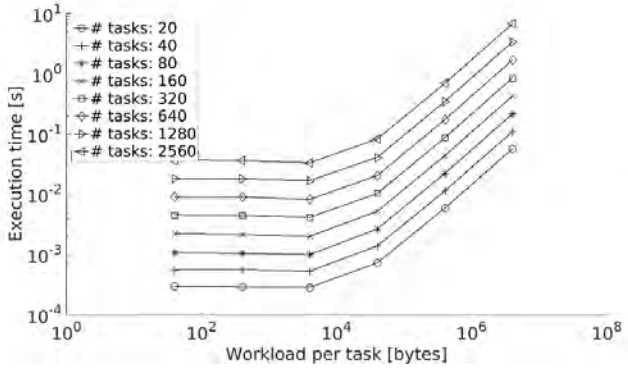


Figure 7-9: Average execution time of test program over the workload for a single task.

For doubling the work amount, the dataflow engine needs twice as much time to finish the program. Secondly, it is observed that the execution time only begins to increases over a certain workload threshold. This is because of the scheduling overhead for each task which becomes dominant with decreasing workload (i.e. small D_t). Under the mentioned threshold, the execution time is not anymore dependant on the actual workload but pureliy driven by the overhead.

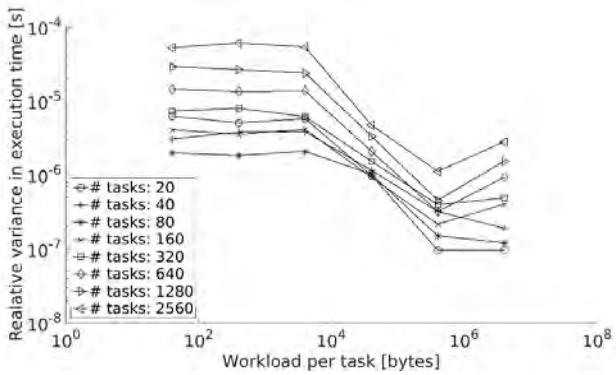


Figure 7-10: Variance of execution time scaled with the mean value of execution time

Figure 7-10 shows the second performance indicator, the jitter i.e. the variance of execution time normalized with the expected value of execution time. Again, the variances are calculated for 50 consecutive program runs. It can be observed that the relative variance goes down if the workload goes up.

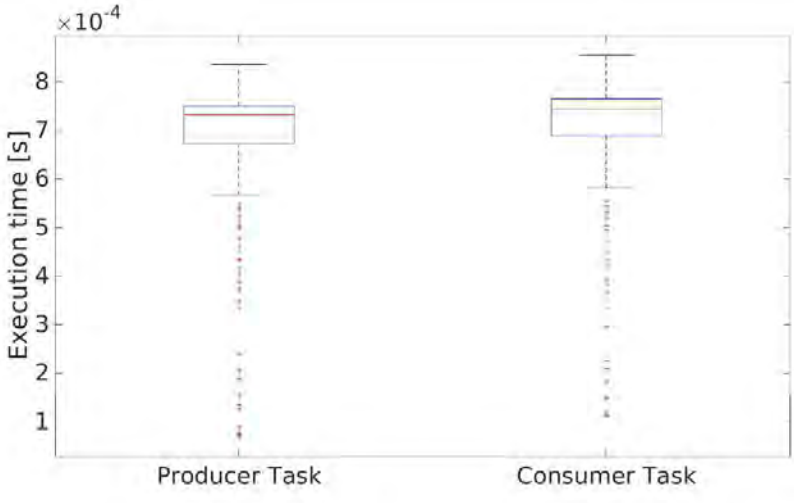


Figure 7-11: Boxplot for the two different task types

To investigate this point further, Figure 7-11 shows the makespan (time span between a task creation until the task has completely executed) of the producer and consumer task for a program run with $N_t = 1000$ and a workload of $D_t = 1$ byte. Since the workload is at a minimum level, this can also be seen as the minimum achievable execution time for one task. The mean value is found to be approximately 720 µs. Again it should be mentioned that this is the total makespan of one task. Due to the task pipeling, the pure execution time on a processing element is likley to be lower by several

decades. For the evaluation of efficiency, the utilisation of processing elements should be considered instead.

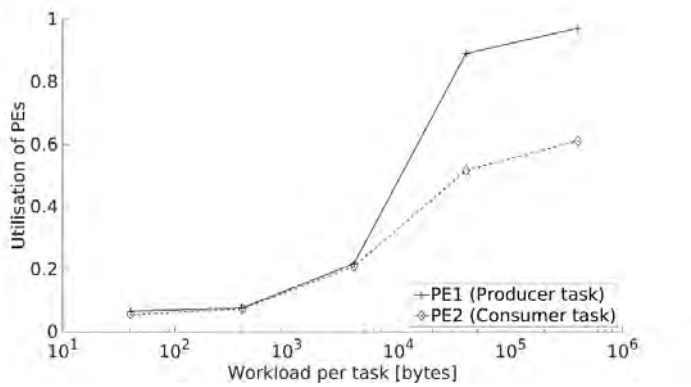
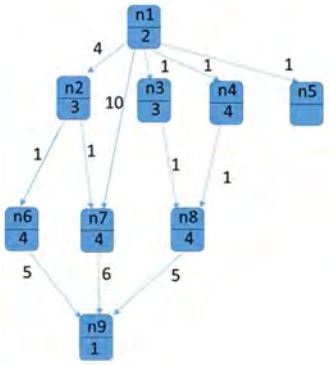


Figure 7-12: Utilisation of processing elements

The utilisation of the processing elements is depicted in Figure 7-12. For low workloads the utilisation is also low due to the mentioned overhead. However, for higher workloads an utilisation of over 90% can be achieved. The utilisation of PE2 stays lower due to the data dependencies. Since the producer task executes slower, it is the bottleneck within the test application and PE2 cannot achieve a high utilisation.

7.4 Task Scheduling

Efficient task scheduling is one of the important challenges in multi-processor system. Finding an optimal solution for a scheduling is NP-complete, thus it is necessary to identify suboptimal scheduling algorithm with reduced complexity. In general, list-scheduling (LS) is accepted as a good approach, since it combines advantage of low-complexity with nearly optimal results. Here the scheduling is done by taking into consideration the priority list. The priority can be calculated statically (using static application parameters) or dynamically (exploiting runtime task and engine status). The aim of this part of the work was to identify, implement and analyse relevant CoreManager scheduling algorithms. A short description of the selected scheduling algorithms follows. The description is based on the same task graph and an associated table with priority levels in Figure 7-13. Basically, a large set of known task scheduling algorithms rely on the exploitation of the t-level (top-level i.e. earliest task start time) and b-level (bottom-level i.e. latest task start time) parameters. The b-level of a task is the length of the longest path from the task to an exit node.



node	static b-level	b-level	t-level
n1	11	23	0
n2	8	15	6
n3	8	14	3
n4	9	15	3
n5	5	5	3
n6	5	10	10
n7	5	11	12
n8	5	10	8
n9	1	1	22

Figure 7-13: Example dataflow task graph with task execution time and communication cost parameters.

Highest Level First with Estimated Time (HLFET) is a simple static scheduling algorithm. HLFET uses b-level as a priority for scheduling. Here the communication costs are used when transferring a task from one processing element to the other one. The time-complexity of the HLFET algorithm is O(v2) (v is the number of vertexes). Figure 7-14 presents the principle of scheduling and the final mapping after resource allocation, resulting in the scheduling length of 17.

node	static b-level	t-level
n1	11	0
n2	8	6
n3	8	3
n4	9	3
n5	5	3
n6	5	10
n7	5	12
n8	5	8
n9	1	22

Step	ni	PE0	PE1	PE2
1	n1	0	NC	NC
2	n4	2	3	NC
3	n2	6	6	NC
4	n3	9	3	NC
5	n5	9	6	3
6	n6	9	10	10
7	n7	13	12	12
8	n8	13	16	8
9	n9	16	22	22

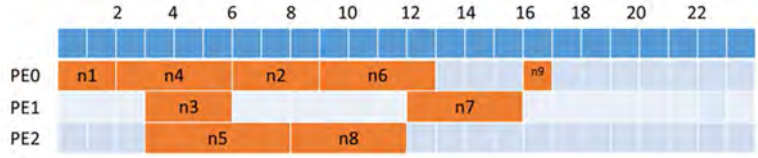


Figure 7-14: Principle of HLFET Algorithm

Critical path (CP) algorithm is a static scheduling algorithm that exploits CP as a priority level for scheduling. The CP is the length of the longest path from the task to an exit node without taking into consideration the communication costs. Here the communication costs are not assumed when transferring a task from one processing element to the other one. The time-complexity of the CP algorithm is O(v). The Scheduling principle and the final mapping are illustrated in Figure 7-15, resulting in the scheduling length of 14.

A modification of CP (MCP) uses ALAP as a priority level for scheduling. The ALAP start-time of a task is a measure of how far the node's start-time can be delayed without increasing the scheduling length. Here the communication costs are used when transferring a task from one processing

element to the other one. The time-complexity of the MCP algorithm is O(v2logv). The resulting scheduling length is 17.

node	CP
n1	11
n2	8
n3	8
n4	9
n5	5
n6	5
n7	5
n8	5
n9	1

Step	ni	PE0	PE1	PE2
1	n1	0	NC	NC
2	n4	2	2	NC
3	n2	6	2	2
4	n3	6	5	2
5	n5	6	5	5
6	n6	6	10	5
7	n7	6	10	9
8	n8	10	10	9
9	n9	13	13	13

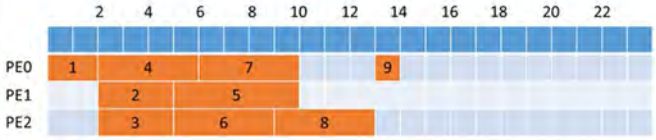


Figure 7-15: Principle of CP Algorithm.

Earliest Time First (ETF) algorithm is a dynamic scheduling algorithm. The ETF uses t-level as a priority for scheduling and chooses the task with the smallest t-level as a first executable task. The t-level is the length of the longest path from an entry task to the selected task with taking into consideration the communication costs. Here the communication costs are used when transferring a task from one processing element to the other one. The time-complexity of the ETF algorithm is O(pv2) (p is number of processors and v number of vertexes). Scheduling principle and final mapping are illustrated in Figure 7-16 resulting in the scheduling length of 17.

node	static b-level	t-level
n1	11	0
n2	8	6
n3	8	3
n4	9	3
n5	5	3
n6	5	10
n7	5	12
n8	5	8
n9	1	22

Step	node	PE0	PE1	PE2
1	n1	0	NC	NC
2	n4	2	3	NC
3	n3	6	3	NC
4	n5	6	6	3
5	n2	6	6	8
6	n8	9	8	8
7	n6	9	11	10
8	n7	13	12	12
9	n9	16	22	22

Figure 7-16: Principle of ETF Algorithm

We implemented above mentioned scheduling algorithms into dataflow engine. Our pipelined DFE architecture allows to combine the scheduling algorithms. In our approach, we employ CP and MPC static algorithms in the dataflow graph composition phase at slice processor. This allows us very early to define static priorities of tasks and to simplify the computation of dynamic schedule in runtime. In the runtime, the CoreManager implement ETF algorithm.

For proper functionality of scheduling algorithms and dataflow engine, the task and communication cost parameters of dataflow graph have to be correctly estimated. Currently, we use ad hoc method based on sampling during simulation for the parameter estimation that not perfectly match the real parameter values. For that reason, the scheduling results are not satisfactory at the moment. In order to cope with this problem, we attempt to extend our tool flow in order to automatically estimate the task parameters according to specific platform characteristics.

The Figure 7-17 shows an example with runtime traces acquired by dataflow engine in real-time. The traces represent baseband signal processing of 1 subframe (1ms). The benchmark implements LTE UL receiver at 5MHz channel bandwidths, 4x4 MIMO spatial multiplexing, 64QAM, and eight users with nearly equally distributed radio resources are processed. This results in total to 832 tasks processed in real-time by dataflow engine. The system comprise four PEs (cores) represented by traces TaskA-TaskB1/2/3/4. In addition, the traces of queue reading and task completion events are included. It is obvious that the total processing time is less then 2.5ms that meets the LTE requirements.

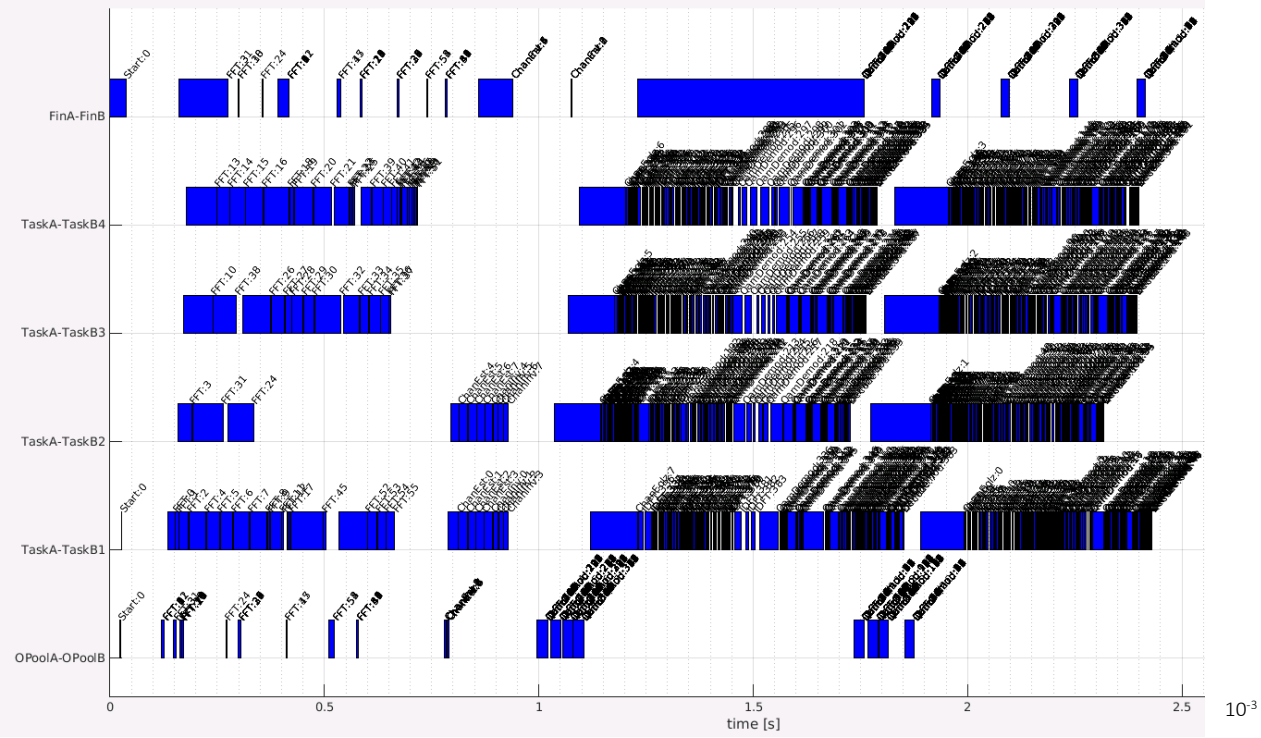


Figure 7-17: Runtime execution traces of dataflow engine comprising four PEs (TaskA-TaskB1/2/3/4). For illustration are displayed also queue reading (OPoola-OpoolB) and task completion events (FinA-FinB). Baseband benchmark is the LTE UL receiver with 5MHz channel bandwidth, 4x4MIMO, 64QAM and four users.

