SUPERFLUIDITY

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

Research and Innovation Action GA 671566

DELIVERABLE I5.2.:

MECHANISMS FOR NETWORK SERVICE DYNAMICS AND PERFORMANCE

Deliverable Type:	Report
Dissemination Level:	CO (I5.2)/PU (D5.2)
Contractual Date of Delivery to the EU:	M12 (June 2016) 30/06/2016
Actual Date of Delivery to the EU:	M12 (June 2016) 30/06/2016
Workpackage Contributing to the Deliverable:	WP5

Editor(s):	ONAPP
Author(s):	CNIT (Stefano Salsano, Pier Luigi Ventre, Nicola Blefari Melazzi), CITRIX, **NEC (Filipe Manco, Felipe Huici,** Giuseppe Siracusano**), ONAPP (John Thomson, Michail Flouris, Anastassios Nanos, Xenia Ragiadakos, Julian Chesterfield),** PTINS (ALTICE), REDHAT, TELCA, TID, TUD, UPB
Internal Reviewer(s)	Stefano Salsano (CNIT)

Abstract:

Keyword List: Profiling, performance, benchmarking. Virtualisation.

INDEX

List of Figures

List of Tables

Glossary

(TO BE FILLED OUT IN THE FINAL VERSION OF THE DELIVERABLE)

SUPERFLUIDITY DICTIONNARY	
TERM	DEFINITION

Table 1: SUPERFLUIDITY Dictionary.

1 Introduction

This Interim Deliverable records the first six months of effort of Task 5.2 “Network Services Dynamics, Performance and Scalability” and has been edited by ONAPP. The final deliverable is due to be completed 12 months from this Interim Deliverable and will be prepared by NEC. As such contributions are incomplete at this stage and the emphasis for someone reading this document is to understand the methodology and aims for the Task and to understand how it fits in with the scope of the rest of SUPERFLUIDITY.

1.1 Deliverable Rationale

The emphasis on Task 5.2 is to create the capabilities for virtualised network services running on the SUPERFLUIDITY platform. This work therefore focuses on the platform improvements and optimisations needed to enable network functions (see use cases in WP2, specifically D2.2) and Re-usable Function Blocks (RFBs) as described in the SUPERFLUIDITY architecture (see I3.1 for the current version at time of delivery and D3.1, which is due at the same time as D5.2).

The focus therefore is to develop support for extremely light-weight virtualisation techniques that allow network functions to be run independently and isolated from each other but that can then be chained together to form more complex services, see Service Function Chaining in D/I3.1. The motivation for light-weight virtualisation techniques as opposed to legacy virtualisation methods is that it allows for greater consolidation (more services can be located at the optimal location), greater scale (more services for a given set of resources), easier deployment (short boot up times) and better manageability (orchestration of a large number of workloads across a large set of devices).

Given the larger bandwidth and capacity envisaged for the increasing number of devices in 5G-PPP the core and backhaul network will need to utilise faster links and respond to services faster. In order to respond

to network packets and user-load dynamically, we will have to move away from pre-allocated static VMs to fast deploying service responders.

In recent years, light-weight containers and specialised Unikernel systems have been developed to bridge the gap between the demand and the legacy virtualisation systems. There is invariably a trade-off as the systems get smaller as they become more specialised for a certain function. This has been the case for the divide between hardware and software for many years, with functions that are specialised and requiring high performance being designed and implemented in hardware but at the cost of being hard to modify and requiring a long deployment time. The same choices are becoming apparent for software virtualisation techniques, with the trade-off between design, build, configuration and deployment time being compared to the performance.

The vision of this task is to tie in with the KPI mapping in WP4 and orchestration elements in WP6 to allow for the assessment of workloads running in a platform and to then determine whether a given workload is better suited in a specialised light-weight VM or in a generic VM and calculating the cost of reconfiguration. The platform support and analysis will allow a comparison of the performance for different types of workloads in different execution/virtualisation environments.

When the performance of light-weight containers and specialized Unikernels increases, in terms of reducing the instantiation and boot time up to the order of 10s of ms or even less, the performance of Virtual Infrastructure Managers (VIMs), that control them may become a critical bottleneck. Therefore, this deliverable also considers the VIM performance, by defining models for their characterization and by identifying solutions for their improvements.