7.5 Conclusions

In this section we addressed achievements associated with the design and implementation of dataflow engine dedicated to C-RAN baseband signal processing. The dataflow framework include dataflow application abstraction and associated runtime system enabling to handle a plurality of parallel baseband signal processing slices. More particularly, it comprises i) flexible dataflow baseband applications ii) dataflow API and iii) dataflow engine. This deliverable primarily focus on the implementation of the methods for efficient mapping of dataflow applications to computation resources associated with T5.1 and T5.2. We presented briefly the application abstraction model and

the process of dataflow graph composition and transformation. Further we discussed SliceManager and CoreManager, the two key components of dataflow engine. We explained the CoreManager processing flow adn scheduling algorithms. Performance analysis of CoreManager subsystem is provided. Moreover, the application execution traces are presented that clearly indicated the full functionality and capability of dataflow framework.

7.6 References for section 7

[1] G. Fettweis, “A 5G Wirelesss Communication Vision,” Microwave Journal, Vol.55, No.12, pp 24-36, December 2012.

[2] G. Fettweis, “The Tactile Internet: Applications and Challenges,” IEEE Veh. Technology Magazine, vol.9, no.1, pp. 64-70, 2014.

[3] 3GPP, “TS 23.501: System Architecture for the 5G System,” TS 23.501, V1.5.0, Release-15, November 2017.

[4] P. Rost et al, “Network Slicing to Enable Scalability and Flexibility in 5G Mobile Networks,” IEEE Communications magazine, 2017.

[5] 3GPP, “TR38.802: Study on new radio access technology Physical layer aspects,” TR38.802 V14.2.0, Release-14, September 2017, http://www.3gpp.org/DynaReport/38-series.htm.

[6] O. Teyeb, G. Wikström, M. Stattin, T. Cheng, S. Faxér and H. Do, “Evolving LTE to fit the 5G future,” Ericsson technology review, Januar, 2017T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In ANCS 2015.

[7] Superfluidity deliverable D4.3: Innovative radio and network processing functions. December 2017.

[8] S. Haas et all, “A Heterogeneous SDR MPSoC in 28 nm CMOS for Low-Latency Wireless Applications,” in Proceedings of the Design Automation Conference (DAC'17), Austin, Texas, 2017.

8 Open vSwitch (OVS) firewall

8.1 Introduction

Just like any virtualization platform, the Superfluid platform relies on a back-end software switch to efficiently and intelligently distribute packets between the platform's network cards and the virtual ports to which the virtualized instances are attached. In this section we describe a set of improvements and optimizations to the platform's back-end switch.

8.2 Open vSwitch Firewall Driver

Open vSwitch (OVS) is the most popular network back-end for OpenStack deployments and it is widely accepted as the de facto standard OpenFlow implementation. Its main goal is to be a programmable switch, implementing both traditional switching functionality as well as programmability through OpenFlow. It is commonly used in network virtualization and most of its functionality is implemented/evaluated in user-space and a set of flows are programmed into the kernel with matches and actions.

OVS is good for stateless, flow-based networking. For instance to compose a network, including switching, routing and building network processing pipelines. However, it leaves out services that might be inserted into that network, such as firewalls. However, OpenStack security groups (SG) give you a way to define packets filtering policy that is implemented by the cloud infrastructure. Unfortunately, OVS could not interact directly with iptables to implement security groups. To overcome this problem, OVS integration into OpenStack (ML2 + OVS) make use of iptables to implement security groups, however a linux bridge between each instance (VM) and the OVS integration bridge is required, i.e., the VM needs to be connected to a tap device that is put on a linux bridge, and then connect that linux bridge to the OVS bridge using a veth pair (see Figure 8-1). Then, the linux bridge contains the iptables rules pertaining to the instance.

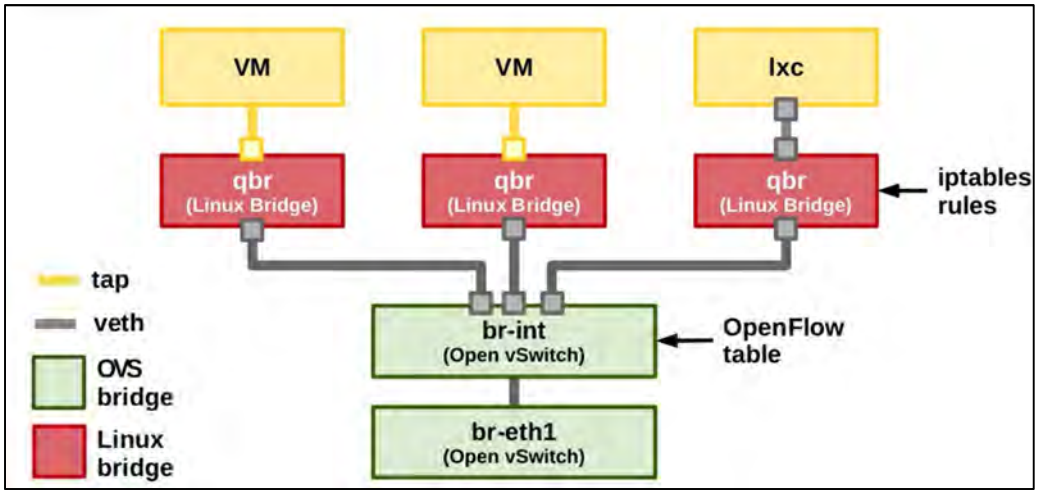


Figure 8-1: Open vSwitch integration with Linux Network stack

Even though it is great that this works, the extra layers are not ideal and create both scalability and performance problems that could be of great impact in the 5G/Edge cloud scenario, therefore being a barrier for OpenStack adoption for NFV purposes. In order to get rid of all of the extra layers between VMs (and/or containers) and OVS, and yet being able to apply security groups, there is a need of building stateful firewall services in OVS directly. Consequently, the solution was to modify the OVS agent to include an optional firewall driver that natively implements security groups as flows in OVS rather than linux bridge and iptables. There were two ways of implementing a firewall in OVS, but none ideal:

- Match on TCP flags by enforce policy on SYN, allowing ACK|RST. Although this solution is fast, it allows non-established flows through with ACK or RST set (only TCP).
- Use “learn” action to setup new flow in reverse direction. This solution seems more correct than the previous one, but it forces every new flow to OVS user-space, reducing flow setup by order of magnitude.

OVS development initially had a narrow focus on supporting novel features necessary for advanced applications such as network virtualization. However, due to its wider use it became clear it was not enough with OVS being highly programmable and general, but it also needed to be blazingly fast. Consequently none of the previous solutions was good enough and a new one was needed.

The solution proposed is the integration of connection tracking (conntrack module) from linux kernel to enable stateful tracking of flows, i.e., the OVS can call into the kernel connection tracker. To understand how this is achieved, it is needed to understand the OVS architecture – see Figure 8-2. The forwarding plane consist of two parts: a slow-path user-space daemon called ovs-vswitchd and a

fast-path kernel module. Most of the complexity such as forwarding decisions and networking protocol processing are handled at user-space; while the kernel module is in charge of tunnel termination and caching traffic handling.

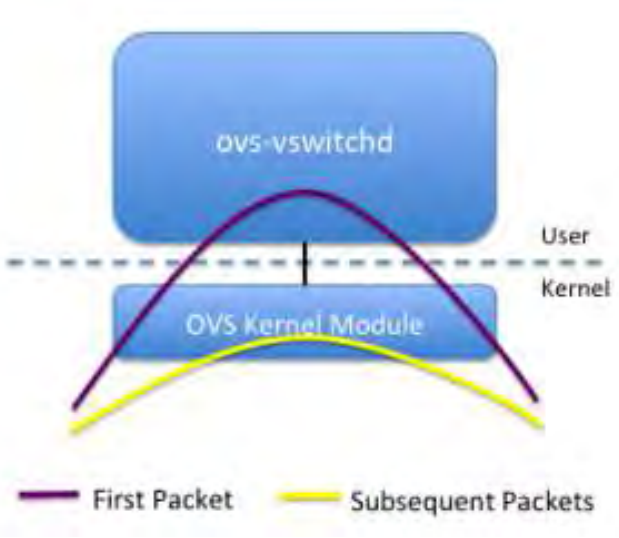


Figure 8-2: OVS architecture

The conntrack functionality can be used to build stateful services in OVS as a stateless flow can select a class of traffic that needs to be tracked, passing matching packets off to the conntracker for further evaluation. Thanks to its use, when a packet is received by the kernel module, its cache of flows is consulted. If a relevant entry is found, then the associated actions (e.g., modify headers or forward the packet) are executed on the packet. By contrast, if there is no entry, the packet is passed to ovs-vswitchd (i.e., user-space) to decide the packet's fate. Ovs-vswitchd then executes the OpenFlow pipeline on the packet to compute actions, passes it back to the fast-path for forwarding, and install a flow cache entry so similar packets will not need to take these expensive steps. Therefore, subsequent packets are processed entirely in the kernel, with the consequent processing speed up.

Without going into tedious details about the flows, here is an idea of what it lets you do in your packet processing pipeline:

1. In one stage, you can match all IP traffic and send it through the connection tracker.
2. In the next stage you now have the connection tracker's state associated with this packet, so:
    a. For packets representing a new connection, a custom policy can be used to decide if connection should be accepted or not. If it is accepted, you can tell the connection tracker to remember this connection
    b. When packets are associated with existing connections they can be allowed through, and the opposite for return traffic.


c. It is known if a packet is invalid because it is not the right type of packet for a new connection and does not match may existing known connection.

8.2.1 OVS Features

The OVS firewall driver has the same API as the current iptables firewall driver, keeping the state of security group and ports inside of the firewall. Class SGPortMap was created to keep state consistent and maps from ports to security groups and vice-versa. Every port and security group is represented by its own object encapsulating the necessary information. Note: Open vSwitch firewall driver uses register 5 for marking flow related to port and register 6 which defines network and is used for conntrack zones, i.e., to support overlapping CIDR when belonging to different tenants.

8.2.1.1 Firewall API calls

There are two main calls performed by the firewall driver in order to either create or update a port with security groups - prepare_port_filter and update_port_filter. Both methods rely on the security group objects that are already defined in the driver and work similarly to their iptables counterparts:

- prepare_port_filter must be called only once during port creation, and it defines the initial rules for the port.
- When the port is updated, all filtering rules are removed, and new rules are generated based on the available information about security groups in the driver.

Security group rules can be defined in the firewall driver by calling update_security_group_rules, which rewrites all the rules for a given security group. If a remote security group is changed, then update_security_group_members is called to determine the set of IP addresses that should be allowed for this remote security group. Calling this method will not have any effect on existing instance ports. In other words, if the port is using security groups and its rules are changed by calling one of the above methods, then no new rules are generated for this port. update_port_filter must be called for the changes to take effect.

All the machinery above is controlled by security group RPC methods, meaning that the firewall driver does not have any logic of which port should be updated based on the provided changes, it only accomplishes actions when called from the controller.

8.2.1.2 OpenFlow rules

At first, every connection is split into ingress and egress processes based on the input or output port respectively. Each port contains the initial hardcoded flows for ARP, DHCP and established


connections, which are accepted by default. To detect established connections, a flow must be marked by conntrack first with an action=ct() rule. An accepted flow means that ingress packets for the connection are directly sent to the port, and egress packets are left to be normally switched by the integration bridge.

Connections that are not matched by the above rules are sent to either the ingress or egress filtering table, depending on its direction. The reason that forces rules based on security group rules to reside in separate tables, is to allow their easy detection during removal. For more information, please visit: http://docs.openstack.org/developer/neutron/devref/openvswitch_firewall.html

8.2.2 OVS Usage

The native OVS firewall implementation requires the kernel and user space support for conntrack and consequently a minimum versions of the Linux kernel and Open vSwitch:

- Kernel version 4.3 or newer includes conntrack support.
- Kernel version 3.3, but less than 4.3, does not include conntrack support but OVS modules can be built
- Open vSwitch version 2.5 or newer

On the other hand, to enable the OVS firewall driver on the nodes running the OVS agent, it is needed to edit the openvswitch_agent.ini file and enable the firewall driver like this:

```
[securitygroup]
firewall_driver = openvswitch
```

9 Cache Allocation Technology and Noisy Neighbour effects

A major challenge with Network Functions Virtualisation (NFV) is ensuring protection of Virtual Network Function (VNF) resources against "Noisy Neighbour" effects. This is essentially the result of shared resources being consumed *in extremis* within a multi-tenant setup, meaning one VNF's resources are restricted by that of another VNF. One of the major shared resource bottlenecks is the central processor's Last Level Cache (LLC). In this section, we discuss a solution implemented in a testbed which enables "Cache Allocation Technology" (CAT), so as to deterministically prioritize LLC resources between competing workloads. A number of CAT "Class of Service" (CoS) paradigms are explained for a range of service chain scenarios, involving virtual Firewall and virtual Router VNFs alongside a Noisy Neighbour VNF. Significant performance benefits are confirmed, bringing the performance of target VNFs in the presence of a LLC-hungry Noisy Neighbour, into alignment with the baseline scenario of a "Noise-Free" neighbouring VNF.

9.1 Summary of results

Cache Allocation Technology (CAT) can be used to make NFV performance more *predictable & deterministic*, by mitigating Noisy Neighbour effects on shared processor resources such as the Last level Cache (LLC). For specific tests undertaken, the most notable impact on performance was observed when target VNFs were deliberately starved of LLC resources by allocating minimal LLC bandwidth (i.e. a single CAT CoS way). This was referred to as the "Uneven", or "11-1" model, whereby 11 of the cache's 12 available CoS ways were assigned to the Noisy Neighbour VNF, while the remaining single CoS way was used by all other workloads. The Even CAT CoS settings mitigate against performance impacts, and bring the results into close approximation to the "Stress-Free" Neighbour scenario where the LLC resources are not being aggressively hogged. As an example, the "Even" CAT CoS model, reduced average latency between 47% & 92% compared to Uneven CoS, while the maximum latency was reduced between 49% & 98%.