1.2 Quality Review

Review Team member responsible of the deliverable: Stefano Salsano (CNIT)

VERSION CONTROL TABLE			
VERSION N.	PURPOSE/CHANGES	AUTHOR	DATE

0.01	Initial Document	John Thomson (ONAPP)	2016-05-10
0.02	Merging text contributions	John Thomson (ONAPP)	2016-06-21
0.03	Preparing sections for review	John Thomson (ONAPP)	2016-06-23

1.3 Executive summary

1.3.1 Deliverable description

This internal deliverable will carry an initial survey and measurements of available virtualization technologies (e.g., type-1 hypervisors such as Xen, type-2 hypervisors like KVM or lightweight virtualization mechanisms like containers) to see the trade-offs between them and which are more suitable to achieve the platform goals of quick instantiation, migration and massive consolidation, to name a few. This deliverable will further evaluate such mechanisms on different types of commodity-hardware, from full-fledged x86 servers down to microservers. Finally, the deliverable will consider the performances of Virtual Infrastructure Managers (VIMs) that control the virtualization technologies.

Task description – T5.2

Description: This task will design and implement next generation, superfluid capabilities for virtualized network services running on the SUPERFLUIDITY platform at speeds of 10Gbps and above. More specifically, we will provide mechanisms for near-instantaneous reconfiguration (e.g., in milliseconds, such that changing or even moving the processing between servers is transparent to end points), quick instantiation (e.g., we could target the ability to instantiate network processing on a per-flow basis, or on a as-needed basis, as new flows arrive) and dynamic scalability at tiny timescales. To achieve these goals, the work will look into specialized virtual machines and OSes, but also into lightweight virtualization techniques such as containers (e.g., LXC or FreeBSD Jails). Further, the task will target massive consolidation of network services, perhaps thousands or even higher number of VMs on a single, inexpensive commodity server, with the aim of (1) severely


reducing the platform operator's operating and electricity costs and (2) improving the scalability, and thus the fluidity, of the platform.

To complement the migration of processing we also need to coordinate the migration of network traffic. To this end we will rely on Openflow/MPLS for routing inside operator networks, and will also integrate VM migration and cellular offloading mechanisms with endpoint mobility, such as Multipath TCP.

In the RAN part, we will study and implement dataflow engine with dynamic allocation and mapping capabilities for Cloud-RAN applications enabling simultaneous operation of multiple virtual baseband stacks developed in WP4 on same SUPERFLUIDITY platform. The engine will cope with modularized, scalable and heterogeneous hardware platform by integrating scheduling algorithms and methods of T5.1 allowing dynamic mapping of abstracted task graphs to particular hardware arrangement and configuration. The engine will make use of standard parallel programming interfaces (e.g. POSIX) for the purpose of seamless portability and dynamic instantiation within SUPERFLUIDITY platform. The main motivation is to focus on engine architectures allowing dynamic adaptation and configuration to particular workload characteristics which is inevitable for Cloud-RAN applications.

1.3.2 Summary of results

These will be produced for the final iteration of the Deliverable that is due in June 2017 (M24).

2 Background

2.1 Virtualisation approaches

There are many different Virtualisation approaches that can be used including the use of Containers, Linux Chroot / jails, Unikernels as well as a spectrum of Virtual Machines.

At the root of all virtualisation techniques though is an abstraction between the underlying physical resources and the logical or virtual resources that are presented to environments for running.

Linux Chroot and jails rely on process based isolation and allow multiple processes to run on the same kernel without being able to interact with the behaviour of other processes. Unikernels are specialised at compile time to run a particular process with possibly multiple threads. It removes the memory management aspects and process overhead that is present in a multi process OS. This removes a lot of the overhead that is present in todays generic VMs.

An effort that is being carried out as part of SUPERFLUIDITY is to analyse the performance of Linux VMs that are purpose built for running a specific application. This differs from UniKernels that require the application be rebuilt. An application or service is chosen and then only the relevant parts are moved into a bare file-system. Analysis is then performed to determine the static and dynamic linked libraries that are accessed by that application and then build up from there until the minimal OS with the application is running. This does not optimise the individual components that are present in the resulting VM but allows for Linux applications (ABI – application binary interface) to run without needing extra functions and without incurring the extra costs of converting the applications into a new programming language. This effort is described in the ‘TinyX’ description that will be included in the final report.