9.2 CAT Background

9.2.1 Noisy Neighbours and Last Level Cache (LLC)

A crucial aspect of NFV performance management is ensuring protection of VNF resources against "Noisy Neighbour" effects. This is the result of shared resources being consumed *in extremis* within a multi-tenant setup, meaning one VNF's resources are restricted by that of another VNF, in such a way as to negatively impact performance. Within a physical Central Processing Unit (CPU) socket

architecture (Figure 9-1), a number of shared resource pools exist including interconnect/IO, Last level Cache (LLC) and Shared Memory Bandwidth (SMB). So although a VNF can have CPUs “pinned” for performance assurance, they will still be subject to resource-sharing within the LLC, and if a memory-hungry neighbouring VNF is instantiated on the same host OS, this constitutes a “Noisy Neighbour” problem.

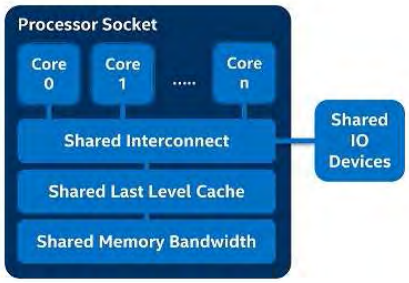


Figure 9-1: Shared Processor Resources

Since 2015, Intel launched a range of core processors enabled to support features such as Cache Monitoring Technology (CMT), Cache Allocation Technology (CAT) and Memory Bandwidth Monitoring (MBM). The technologies provide pro-active monitoring of shared resource consumption, and active management of the available resources in order to achieve platform-level Quality-of-Service for different VNFs in a multi-tenant environment. This paper provides a detailed practical analysis of the impact of CAT on a realistic multi-tenant NFV scenario, using commercially-available virtual Router and virtual Firewall VNFs as the “systems under test”. This builds and expands on previous studies such as [1], which although able to demonstrate the performance benefits of CAT, were based on completely synthetic workloads for both the “Target” and “Noisy Neighbour” VNFs. Section II provides an overview of the testbed, including the methodology for implementing active CAT. Section III presents the detailed test results for a range of scenarios, while Section IV presents the conclusions and key directions for further work.

9.3 Testbed Overview

The testbed is based on Linux KVM (Kernel-based Virtual Machine) using Open vSwitch plus DPDK, and the x86 server uses Intel “Xeon D” processors, enabled to support CMT and CAT software. Table 1 lists key hardware/software elements.

Table 1: Testbed Hardware and Software Components

Testbed Component	Version/Description
X86 Server Hardware	Intel Xeon D-1537 (16 * logical vCPU processors, 16G RAM, 256GB SSD Storage)
Hypervisor Base OS & Kernel	Centos 7 3.10.0-327.22.2.e17.x86_64
Hypervisor Open vSwitch (OVS)	2.6.0
DPDK (Data Plane Development Kit)	16.07
QEMU	2.5.1.1
VNF 1- vRouter	Brocade 5600 virtual router: 4 vCPU, 4G RAM
VNF2- vFirewall	Fortinet Fortigate VM64-KVM: 1 vCPU, 1G RAM
VNF3- Noisy Neighbour VM	Fedora 22: 2 vCPU, 2G RAM Stress Processes: stress-ng-0.03.20, memtester version 4.3.0
Test Equipment	Spirent Avalanche 4.40, running "RFC2544" throughput test wizard, fixed +"iMIX" profiles

The testing involved two distinct service chain scenarios as shown in Figure 9-2. The first comprised of the so-called "Target" VNF, which could either be the virtual Firewall or virtual Router, alongside a "Noisy Neighbour" VNF in a multi-tenant, mixed VNF scenario. The target VNFs are in the data path and considered revenue-generating and high value. The second scenario involved two serially-connected virtual Routers alongside the Noisy Neighbour VNF. In both scenarios, the Noisy Neighbour is not in the data path, but has access to the Last Level Cache (LLC) and other processor resources shared by all VNFs. This is an ideal test case, as the impacts of Noisy Neighbour on the "Target" VNFs can be observed, while maintaining a distinct separation of the data path used by the traffic generator. The Noisy Neighbour makes use of multiple synthetic resource-hogging stress processes installed and run from the Linux OS, and as shown in Table 1, include "*stress-ng*" and "*memtester*" applications.

The target VNFs use "vhost" drivers connecting to the virtual switches (labelled as Lan-Bridge and Wan-bridge in the diagram), which themselves are connected to physical Gigabit Ethernet interfaces set-up using DPDK "Poll-Mode Drivers" (PMDs). As already stated, the Noisy Neighbour VNF is not in the data path and thus uses standard "virtio" drivers connecting to the virtual switch labelled "vm-bridge".

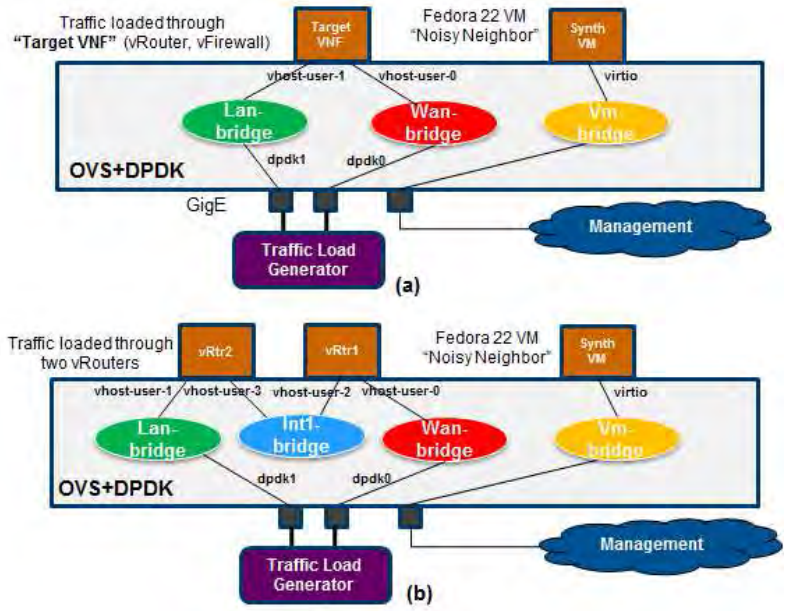


Figure 9-2: (a) Target VNF + Noisy Neighbour (b) Dual VNFs + Noisy Neighbour

9.4 Methodology

Cache Allocation Technology is run in the host hypervisor, and involves the use of a "Class of Service" (CoS) construct acting as a resource control tag into which a CPU thread can be grouped. Other, more granular mappings at application/VM/container level are also possible, but this paper covers the CPU thread case. Software-programmable control is provided over the amount of Last-Level Cache (LLC) space that can be consumed by a given CPU core thread, hence prioritization can be applied for Virtual Network Functions (VNFs) running on the host hypervisor.

Definition of capacity bitmasks determines how much of the LLC space is available, and the degree of overlap and isolation. Typically, a capacity bit to cache way mapping is 1:1 but one-to-many options are also possible. For the Intel® Xeon® D-1537 processor, the LLC has 12 ways, and the CoS capacity bitmask has 12 bits, each comprising 1MB of cache storage. The term "CoS ways" is also used in this paper to describe the capacity bitmask. Table 2shows which CPUs are pinned for specific VNFs and processes; it is worth noting that in the Linux system setup, isolation of CPU cores 1-15 ensures core 0 is reserved for Linux processes.

Table 2: VNF and System Process CPU Pinning

Function/Process	Pinned CPUID(s)- 16 vCPU processor
Noisy Neighbour VNF	6,7
Target VNF- Single Firewall	5
Target VNF- Single Router	1,2,3,5
Target VNF- Second Router	9,10,11,13
OVS-PMD- (Open vSwitch Poll Mode Driver)	4, 12
OVS-db (Open vSwitch Database)	1

Table 3 shows the range of CoS definitions and corresponding CPUID assignments used for the performance evaluation described in detail in the following section. In the Uneven CoS model, 11 of the 12 CoS Ways are assigned to the Noisy Neighbour, while only a single CoS way is available for the remaining CPUIDs, which includes that of the target VNF (e.g. vRouter or vFirewall). This model encourages an extremely aggressive "hogging" of LLC resources by the Noisy Neighbour to demonstrate what impact on performance the cache starving can have on the "Target VNFs"; this scenario could indeed be representative of a worst-case scenario whereby a *malicious* Noisy Neighbour generates intentional Denial-of-Service behaviour to shared cache resources. The Even CoS models allow a fairer distribution of LLC bandwidth (i.e. an equal split of CoS ways) between competing Guest VNFs on the host platform, as well as other key system processes such as OVS-PMD and OVS-db.

Table 3: Initial Class of Service Definitions Using Capacity Bitmasks

CAT CoS Model	Binary Bitmasks	Hex	Cache capacity	CPU Assignments
Uneven ("11-1")	**1111 1111 1110** 0000 0000 000**1**	0xffe 0x1	11 MB 1MB	6,7 (Noisy N'bor) 0-5,8-15 (Everything Else)
Even ("4-4-4", Single virtual Firewall)	**1111** 0000 0000 0000 **1111** 0000 0000 0000 **1111**	0xf00 0xf0 0xf	4 MB 4 MB 4 MB	6,7 (Noisy N'bor) 5 (virtual Firewall) 0-4,8-15 (Everything Else)
Even ("4-4-4", Single virtual Router)	**1111** 0000 0000 0000 **1111** 0000 0000 0000 **1111**	0xf00 0xf0 0xf	4 MB 4 MB 4 MB	6,7 (Noisy N'bor) 1,2,3,5 (virtual Router) 0,4,8-15 (Everything Else)
Even ("3-3-3-3", Dual virtual Routers)	**111**0 0000 0000 000**1 11**00 0000 0000 00**11 1**000 0000 0000 0**111**	0xe00 0x1c0 0x38 0x7	3 MB 3 MB 3 MB 3 MB	6,7 (Noisy N'bor) 1,2,3,5 (1st virtual Router) 9,10,11,13 (2nd virtual Router) 0,4,8,12,14,15 (Everything Else)

9.5 Results

9.5.1 Virtual Firewall Tests

Figure 9-3 shows the maximum throughput (Mbit/s) for the virtual Firewall alongside the Noisy Neighbour VNF for a selected range of fixed frame sizes, and three setup variations:

- No stress processes running in Neighbouring VNF.
- Stress processes running in Neighbouring VNF and Uneven "11-1" CoS model of Table 3.
- Stress processes running in the Neighbouring VNF and Even "4-4-4" CoS model of Table 3.

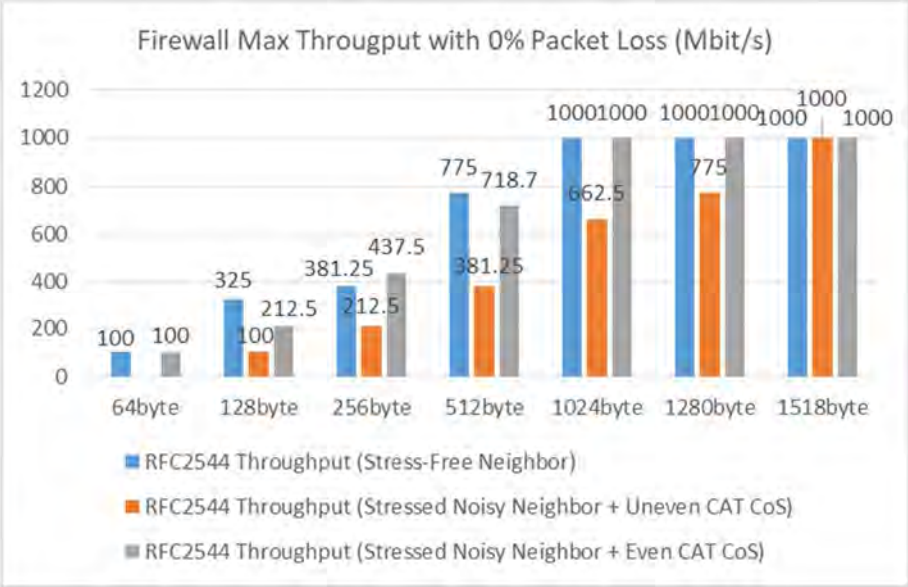


Figure 9-3: vFirewall Maximum "RFC2544" Throughput

Figure 9-4 and Figure 9-5 show the corresponding average and maximum latency for a subset of fixed frame sizes, with measurements taken after 60 seconds, and the traffic load used for each case was the corresponding baseline throughput realized with a *stress-free* neighbouring VNF: 64byte @ 100Mbits, 256byte @ 381Mbit/s, 1024byte & 1518byte @ 1000Mbit/s.

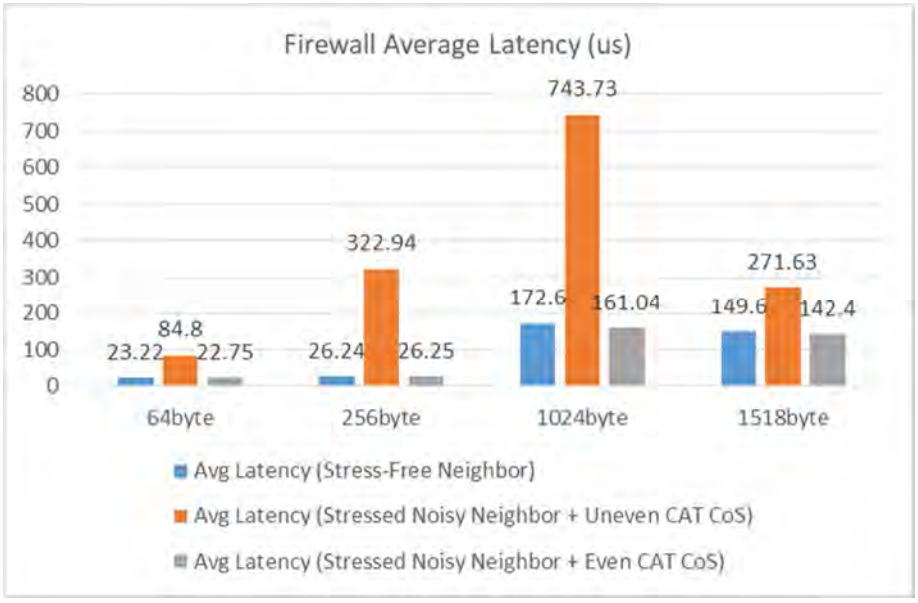


Figure 9-4: vFirewall Average Latency

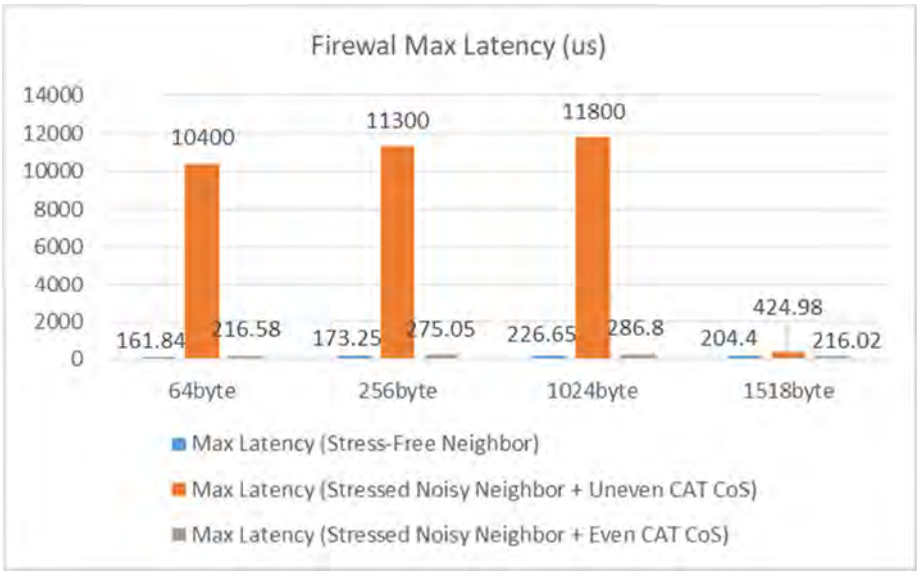


Figure 9-5: vFirewall Maximum Latency

The graphs show that Firewall throughput & latency are significantly impacted by the presence of a Noisy Neighbour + Uneven CAT CoS settings. The Even CAT CoS settings mitigate against performance impacts, and bring the results into close approximation to the "Stress-Free" Neighbour scenario where the LLC resources are not being aggressively hogged. The average latency is below 170us, while the maximum is below 300us achieving more deterministic performance. In percentage terms, the "Even" CAT CoS model, reduces average latency between 47% & 92% compared to Uneven CoS, while maximum latency is reduced between 49% & 98%. It is also insightful to observe the LLC occupancy of the Noisy Neighbour and virtual Firewall VNFs during the load tests. Figure 9-6 shows

the measured LLC in KB using the embedded Cache Monitoring Technology (CMT)- for the Uneven and Even CoS cases, respectively.