2.2 Virtual Infrastructure Managers

In the ETSI NFV MANagement and Orchestration (MANO) reference architecture, the entity that deals with the management of the virtual computing resources is called Virtual Infrastructure Manager (VIM). The VIM is in charge to instantiate a Virtual Machine / Container / Unikernel under the overall control of the Orchestrator. The VIM controls and manages the virtual resources in the Network Functions Virtualisation


Infrastructure (NFVI). Existing Open Source solutions, designed for less specialized Cloud infrastructures, provide a solid base to build VIMs for NFVI. However, a deeper analysis reveals that these do not support Unikernels and that there is room for tailoring the instantiation process to the NFV scenario to enhance its performance.

3 Methodology

Currently there are two forms of benchmarking that have been carried out related to the hypervisor platforms. The first approach has been to look at systematic platform benchmarking that has been carried out for low-power, embedded, ARM-based platforms that is described in Section 3.1. The second approach, described in Section 3.2, relates to performance optimisations and effort in improving the MicroVisor platform.

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 interaction of hypervisors with Virtual Infrastructure Managers (VIMs) is considered in Section 3.3, we discussed the VIM instantiation process and proposed a generic reference model, starting from the analysis of two Open Source VIMs, namely OpenStack Nova and Nomad. We had to modify these VIMs to instantiate Micro-VNFs based on the ClickOS [1] Unikernel. We realized performance evaluation tools for the two VIMs to measure VM instantiation times. Finally, we designed and implemented tuning of the VIMs to reduce the instantiation times.

In the full version of the Deliverable we will also describe how the CRAN use-case has been aligned with the software effort in the VM and hypervisor environments. Aligning the software stack with the Virtual Machine and the hypervisor will help ensure that the software is running efficiently on the underlying hardware platform. The orchestration system and decision logic will determine where to place the workloads.

3.1 Systematic platform benchmarking

Single Board Computers (SBC) Benchmarking

The benchmark has two main goals:

1. compare the performance of the available single board computers on the market and
2. analyse the performance penalty of running different types of virtualization technologies on each of the boards.

We analyse performance in 4 different dimensions: CPU, RAM, Network IO and Block IO. Each of these different benchmarks are detailed later.

Running this analysis requires a combination of the different benchmarks, platforms and boards. The number of possible combinations is quite big, so to simplify the process each benchmark is completely identified by a quadruplet of the type '<benchmark>.<os>.<platform>.<arch>'.

<arch>: The benchmark targets the three major architectures available on the market of single board computers: 'x86_64', 'arm32', and 'arm64'.

<platform>: Platform is taken as a generic term for whatever provides the execution environment for the benchmark. We consider 4 different platforms: 'linux', 'docker', 'xen' and 'kvm'.

<os>: This is the guest operative system used when running virtualized on top of 'xen' or 'kvm'. As one of the goals of the benchmark is to analyse the penalty of running virtualization technologies, we tried to keep the guest operative system as lean as possible to show the best case scenario. With that in mind we selected two guest OSes to run on:

- 'Rumprun'
  Rumprun is a unikernel built on rump kernels. Due to the number of optimizations made possible by unikernels, a unikernel benchmark should give us the best performance. We choose `rumprun` for two reasons: (1) it is very simple to build applications on top of it, and (2) it is compatible with `xen` and `kvm`. Unfortunately `rumprun` is still not ready to be build to for `arm32` or `arm64` platforms, so it for the time being only used on `x86_64`.
  https://github.com/rumpkernel/rumprun
- 'Linux'
  Linux is used to provide a baseline result, given is the most commonly used operative system in this context. Given the resource restrictions common on single board computers, we used a

minimalistic Linux build called ‘tinyx’. ‘Tinyx’ consists of purpose built kernel, only including the necessary bits and pieces to run on the desired platform, and a purpose built filesystem containing only the necessary binaries to run a specific application.

<benchmark>: The benchmark application used to measure a certain dimension mentioned above. We are using ‘himeno’ for CPU, ‘stream’ for RAM, ‘iperf3’ for network IO and ‘ioping’ for block IO. Each of these is detailed below.

- Himeno [CPU]
  Himeno measures the performance of solving Poisson's equation using the Jacobi iteration method. The result is given in MFLOPS.
  http://accc.riken.jp/en/supercom/himenobmt/

- Stream [RAM]
  Stream measures is a basic memory benchmark that measures sustained memory bandwidth. Stream can measure bandwidth in multiple different ways (using different operations and different conventions to count bytes), for this benchmark we consider what's called ‘bcopy’. For ‘bcopy’ stream counts the number of bytes that got moved from a region of memory to another over a certain period of time. The result is then given in bytes per second.
  https://www.cs.virginia.edu/stream/

- Iperf3 [NET]
  Iperf is a standard network benchmarking tool to measure the maximum achievable throughput of an IP network, using both TCP and UDP. We measure both RX and TX performance for both UDP and TCP.
  https://iperf.fr/

- Ioping [BLK]
  Ioping is a simple disk IO measurement benchmark. For this benchmark we measure both IOPS and bandwidth for both read and write operations. To measure bandwidth we use big request sizes, whereas to measure IOPS we use the smallest possible request size.
  https://github.com/koct9i/ioping

Power consumption

Power consumption is an important metric when considering single board computers that should run in constrained environments. Therefore we also measured power the consumption of the different boards both idle, and while executing the different benchmarks.

To measure the power consumption we used a Baylibre ACME.

http://baylibre.com/acme/

3.2 Hypervisor level improvements

In this section we describe some of the performance improvements to the hypervisor platforms to align better with the underlying hardware.

3.2.1 MicroVisor performance improvements

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 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.

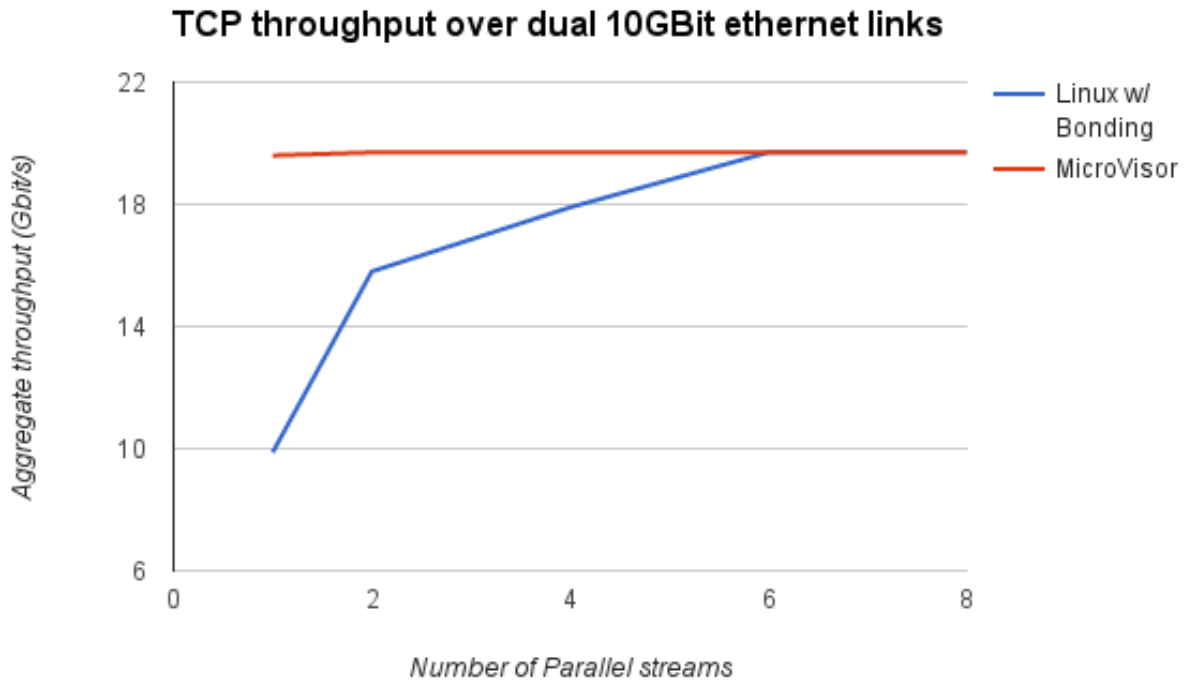

Figure 1: TCP throughput comparison of MicroVisor and Linux native with bonding across dual 10Gb Ethernet links

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 2. 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.