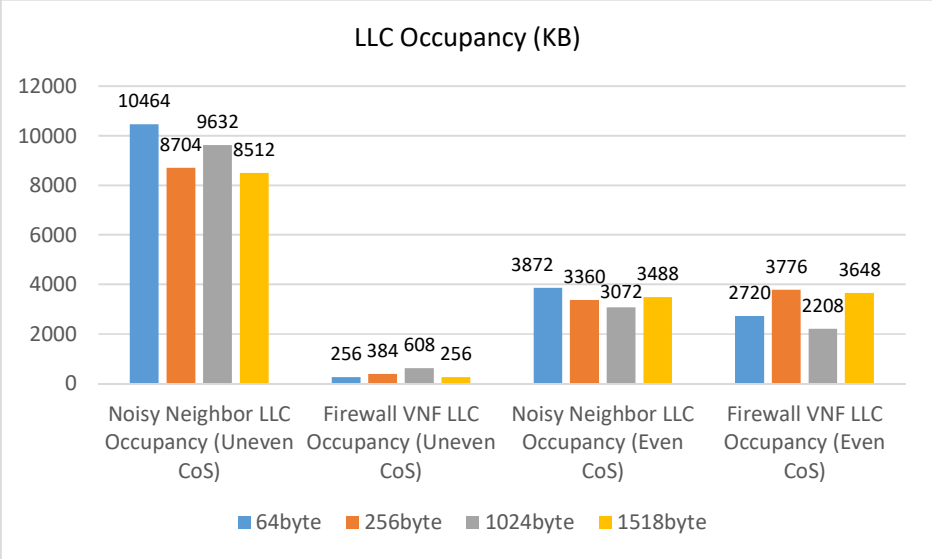


Figure 9-6: Comparative LLC Occupancy for Uneven and Even CoS Paradigms

These experiments confirm the occupancy levels of the LLC is heavily dominated by the Noisy Neighbour in the Uneven "11-1" case, (e.g. ~10.5MB utilisation for 64byte packets), but is much more balanced between the Noisy Neighbour and Target VNF for the Even CoS "4-4-4" case.

9.5.2 Virtual Router Tests

A similar set of tests were executed with the virtual Router substituting the virtual Firewall with the same three setup variations as described earlier. The tests achieved broadly comparable results with those for the virtual Firewall, hence for brevity are summarized in Table 4.

Table 4: Average and Maximum Latency Single Virtual Router

Fixed Frame Size	Average Latency us (Noisy Neighbour, Uneven CoS)	Average Latency us (Noisy Neighbour, Even CoS)	Max Latency us (Noisy Neighbour, Uneven CoS)	Max Latency us (Noisy Neighbour, Even CoS)
64byte	7096	29.59 (-**99%**)	8542	115.08 (-**98%**)
256byte	119.14	38.83 (-**67%**)	207.77	88.84 (-**57%**)
1024byte	140.11	86.59 (-**38%**)	206.22	102.34 (-**50%**)
1518byte	143.08	91.62 (-**36%**)	164.71	109.11 (-**33%**)

With Even CoS, the average latency is below 100us, while the maximum is below 120us achieving more deterministic performance. In percentage terms, the "Even" CAT CoS model, reduces

average latency between 36% & 99% compared to Uneven CoS, while the max latency is reduced between 33% & 98%. The same virtual Router was also used in a "Three VNF" set-up whereby two virtual Routers were service chained together, plus a Noisy Neighbour (i.e. the testbed setup of Figure 9-2(b)) the Even CoS model in this case is the "3-3-3-3" model of Table 3. Table 5 presents both the Average and Maximum Latency Figures.

Table 5: Average and Maximum Latency Single Virtual Router

Fixed Frame Size	Average Latency us (Noisy Neighbour, Uneven CoS)	Average Latency us (Noisy Neighbour, Even CoS)	Max Latency us (Noisy Neighbour, Uneven CoS)	Max Latency us (Noisy Neighbour, Even CoS)
64byte	13438.95	41.64 (-**99%**)	16335.98	163.77 (-**99%**)
256byte	14948.75	76.75 (-**99%**)	17506.31	171.47 (-**99%**)
1024byte	186.53	134.1 (-**28%**)	261.24	160.79 (-**38%**)
1518byte	328.81	138.8 (-**57%**)	1321.77	163.59 (-**87%**)

For the dual virtual Routers with Even CoS, average latency is sub 140 us, while the maximum is sub 180 us. In percentage terms, the "Even" CAT CoS model, reduces average latency between 28% & 99% versus Uneven CoS, while maximum latency is reduced between 38% & 99%.

9.5.3 Variable CoS Model Results

The availability of multiple "CoS ways" to manage LLC resources opens up the possibility of a large and flexible number of set-up permutations in terms of CPU thread allocations, whether isolated or overlapping assignments are made, and so on. The earlier results were based on distinct paradigms to show the performance impact of a Noisy Neighbour on target VNFs while using distinctly "Uneven" and "Even" CAT CoS models. It is insightful to run additional tests using *intermediate* CAT CoS settings as shown in Table 6. Compared to the "11-1" model, the "7-4-1" model reduces the LLC bandwidth allocation for the Noisy Neighbour (from 11 to 7 CoS ways), whilst dedicating resources for key system processes, hence assigns 4 CoS ways for "Everything Else" including host OS, OVS-db, and OVS-PMD. Just a single CoS way is used for the target VNF, which, in these tests is the virtual Firewall. The "6-4-2" model meanwhile, doubles the LLC bandwidth for the target VNF (1 to 2 CoS ways), while reducing the Noisy Neighbour CoS ways from 7 to 6.

Table 6: Additional (intermediate) Class of Service Definitions

CAT CoS Model	Binary Bitmasks	Hex	Cache capacity	CPU Assignments
"7-4-1"	**1111 111**0 0000 0000 000**1 111**0 0000 0000 000**1**	0xfe0 0x1e 0x1	7 MB 4 MB 1 MB	6,7 (Noisy Neighbour) 0-4,8-15 (Everything Else) 5 (virtual Firewall)
"6-4-2"	**1111 11**00 0000 0000 00**11 11**00 0000 0000 00**11**	0xfc0 0x3c 0x3	6 MB 4 MB 2 MB	6,7 (Noisy Neighbour) 0-4,8-15 (Everything Else) 5 (virtual Firewall)

The average and maximum latency for both fixed 256 byte frames and "iMIX"- generating a *realistic* split of frame sizes - are shown in Figure 9-7 and Figure 9-8: the results are plotted for the various CAT CoS paradigms, and clearly illustrate how different permutations of CoS way allocations yield different outcomes, with some dependency on traffic profiles. The "7-4-1" profile reduces average latency compared to "11-1" (i.e. Uneven CoS), but is more notable for fixed 256byte frames, than with iMIX traffic. For maximum latency, the "7-4-1" profile has no notable impact compared to "11-1". The "6-4-2" profile visibly reduces both average and maximum latency compared to "11-1" (i.e. Uneven CoS) for both traffic cases and indeed has comparable values to the "4-4-4" paradigm, suggesting even a single extra CoS way can notably improve target VNF performance. A compelling observation therefore is confirmation of a *minimum* required number of CoS ways for the target VNF (i.e. two), to achieve comparable performance with the completely "Even CoS" model.

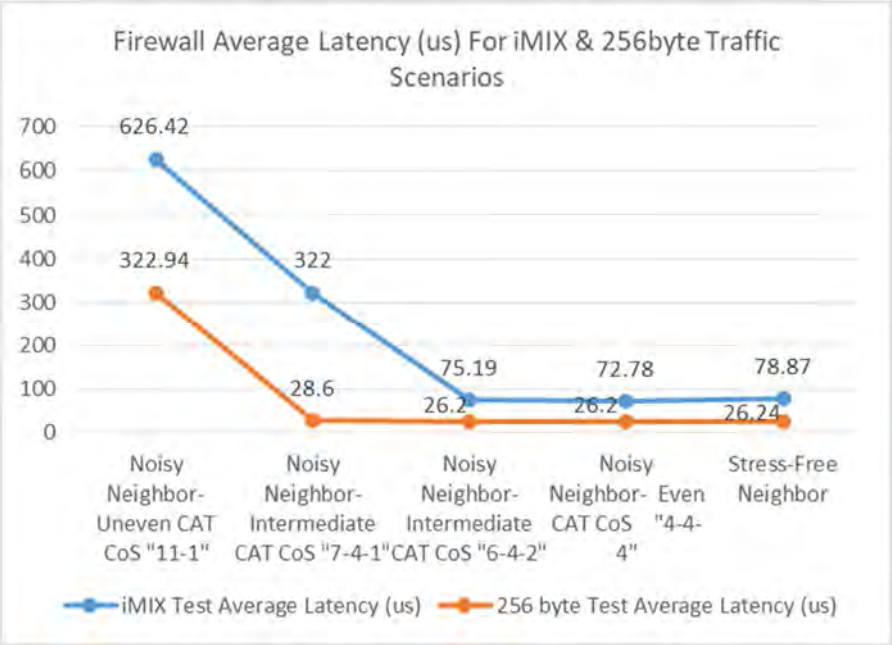


Figure 9-7: vFirewall Average Latency

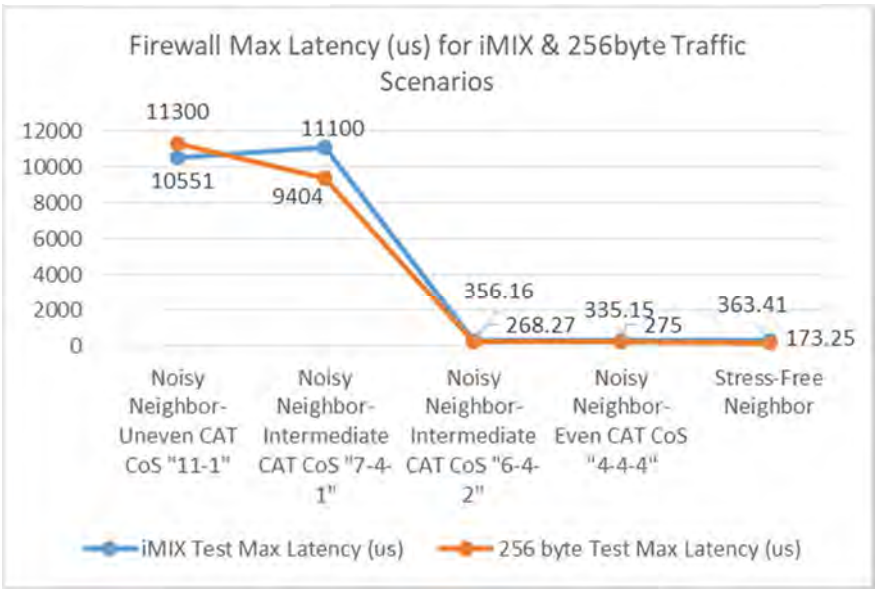


Figure 9-8: vFirewall Maximum Latency

10 MicroVisor: Hypervisor Level Improvements

The MicroVisor platform has been developed as part of the FP7-EUROSERVER project that is tasked with developing the server platform for next generation data centers that is power efficient. Within Superfluidity the effort has been on optimising the platform to support network functions and RFBs at the density and scale that is expected for 5G-PPP. In order to provision VMs and services within the order of 10s of ms it is essential to understand the full operation of the stack and perform the operations as efficiently as possible with minimal overhead. Although improvements can be made to VMs to make them smaller and more optimised, which is also described in this document, it is important to align the underlying hypervisor platform to be aligned with the hardware architecture to benefit from the maximum performance and minimal overhead.

The following section describes the performance analysis of ONAPP's MicroVisor platform for running NFV type operations. In most of the comparisons, we compare the performance of stock Linux, Xen, a Virtual Machine (VM) running on Xen and a VM running on the MicroVisor (MV).

10.1 Benchmark Setup

10.1.1 Benchmark Sets

a) Memcached (v1.4.36)

   Memaslap (libmemcached v1.0.18)

   Client: memaslap -s host:port -t 60s -T 1 -c 20 -X 64

   Server options: -u root -m 8192

b) Redis (v3.2.8)

   client: redis-benchmark -h host -p port -q -c 1

   Server options: maxmemory $[4096 * 1048576]

c) Netperf (2.7.0)

   netperf -t UDP_RR, TCP_RR, UDP_STREAM, TCP_STREAM

   Server options: default

10.1.2 Benchmark Environments

1) Baremetal

   In this test, we setup two nodes with baremetal linux (v4.11) and run all benchmarks from the above set between the two nodes and localhost

2) Docker

In this test, we setup one docker host with the respective containers (memcached, redis and netperf) and execute the tests between the baremetal node and the docker container. There are cases where for completeness we ran the tests from the dockerhost to the container.

The docker containers used are:

- paultiplady/netperf (netperf, iperf, NPtcp)
- Redis:3.0.7-alpine (redis)
- Memcached:1.4.35-alpine (memcached)

3) Stock Xen VM

   We setup a Xen host with default values and run the tests from/to the baremetal node.

4) Stock Xen unikernel

   Instead of using a fully blown VM, we use a unikernel built with the rumpkernel framework.

5) MV unikernel

   We setup one node as a MV and the other as the baremetal node.

6) MV integrated driver unikernel

   Same as Stock Xen unikernel.

10.2 Benchmark Results

10.2.1 Network Throughput and Latency

For many types of workloads on the Cloud, the main performance bottleneck is the network connectivity. Distributed platforms rely on networked resources and as such any improvements to the networking subsystem will also lead to benefits in different areas. Distributed storage platforms for instance rely on good mixed mode data and control transport. TCP tends to behave very poorly for small message sizes with throughput being severely limited. This has an impact on control messages and small, randomised data IO. Various techniques can be used to aggregate data before transmitting over the network (caching) to align the storage system with the optimal network flow. As such it is important to analyse the behaviour of TCP for the platform.

To analyse the behaviour we have captured the TCP throughput performance comparison between standard native Linux with bonding with MicroVisor in Figure 10-1. The more parallel streams that there are, the more Linux can take advantage of the dual 10Gb Ethernet links. The MicroVisor however takes advantage of all available links even if only one stream is available. The throughput performance of the MicroVisor is very close to the physical capacity of the aggregate links with 19.7Gb across 20Gb links. With 6 or more streams the Linux bonding performs close to the physical capacity with 19.7Gb throughput.

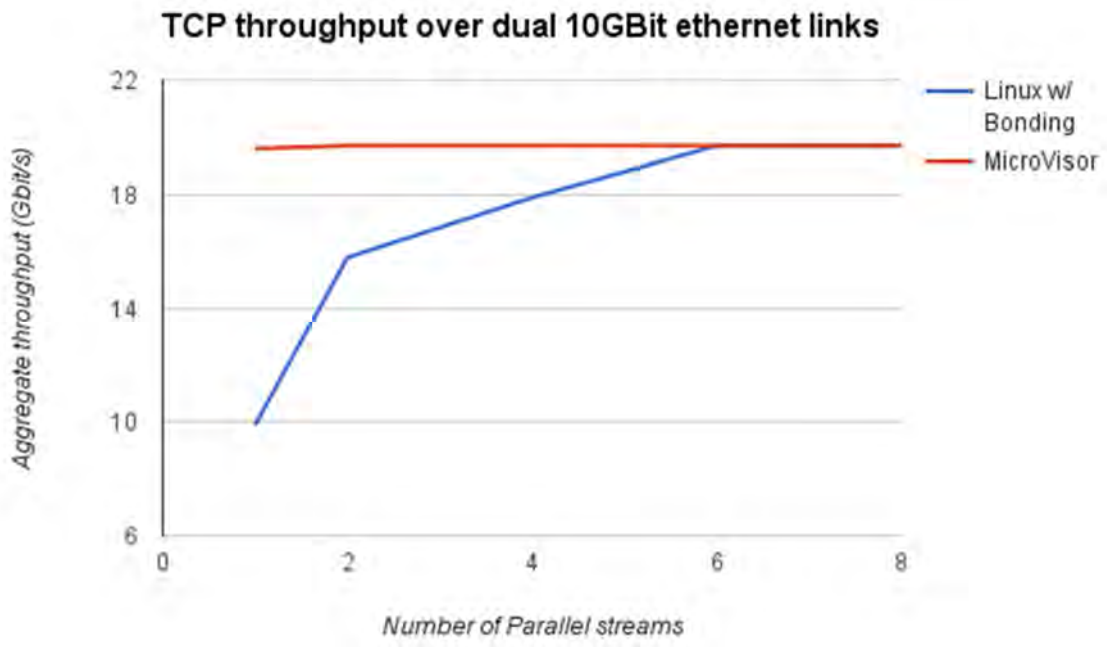


Figure 10-1: TCP throughput comparison of MicroVisor and Linux native with bonding across dual 10Gb Ethernet links

In Figure 10-2, we plot the UDP throughput measured with the netperf benchmark between a bare-metal node running vanilla linux v.4.11 and a node running:

a) Vanilla linux v4.11
b) Stock Xen 4.9
c) MV

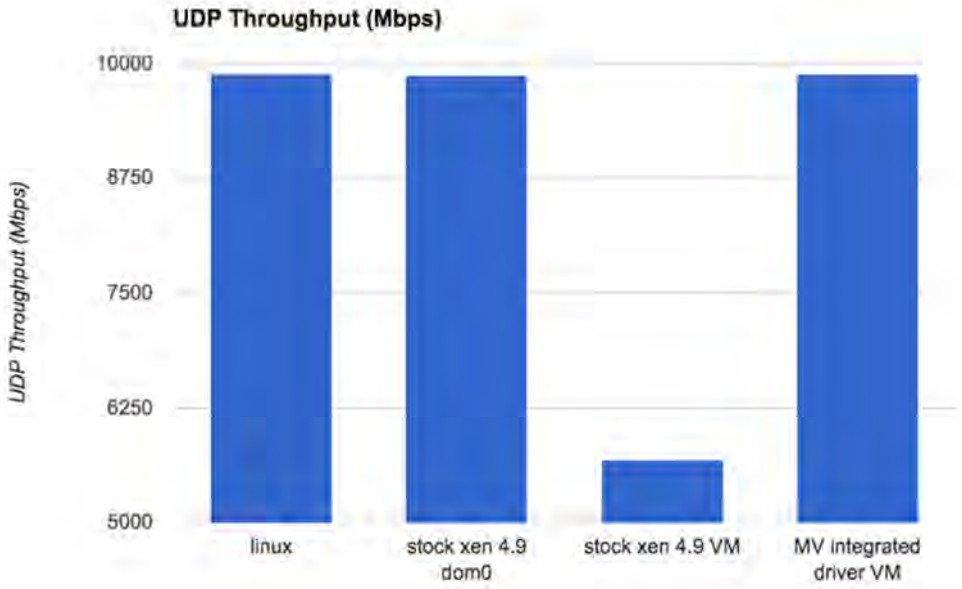


Figure 10-2: UDP throughput compared for stock Linux, Xen, Stock Xen VM and MV VM

The goal of this experiment is to identify the overhead related to virtualising network I/O paths. In the first bar we set our baseline: two linux hosts are able to transfer UDP data at a rate of 9.8Gbps, which is almost the available line rate. This small overhead is considered to be protocol related.

The third bar plots the throughput measurement between a vanilla linux host and a stock Xen guest VM (bridged setup). Clearly there is considerable overhead related to a number of factors:

a) Driver domain intervention. To account for the driver domain overheads, we analyse this in more detail. The second bar in this graph plots the driver domain UDP throughput. The virtualisation overhead involved in this case is PCI passthrough (interrupt mapping to event channel translation) as well as memory access overheads regarding Para-virtualized page table walking and updates.

b) Scheduling effects. The Xen scheduler plays a significant role in I/O throughput with regards to scheduling a specific guest to a physical core in time for it to complete I/O. This is not network specific. When a hardware interrupt reaches the hypervisor, the driver domain is notified using the event channel mechanism and the incoming packet walks up the network stack of dom0's linux kernel. It crosses the network bridge and reaches the netback driver which ends up issuing a hypercall to notify the guest of an incoming frame. The guest is notified and it wakes up on a different core, completing the network transaction. So there needs to be really good collaboration of all the intermediate components (scheduling driver

domain's vCPUs and guest's vCPUs to specific cores) in order to achieve maximum throughput. This is clearly not the case in the stock Xen approach.

The fourth bar presents the UDP throughput achieved by a guest running on a MV without a driver domain (integrated driver). Clearly, the MV integrated driver guest outperforms the stock Xen guest in terms of UDP throughput by almost 70% (9.8 Gbps vs. 5.6Gbps), achieving near-line rate and identical throughput to the baremetal linux case as well as the stock Xen dom0 case.

Furthermore, we measure the number of transactions/packets per second that the network interfaces can transmit and receive in the same testbed. These numbers translate into the 1-way latency that the system is able to achieve. We plot these in Figure 10-3.

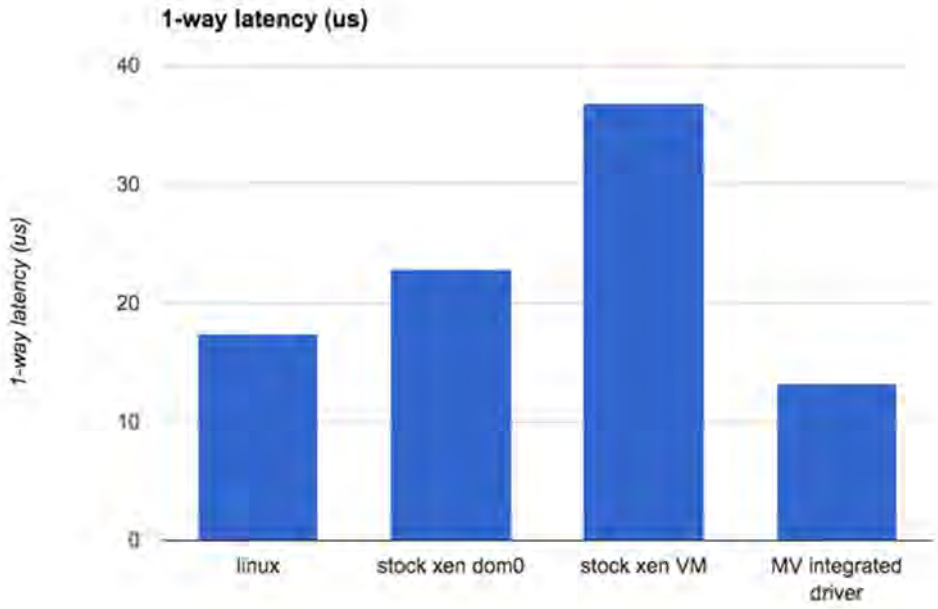


Figure 10-3: 1-way network latency for Linux, Xen dom0, Xen VM and MV integrated driver

Once again, we use the linux case as a baseline -- transferring one byte from a linux host to another one using the UDP protocol takes approximately 17 us. Performing the same benchmark in a guest VM running on stock Xen we see that it takes 37 us, almost 2x more. To validate the hypothesis that the source of overhead is network bridging, pci passthrough and Xen scheduling, we perform the same experiment as above, measuring the 1-way latency for dom0. This case is clearly better (22 us), but still over the baseline (30% more).

To showcase the advantage of using the MV integrated driver case, we measure the time that an ethernet frame with a 1 byte payload needs to leave the MV (hypervisor context) and reach an

opposite node (MV, hypervisor context). We find that the time needed is approximately 13 us. This provides a clear advantage over all the other cases, even baremetal linux. It is expected that the UDP protocol overhead regarding latency is not more than 1-2%.

10.2.2 REDIS Database Performance

To provide a real-life use-case scenario we deploy a popular, in-memory data structure store, REDIS and capture its performance from a remote client. We run the standard redis-benchmark tool, provided by the redis software stack. Network bandwidth and latency usually have a direct impact on REDIS performance.

Figure 10-4 plots the number of requests per second a specific REDIS instance can handle when deployed in a stock Xen guest VM and as a rump unikernel in a MV setup and in a stock Xen setup.

We can derive two important issues from Figure 10-4: first, the unikernel instance of the REDIS application provides some benefit in terms of request handling; second, the improvement in network latency and throughput plays a significant role as seen in the MV case.

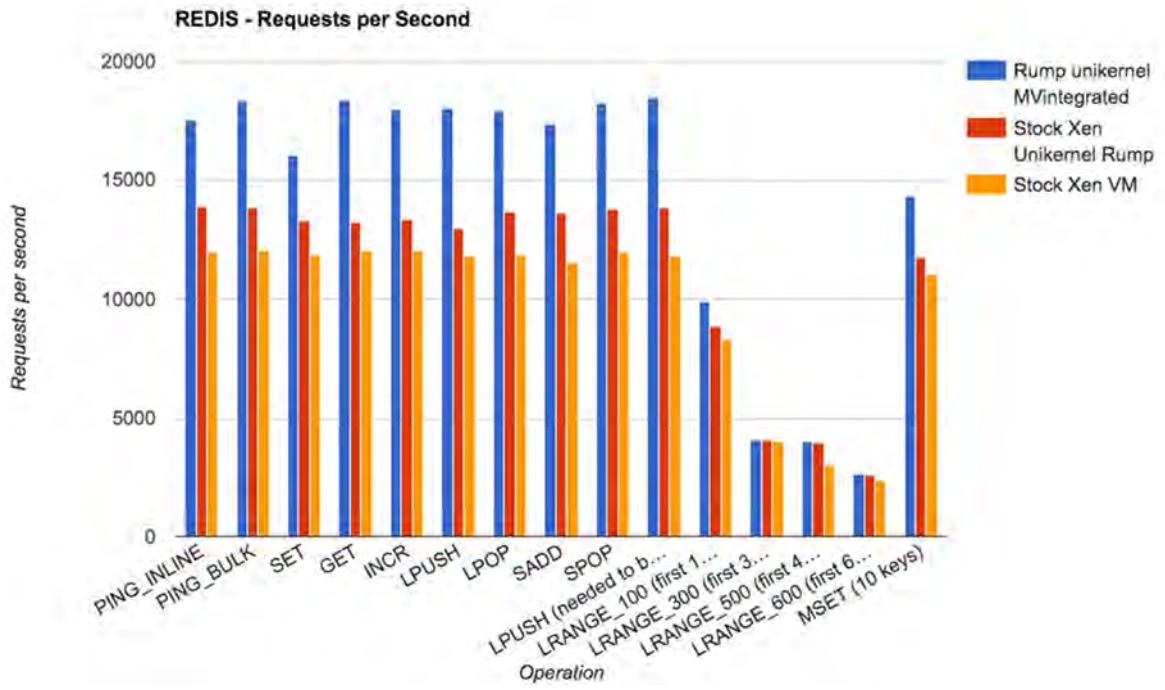


Figure 10-4: Redis DB - requests per second handled by RUMP Unikernel running on MV, Xen and standard Xen

Specifically, we highlight the difference of the two key sets of measurements of Figure 10-4 and plot the percentage improvement of the rump unikernel in the MV and Xen normalized with baremetal linux execution. These results are plotted in Figure 10-5.