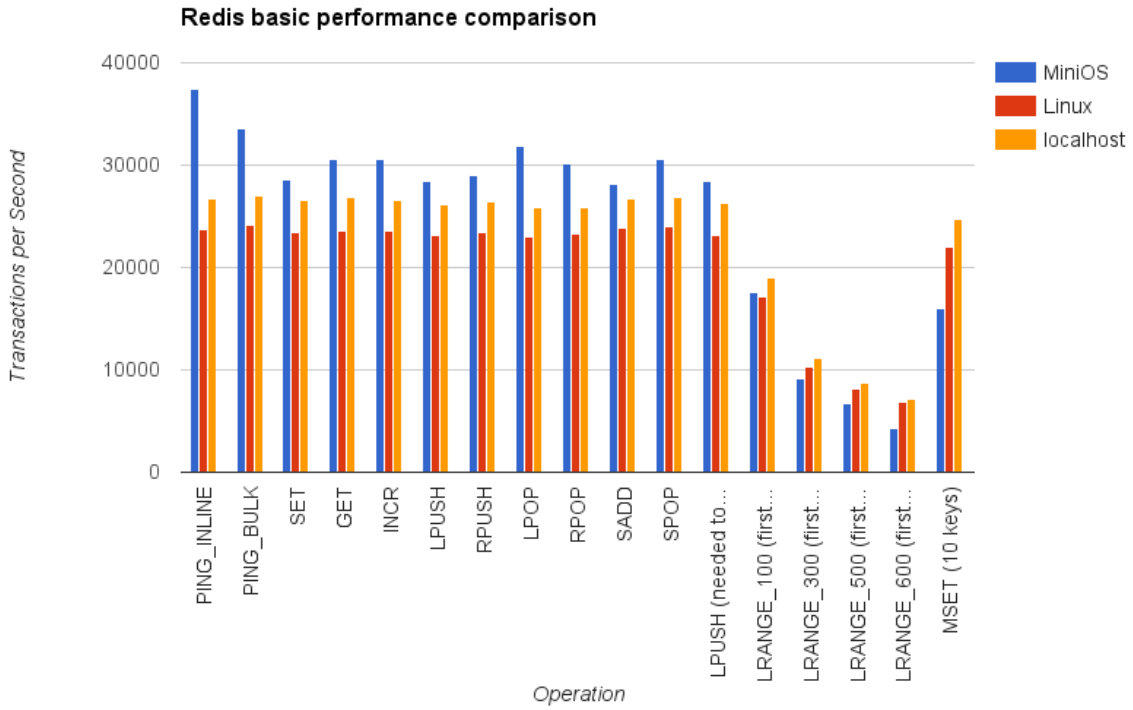

Figure 2: IO performance measured by Transactions per Second for Redis for MiniOS / Linux / co-located

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

As has been shown in Figure 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 3. The throughput for the InterNode case is fairly independent of the number of streams and is close to the physical capacity of the links. For clients that run on the same machine the performance scales as the number of streams increases.

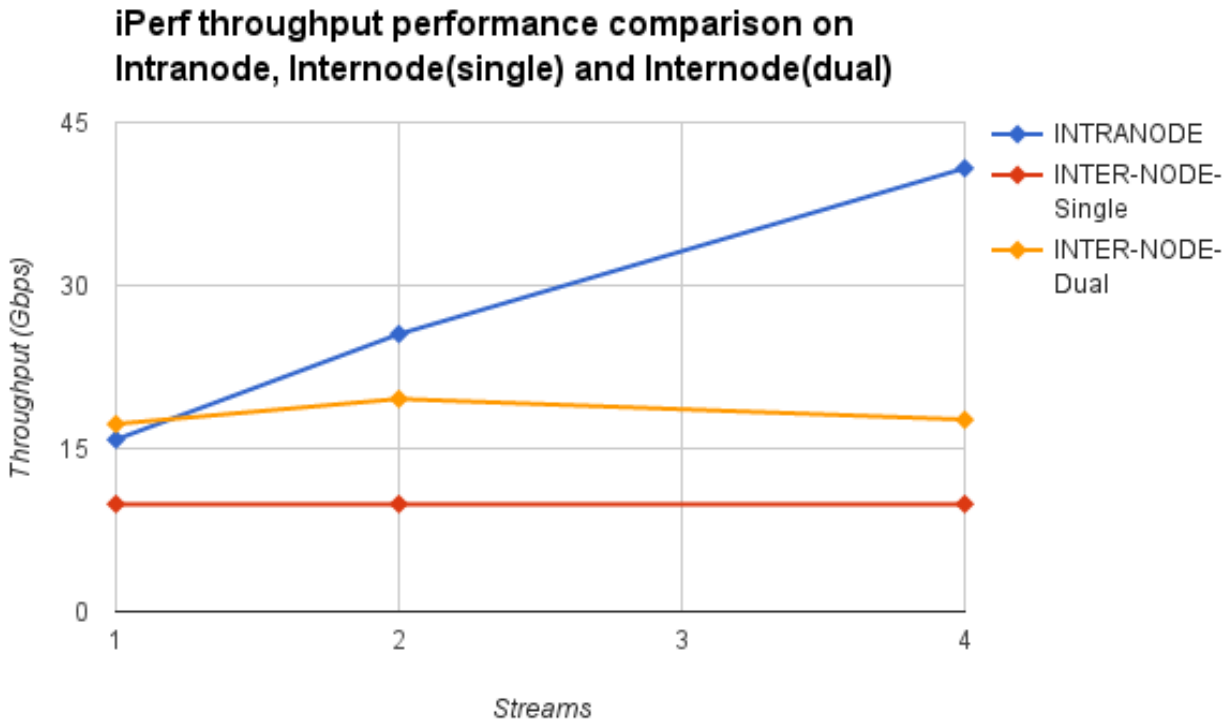

Figure 3: Network throughput measured via iPerf against number of streams for different MicroVisor configurations

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 4.

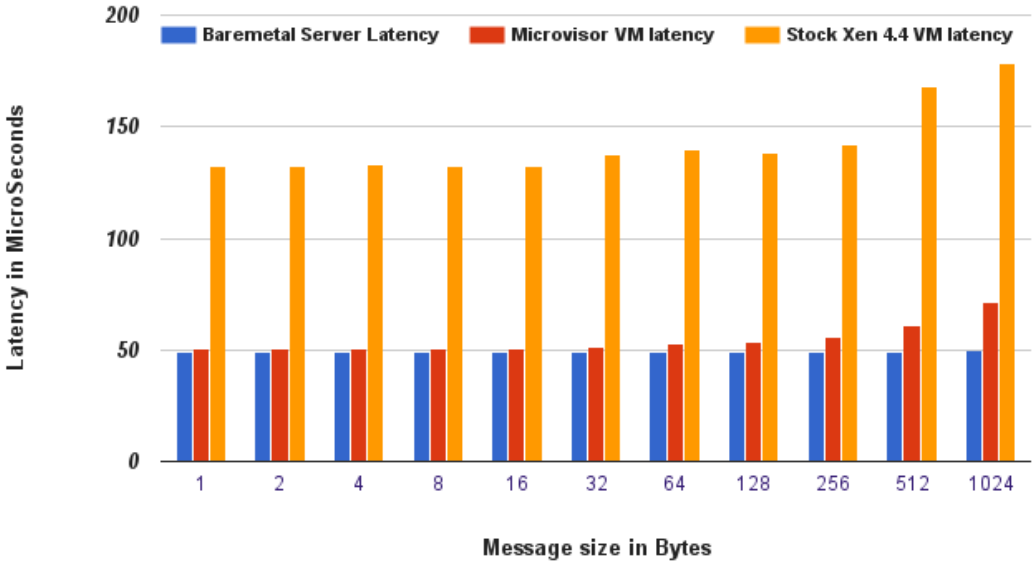

Figure 4: Communication latency derived from NPTcp vs packet size on Baremetal, MicroVisor and Xen 4.4

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 5.

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.