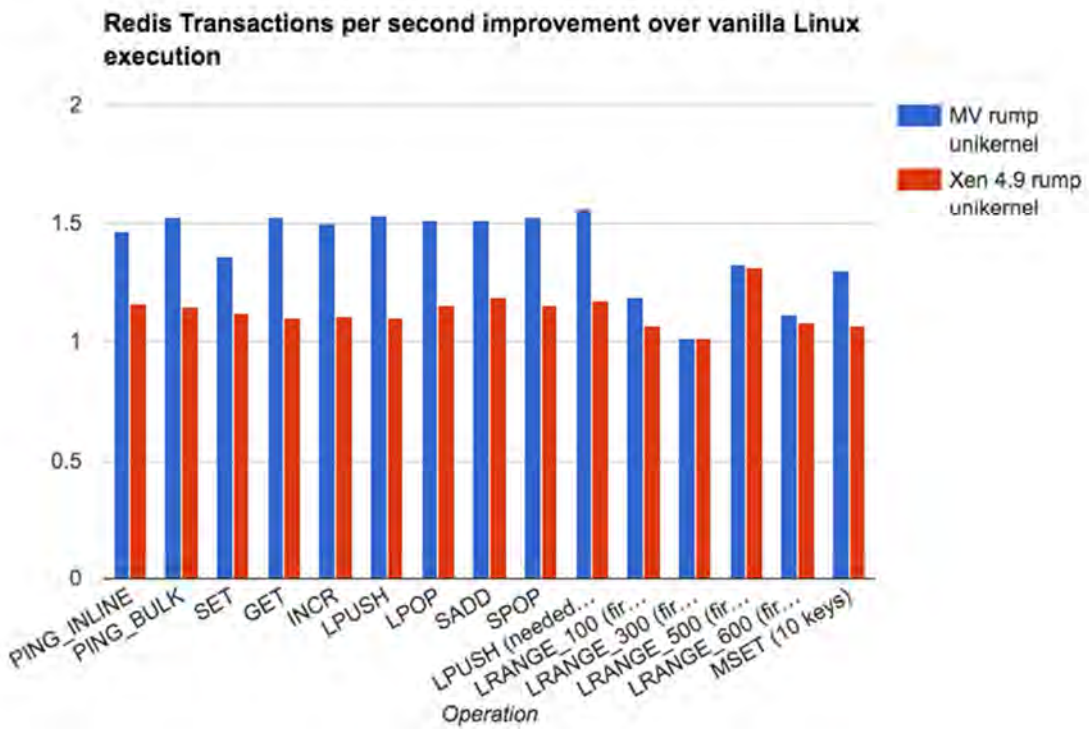


Figure 10-5: Redis DB performance, normalised against stock Linux for MV and Xen running RUMP unikernels

We see that for most of the tests, the rump unikernel ran in the MV is almost 50% better than baremetal linux execution. Compared to the stock Xen case, the benefit is almost 30-40%. For only one set of the tests, execution is exactly the same as in linux and stock Xen. In most of the cases, the improvement in network latency and throughput directly affects the performance. Given that the cpu/memory virtualization overhead is negligible in this kind of benchmark, and that the unikernels are pinned to specific physical CPUs, the performance benefit from running a unikernel instance of this kind of workload is clear.

10.2.3 Unikernel Performance

To highlight the IO performance improvement of using unikernels vs. traditional Linux virtual machines a comparison was performed using the redis database server as seen in Figure 10-6. In this evaluation, the number of transactions per second were recorded for the redis database between the client and server. For all three cases there is a server and client that run. In the case of the MiniOS test, the server is compiled into a Unikernel image that only runs the redis db server. In the Linux and localhost cases the redis db server runs in a traditional Linux guest VM. The difference between Linux and localhost is that in the latter the server and client run in the same guest. The redis client is always run as a native Linux application for these three scenarios. The main comparison point is the

performance difference between having the redis server on MiniOS (as a unikernel) and a Linux guest. For the first 12 tests (left to right) the performance in transactions per second for redis db running as a server is at least 15% greater in the Unikernel case than running as part of a generic VM. The performance deterioration for the list accesses has not yet been analysed but it could be the case that the raw IO access driver was improved between the server implementation in the Unikernel and the Linux case.

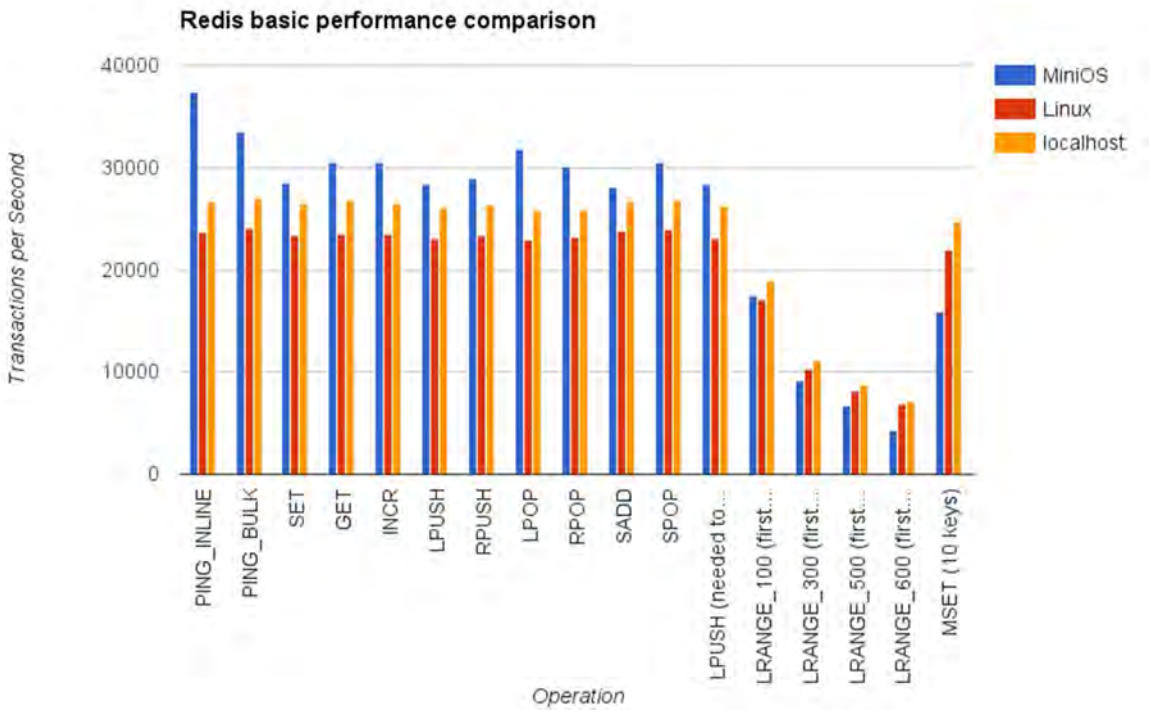


Figure 10-6: IO performance measured by Transactions per Second for Redis for MiniOS / Linux / co-located [1]

10.2.4 Network Performance between Nodes

As has been shown in Figure 10-1 the network throughput for the MicroVisor is very close to the raw physical limit, saturating multiple links when available. To continue this assessment and understand the throughput between clients running either on the same physical machine (IntraNode) or between machines (InterNode) the performance was measured as can be seen in Figure 10-7. The throughput for the InterNode case is fairly independent of the number of streams and is close to the physical

[1] Note:
LPUSH … : LPUSH (needed to benchmark LRANGE)
LRANGE_{n}… : LRANGE_{n} (first n elements)

capacity of the links. For clients that run on the same machine the performance scales as the number of streams increases.

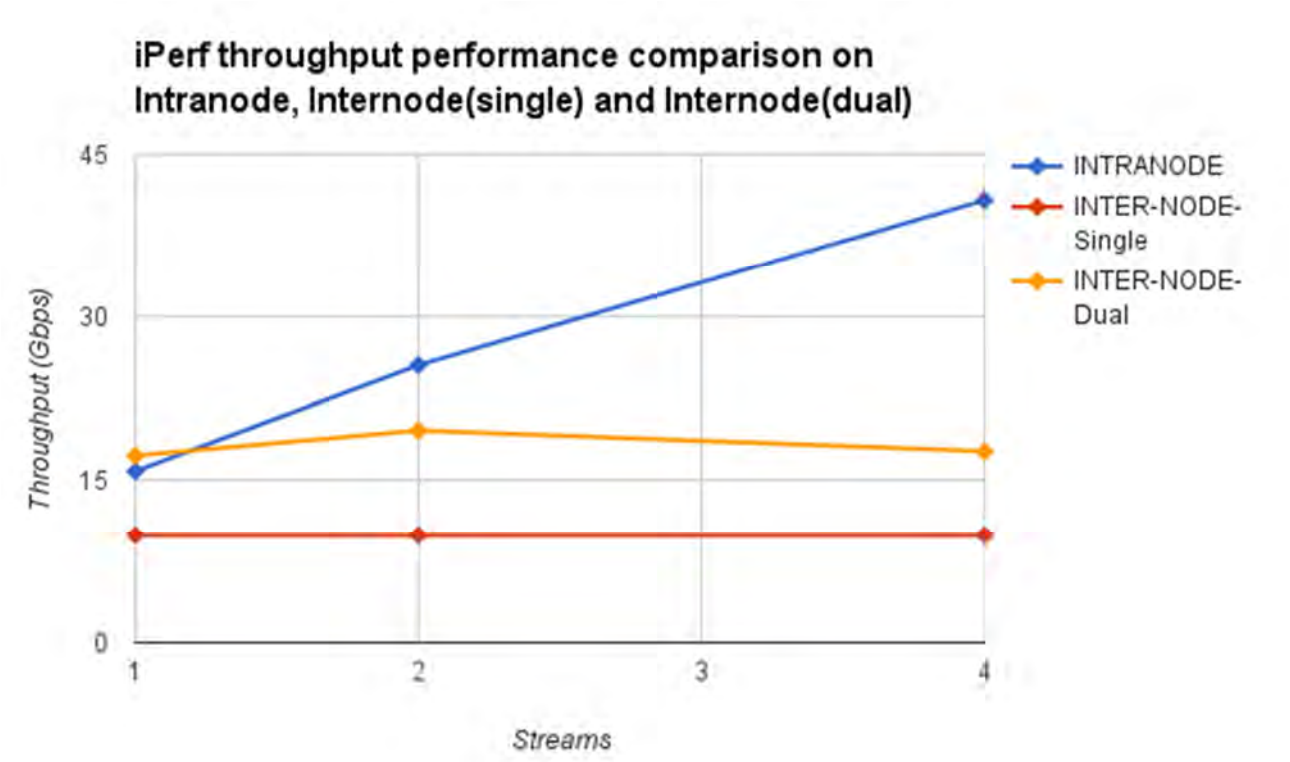


Figure 10-7: Network throughput measured via iPerf against number of streams for different MicroVisor configurations

One of the major benefits of the MicroVisor architecture is the improvement in latency between nodes. The Xen hypervisor on which the MicroVisor is based has a significantly higher latency for TCP responses than the unvirtualised (baremetal) latency. The MicroVisor adds minimal overhead for TCP latency as can be seen in the latency performance results as shown in Figure 10-8.

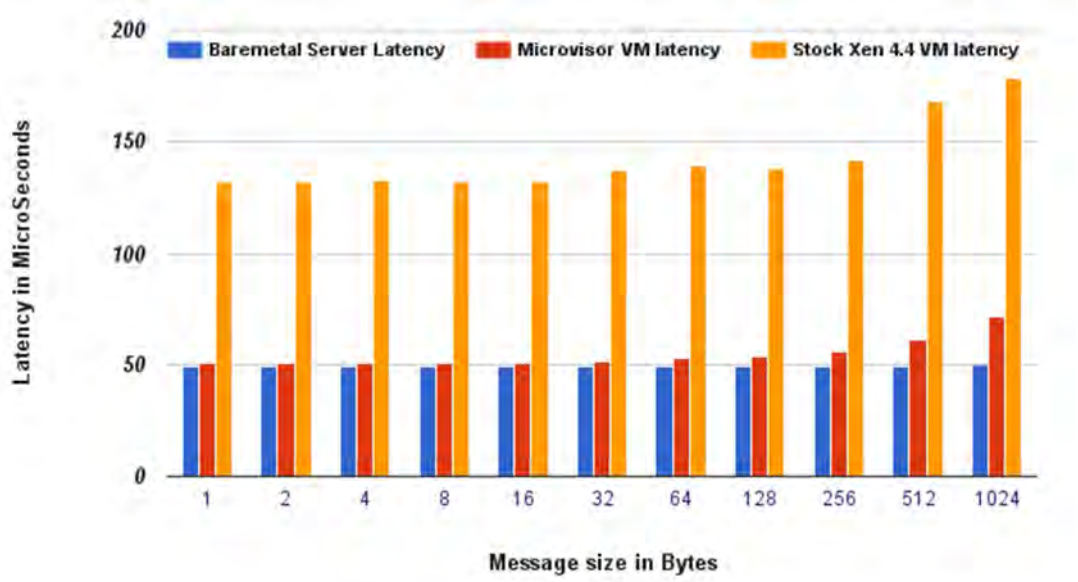


Figure 10-8: Communication latency derived from NPTcp vs packet size on Baremetal, MicroVisor and Xen 4.4

To understand the latency performance of guest to guest communication further assessment was carried out against different platform configurations as can be seen in Figure 10-9. For small packet sizes, the latency as measured by NPTcp is significantly improved on the MicroVisor platform. As the packet size increases the latency increases for all the different configurations but the latency increase is only small when the guests are co-located on the same physical machine (IntraNode). The latency for the Xen IntraNode case is roughly 3x slower than the MicroVisor IntraNode case for packet sizes below 1024 Bytes. The latency difference between two physical machines running baremetal Linux and two MicroVisors running VM to VM communication is minor, so the overhead incurred by using Virtualisation on the MicroVisor is minimal and offers close to baremetal performance for latency.

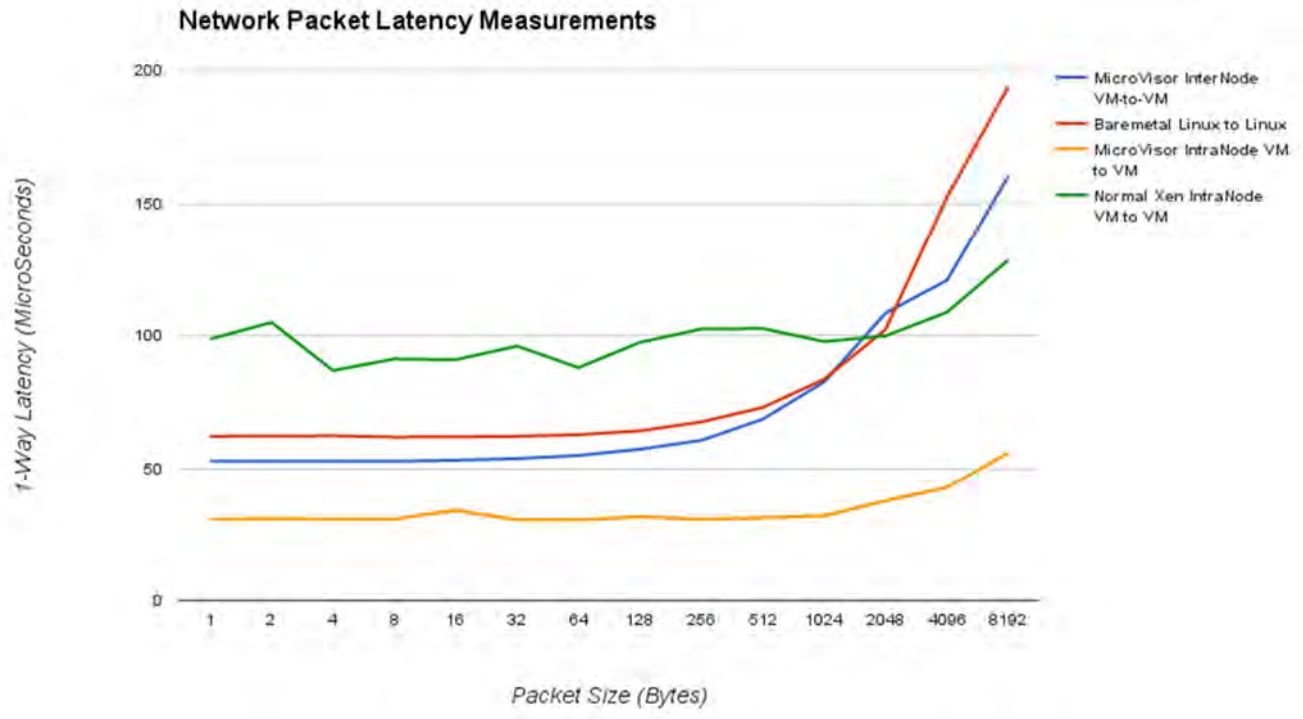


Figure 10-9: Network packet latency derived from NPtcp against packet size for different virtualisation configurations

In Figure 10-10 the network throughput performance of two guests communicating to each other in a setup with dual 10Gb Ethernet links is shown.

Before changes were made, MiniOS generally could not reach the full capacity of dual 10Gb Ethernet links. After that changes were made, the performance increased. Changing the configuration options of Xen though with certain configuration options lead to optimal performance. The best performance was seen when MiniOS had been modified and Xen had the options “multi-queue, next-queue if no slots and drop packets when no space” configured. The test failed when MiniOS was changed but Xen was not changed and hence there are no values for greater than one stream for that case.

Before the changes MiniOS was set up with two polling threads, one for the hardware to handle packets coming from the network whilst the netfront thread handled packets coming from the MicroVisor. For every received packet the transmit function of the complementary end was called for immediate forwarding. If there was no space in the other queue then the packet was dropped.

The changes to MiniOS included blocking the hardware polling thread when the received packets from the network was less than a level (64) and enabling interrupts. Once the interrupt thread was triggered the hardware polling thread was woken to ensure that while the network had more packets than the level the thread remained unblocked. It will be scheduled again after netfront’s polling thread yields. Each packet is placed on a queue and then packets are handled in chunks of 64 packets.

If the MicroVisor queue is full then the packet remains in the queue and waits for a slot. This then leads to trapping less times in the MicroVisor.

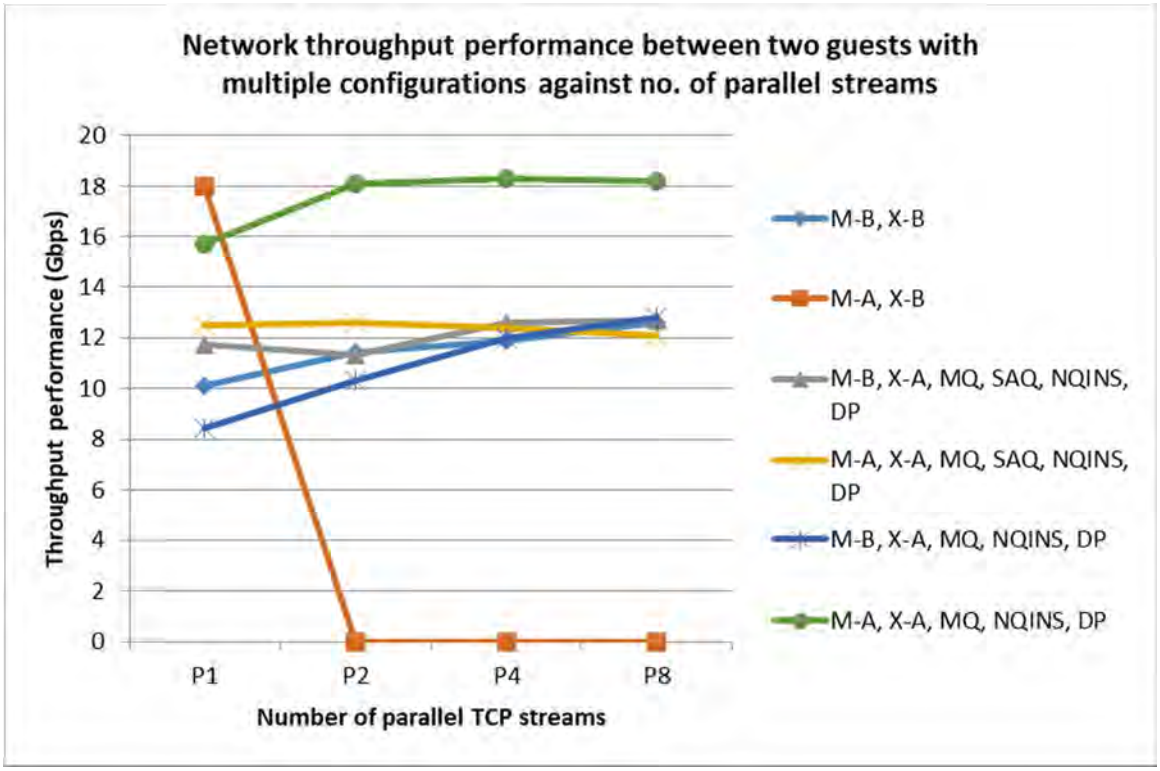


Figure 10-10: Network throughput performance when using different configuration options against no. of parallel streams [2]

Figure 10-11, shows the difference in latencies between a standard virtual machine, a virtualised function that is hosted on the MicroVisor platform and a function running on baremetal. There is a large performance improvement noticed on the MicroVisor platform with respect to the standard

[2] Note:
M-A/B = MiniOS after/before changes.
X-A/B = Xen after/before changes.
MQ = Multiqueue
SAQ = Stripe across queues
NQINS = Next queue if no slots
DP = Drops packets when no space

virtual machine. The performance overhead of the MicroVisor relative to baremetal is much lower than that of the VM.

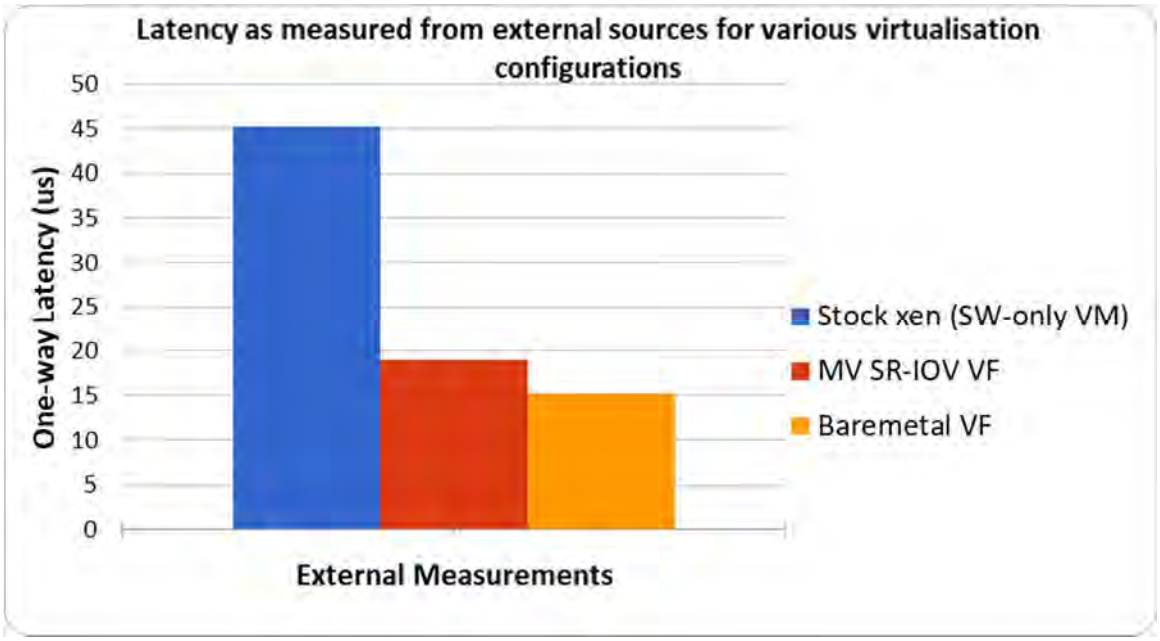


Figure 10-11: Latency as measured for various virtualisation configurations

Figure 10-12 shows a similar performance improvement of the MicroVisor relative to a function running in a VM. There is a slight performance improvement when the MicroVisor has SR-IOV enabled when compared to the native MicroVisor platform.

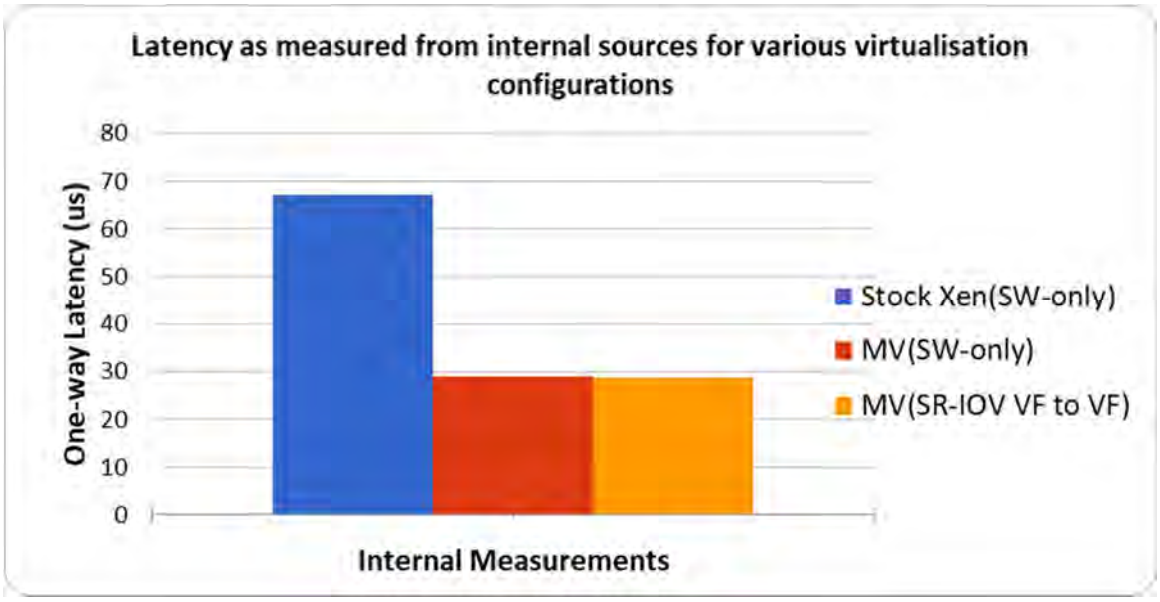


Figure 10-12: Latency as measured from internal sources for various virtualisation configurations

Figure 10-13 shows a comparison of the network throughput between a virtual machine running on Xen, a virtual function running on the MicroVisor platform and the baremetal performance. The main thing to notice is that the throughput in of a guest running on standard Xen is significantly reduced when compared to the baremetal performance. A virtual function running on the MicroVisor shows performance that is almost the same as baremetal performance. The performance difference between the MicroVisor and baremetal for a single TCP stream is currently unexplained and it is expected that performance improvements can be realised to more closely match the baremetal performance for a single TCP stream.

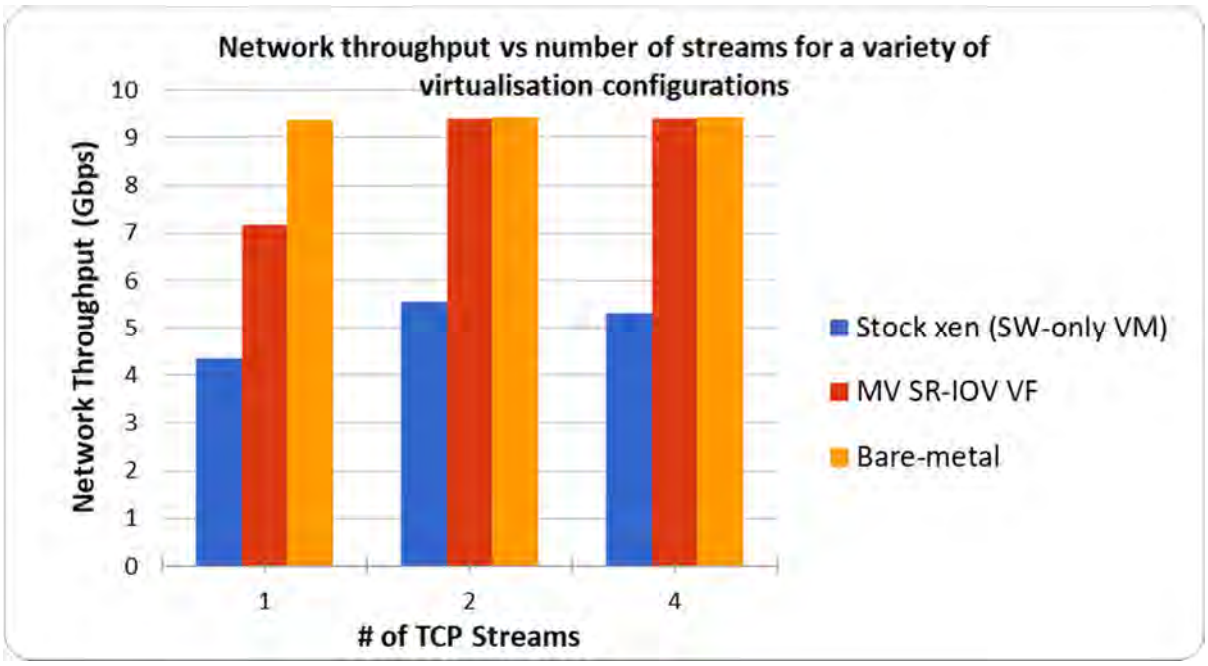


Figure 10-13: Network throughput vs number of streams for a variety of virtualisation configurations

11 Conclusions

In this deliverable we have described the full implementation of the Superfluid platform, including how all of its mechanisms come together to provide unprecedented levels of network service dynamics and performance. To the best of our knowledge, this project is the first to be able to show that it is possible to run virtualized functionality (whether network-based or not) whose performance is on-par or better than so-called lightweight virtualization technologies such as containers, *without* having to compromise on isolation or security, items which are crucial to the operational environments that the project targets for deployment. To give a few numbers, the superfluid platform is able to boot fully isolated, virtualized instances in as little as 2-4 milliseconds (akin to process instantiation) largely irrespective of how many instances are running; or can run as many as 8,000 concurrent instances on a single, inexpensive, off-the-shelf x86 server. To the best of our knowledge, this is the first work to report such kind of achievements, all of the while supporting the large traffic rates (10Gb/s rates) that are needed for NFV deployments.

One of the main pain points for achieving such numbers has been the largely manual and time-consuming development of lean, customized unikernels. To remedy this, we have started, as future work to this deliverable, to develop an automated tool to generate such specialized operating systems and guests. This tool, called Unikraft (https://www.xenproject.org/developers/teams/unikraft.html), is still in its early days but is already an open source project under the auspices of the Xen Project and the Linux Foundation; we are planning on basing the project's Hackathon around it in order to raise awareness of it and to get the wider community to contribute to Superfluidity's results beyond its lifetime.