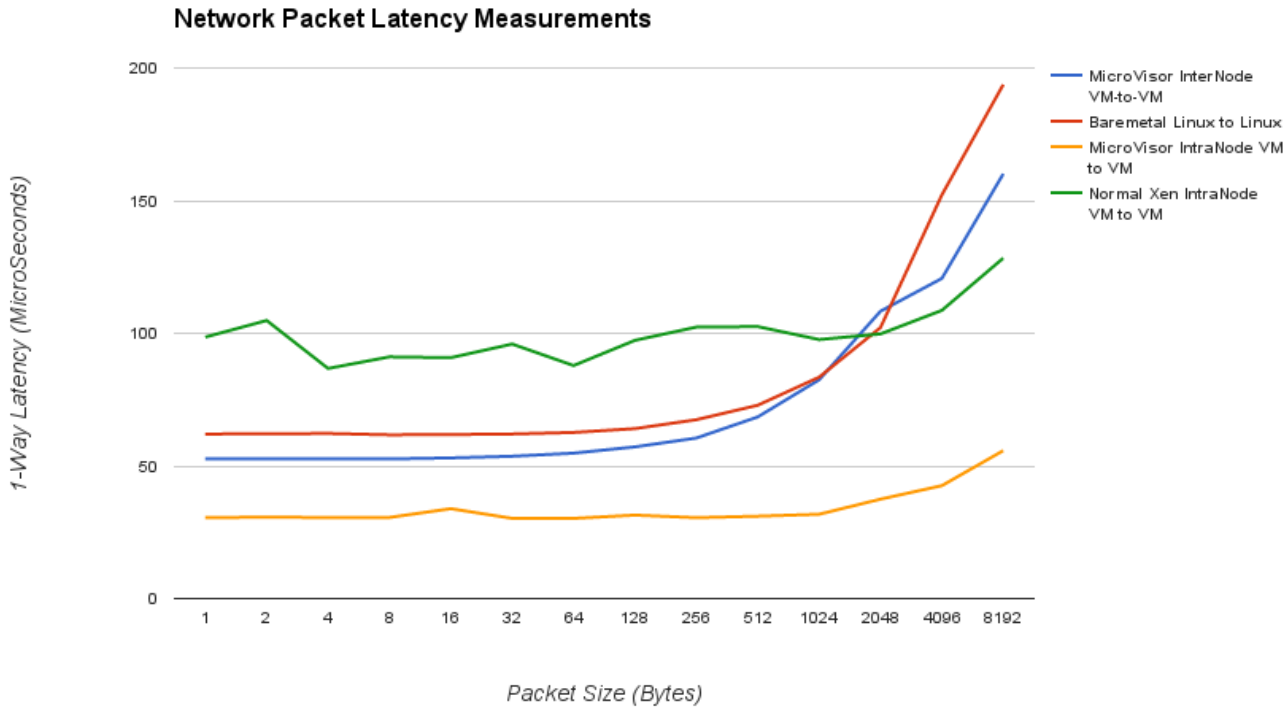

Figure 5: Network packet latency derived from NPtcp against packet size for different virtualisation configurations

In Figure 6 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.