In sum, our hope is that the results in this deliverable will help develop new paradigms, where, for instance, services are instantiated just-in-time, as the first packet of a flow arrives, or to be able to have per-customer instances even when using a single, off-the-shelf inexpensive services. The widespread dissemination of at least a large set of the platform's mechanisms, along with their release as open source, should hopefully go a long way towards influencing the future development of high performance virtualization platforms.

12 Appendix: NFV container management analysis

In this section, we describe our findings on how we could operate and manage Network Functions inside containers. There are many different management options for containerized applications, but we demonstrate that NFV applications have some unique features that render normal management solutions currently incomplete. Following a mixture of documentation review, simple implementations and discussions with the wider container community we identify certain solutions that could be developed appropriately and also provide a more detailed review of these approaches.

12.1 General container management systems

12.1.1 Future NFV architectures

In order to assess the management tools, we need to consider their operational environment. The easiest situation to envisage is that containers are simply used as a replacement for a VM NF. These virtual functions are located in large, managed data centres and need to support the networking of medium to large enterprises. These containers would therefore exist on a per-customer basis and we may wish to ensure the quality of experience as we do today with VMs, through over-provisioning and resource pinning. In this case, the fast spin up times of containers are likely to be restricted to handling of failures. The hardware is likely to be fairly homogeneous.

Going forward, we may look at providing increased flexibility for these customers giving them the ability to scale out some of the functionality, perhaps using a micro-services architecture. To date, what micro-services really mean for NFV is still an open issue. Nevertheless, we can understand that micro-services will lead to greater dynamics, requiring the need to flex not just the containers, but the container networking.

Getting more complex, we may wish to own the containers and the platform or we may be providing a container platform for other tenant container providers. A simple architecture for this just provides a VM per tenant allowing them to use containers; but going forward we may like to consider a container-native solution as it could allow for much greater efficient resource utilisation.

At the same time, we need to be able to provision the containers at specific, remote customer locations. The hardware here is likely to be more heterogeneous. There is a greater chance that the customer will be configuring and using the hardware separately from anything the operator is doing.

Finally, in a “fog” scenario, the required functionality would be instantiated at the “best” location based on the required customer experience and resource utilisation. Resources could be available within the network edge as well as in remote data centres or inside customer premises. The

functionality may also need to follow the customer movement between different locations, introducing the need for efficient handover mechanisms.

The specific issues with containers compared with VMs are that containers tend to be smaller, more dynamic and ephemeral than VMs, with the ability to trade performance and isolation against resource usage. In addition, containers tend to be configured at build or run time, unlike VM NFs which may be configured as a hardware box once launched. NFV applications also have many differences to conventional container applications. NFV applications include security functions such as firewalls and intrusion detection, performance enhancing and measuring functions as well as basic network functions such as routing. Unlike common container applications:

- NFV applications typically require multiple network connections, often with predefined static network addresses. Multicast network support may also be needed. Integration with SDN will become increasingly important
- NFV applications are frequently stateful
- Geographic function placement and potentially limited network connectivity are much more significant. In addition to providing NFs in the data centre, we must consider both the customer premise and the future fog (where the container location is chosen dynamically based on end to end network performance and utilisation).

12.1.2 Summary of management tools investigated

The table below summarises our findings of a number of different container management solutions when applied to NFV applications. In the next section, we look in more detail into Kubernetes and Ansible.

Table 7: Strengths, weaknesses and comments regarding various VIM management systems

TOOL	STRENGTHS	WEAKNESSES	RECOMMENDATIONS
Swarm	Start Docker over a selection of hosts	Very limited functionality and configurability. Docker only	Discounted –small scale and propriety
Kubernetes	Schedule/deploy a group of related containers.	Very basic connectivity. Best for homogeneous	Detailed investigation as the community is open to new ideas

		infrastructure. Docker only	
Openshift	GUI to manage containers, providing load balancing, auto-scaling, recovery, resilience, upgrades with roll-back	Relies on Kubernetes,	Reliant on changes to Kubernetes – re-examine once Kubernetes networking is uplifted
Openstack	A dominant complete and mature VM solution looking to integrate with Kubernetes	Heavyweight, complex, data centre focussed. Containers not equivalent to VM – relies on Kubernetes	Recommended for further investigation due to its compatibility with VM NFs
Mesos/ marathon	2-level scheduler, deployment, load balancing, scaling, recovery & resilience	"Lacks coherent network story". Multiple IPs per container not possible.	Discounted as no plans to change networking
Ansible // Tower	Flexible yet simple and lightweight scripting and inventory management toolset designed for multi-tenant.	No scheduling mechanism, limited automatic monitoring	Detailed investigation. Has proved useful in demos. Community open to new ideas
Rancher	GUI to manage containers, on top of schedulers such as Kubernetes, adds image catalogues, authentication tools and common services (load balancer)	Targeting enterprise market-place. Docker only. Inventory management could be awkward with many systems	Useful for small scale demos
OSM	Strong architecture	immature – not launched in time for investigation	Recommended for further investigation at a later date

12.2 Key Technologies

Here we delve deeper into the most important tools, presenting conclusions from deployment experiments.

12.2.1 Ansible

12.2.1.1 Overview

Ansible is an open-source command-line solution to help manage systems and application deployment across multiple systems (system configuration). Redhat's Ansible Tower is a commercially available product[3] that provides a GUI to Ansible and adds some features such as the ability to schedule events, attribute changes to particular users and provide event notifications.

12.2.1.2 Understanding Ansible

The core of Ansible is the playbooks which describe the desired system state, and the inventory of systems that you are managing. When a playbook is run, Ansible compares the required state to the defined state and modifies the system as required. Where state managing modules are not available, simple commands can be executed, although then care will be needed to manage all outcomes. Note that whilst the basis is state checking, it is essentially a combination of declarative and imperative behaviour.

The inventory can be a very simple list of systems or dynamically created e.g. from a set of cloud resources or a database. Variables can be associated with the systems or groups of systems. Within the Inventory, hosts can exist in groups (eg "Web-Servers" and "Docker-Hosts"). Systems can be in multiple groups; and groups can be created from other groups. Playbooks are then run against systems in the inventory – for example we can run a patching playbook against all "Web-Servers" that are not "Redhat" servers. Tower allows us to set up templates to facilitate the running of playbooks – for example prompt for specific variables or specify specific systems. It also allows the scheduling and execution of playbooks.

[3] £46 - £70 per year per managed host, standard support level

We showed how we could use Ansible to provide containerised advanced fault analytics on edge CPE and also to build and deploy a Docker Quagga router, such as might be useful within a data centre deployment. Here, the fact that Ansible can be used for many more things than just containers was very beneficial, as we could connect the routers to OVS switches within the same playbook making it easy to track container connectivity.

Ansible requires no agent on the managed device, but does need SSH[4] and a way of managing keys. To control Docker (and probably to run other modules) some python code is also needed.

12.2.1.3 Ansible Conclusions

The main advantages of Ansible are its low complexity and its good documentation. It is lightweight, only requiring SSH access to the hosts. It allows us to integrate network and application management within a single playbook. It has a large number of predefined modules, but also allows us to execute basic commands on any system. It allows us to define actions to take on playbook errors, and can be used to deploy a staged upgrade (rolling upgrade) across systems to avoid service outage. Tower could be used to ensure that container images and system patches were kept up to date. Ansible needs to be combined with other tools to provide a complete management solution.

It does not have a scheduler. Whilst it is easy to fix the geographical location through use of the inventory, it is less easy to choose to deploy on a server with lots of free resource for example. If we want a deployment of 1 to N instances, Ansible simply places the instances on the first matching systems in the inventory list. Whilst customised rules could be built - dynamically updating the inventory with system load information for example, for a pure container workload, instead a separate Kubernetes (or other) scheduler could be run, through which Ansible could deploy containers.

Tools will also be needed to help provide the operational features such as rapid failure detection and response to load. Whilst Ansible can deploy systems with a load balancer, it does not manage the service performance, for example scaling systems automatically in response to high load (although an underlying scheduler could provide this). With Tower, it is possible to schedule checks that a system is still in the required state, but this is may be a heavyweight approach if looking at identifying faults in sub second timescales over millions of systems. Another approach is to constantly pull/push metrics and analyse these for unusual behaviour using telemetry applications such as SNAP.

[4] A few other methods of connecting to hosts are possible


For NFV, one key question is how to reconfigure a service chain with minimal service downtime. When working at large scale, the different systems may be in different initial states (e.g. power failure or a local manual reconfiguration). Further experimentation is needed to understand how to manage this.

Overall Ansible and Tower are very useful tools that will likely be used in applications beyond just containers. They allow for the deployment of SDN and containers, as well as providing integration points with scheduling and monitoring solutions.

12.2.2 Kubernetes

12.2.2.1 Overview of Kubernetes

Kubernetes is an open source system for the deployment, scaling and management of containerised applications. It consists of a master node, with additional software installed on each Docker host to make a Kubernetes cluster. Although it is not yet suitable for general NFV applications, it is in scope because of its popularity within the container world and the willingness of the community to develop the system.

Some key concepts of Kubernetes are:

1. Pods. A group of one or more related containers than share a network namespace and are best co-located. A Pod may be a group of micro-services that provide a service. Multiple identical pods may be deployed depending upon the expected load.
2. Service. A construct that can load balance across pods. The service address does not change if a pod changes host. This service endpoint may be a load balancer.
3. Networking plug-ins: There are a number of different systems that can be used to provide the networking between the Kubernetes elements (i.e. between the containers). Each networking option operates in different ways in terms of the use of NAT, tunnels and IP addresses. Note that the method used to reach the containers externally is independent of this internal cluster networking mechanism.

12.2.2.2 Installation of Kubernetes

The technology is immature, rapidly changing and documentation weak. For example, it took several attempts to successfully deploy a Kubernetes cluster. The first successful build was on Centos 7 bare-metal using custom build with the flannel network option. This lets us run Kubernetes, deploy containers from our own repository, and use our own defined network address pool. Other successful

builds, all on Centos-7 have since been made using the alpha release Kubeadm system. Kubeadm[5] is a utility that helps an admin to set up a Kubernetes cluster by setting up role based access control (RBAC).

12.2.2.3 Understanding Kubernetes

The interface is command line driven and scrappy (the new web interface is immature). The --help option leads to a stream of information making it unsuitable for beginners. There are many ways that things silently fail. For instance, "kubectl run" may originally appear to have run successfully, however when the system is later queried it is possible to discover that there was a typo in the image name which means that the container has not actually been deployed.

Kubernetes' main role is as a tool that lets us schedule containers across multiple hosts. Kubernetes automatically schedules pods to the less loaded system based on CPU and memory load. This may not always be the best choice – for example if we wish to dynamically flex the number of available hosts to manage our power consumption, then running containers on the fewest number of systems is preferable.

The scheduler is policy configurable, for example by giving a node a label of `disk_type=ssd`, we can then specify that the Pod must run only on nodes with `disk_type=ssd`. The Kubernetes team are developing this use of labels, for example to allow "*`prefer` disk_type=ssd*", or "*`not` co-located with specifically `labelled Pods`*". There is no (obvious) inventory file or database that can be used to separately manage the labels associated with hosts. Labels could be used to give geographic meaning to a node –for example *location=phone number*. However, based on the experience of using the platform, Kubernetes is unlikely to work well over a very distributed network, with constant chatter between the hosts and master.

In addition to scheduling, Kubernetes helps with replication, checking the status of containers and hosts and updating images by gradually taking replicas out of use and re-starting with the new image. We can scale an application by (manually) changing the number of replicas. If a machine dies, Kubernetes will restart the containers on a different host, and ensures that the network implications of this change are hidden. This recovery process is slow. The working assumption of Kubernetes is that a high availability service will have many replicas, so that in case of isolated failures the service remains alive, just slightly under-resourced.

[5] https://kubernetes.io/docs/getting-started-guides/kubeadm/

Many network functions are stateful. Whilst stateful applications are possible, this is not a primary assumption for Kubernetes. The basic instructions for stateful applications assume no replicas.

12.2.2.4 Kubernetes Networking

Kubernetes relies on additional tools to implement the required networking. The network model is still considered to be weak[6], but this is an area that is being actively developed. Recent communications suggest that many of the issues will be resolved over the next six months.

It starts with the assumption that all containers and hosts in a cluster will be able to communicate with each other without the need for any NAT, although tunnelling is common. This can be achieved by allocating a routable IP address per pod (e.g. using private IPv4 addresses)[7]. The network plugins may also support policy control to e.g. prevent some pods talking to other pods. The status of IPv6 Kubernetes is unclear[8], but solid IPv6 might make a clean network solution possible.

These addresses change if the pod moves host, and multiple pods delivering the same service will have different IP addresses. To ensure that a service remains ***reachable,*** Kubernetes assumes that there will be a single address –the service_cluster_ip - that will always be associated with the service. Any item in the cluster can access the service using this IP address. The implementation of the service cluster IP address (i.e. mapping from service_cluster_ip address to the possible Pod addresses) is often described in relation to a cloud load balancer, such that this address is an address/port pair from the load balancer which will balance traffic between the available service endpoints (i.e. between the different IP addresses of the replicated pods). With our private cloud, we use a "NodePort" address[9]. This maps the same specific port on every host to pods implementing the service using iptables and the kube-proxy which implements (to the best of our understanding) a TCP-split proxy service. The service can then be reached via any host IP address and the service specific port. If the host IP addresses are public, this gives external reachability to the service.

Tests show that NAT and tunnelling can be avoided, but we noticed packet duplications and retransmissions. It appears that this is a known problem[10]. Again, this is simply evidence of immaturity of the solution currently.

[6] http://www.devoperandi.com/load-balancing-in-kubernetes/
[7] Running Flannel with the "Backend" in the flannel.yaml file set to "host-gw" and checking the routing tables of all devices on the same subnet can avoid tunnels if all hosts are on a common physical subnet.
[8] https://github.com/kubernetes/kubernetes/blob/master/docs/design/networking.md
[9] It is not clear if we use the kube-proxy split TCP connection if there is a kuberentes load balancer available
[10] https://github.com/kubernetes/kubernetes/issues/27489

12.2.2.5 Kubernetes Conclusions

Overall we could say that Kubernetes is fragile (many component parts), complex (many parts with poor documentation) and immature. However it is one of the more mature and useful container management and scheduler systems currently available.

Today, Kubernetes would be useful in situations where we might have a homogeneous set of co-located hardware resources on which we want to run reliable services that only require single public interfaces with rapid scale out. Examples are therefore found in control plane services, for example within mobile networks.

The Kubernetes community is pro-active, and we have begun interaction with them to drive changes. We understand that they expect to support i.e. multiple public network interfaces, IPv6, different addressing models and multicast in the near term. This would greatly improve Kubernetes for NFV applications.