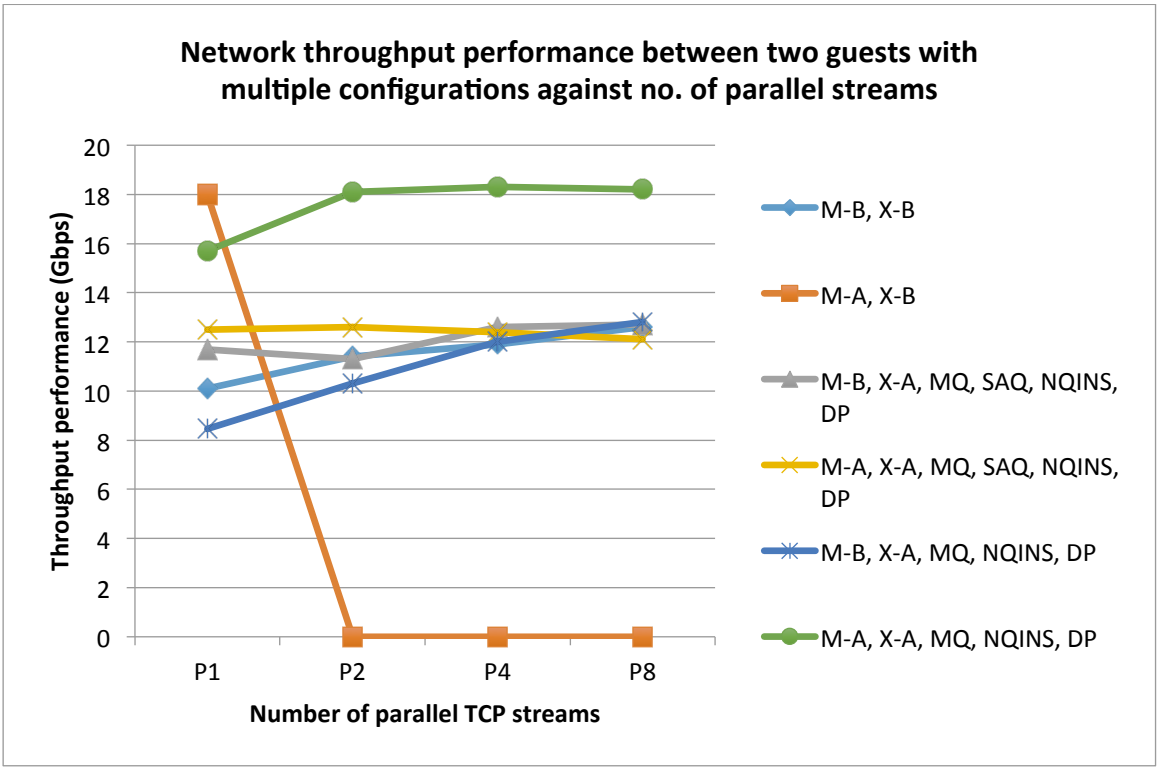

Figure 6: Network throughput performance when using different configuration options against no. of parallel streams

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

Figure 7, 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 virtual machine. The performance overhead of the MicroVisor relative to baremetal is much lower than that of the VM.

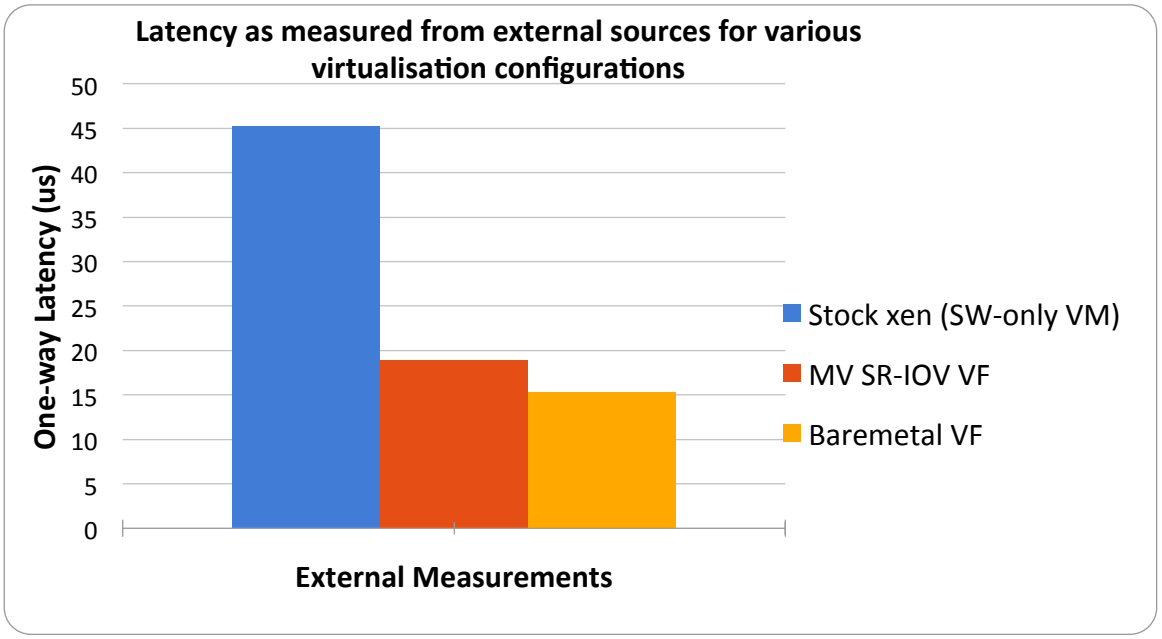

Figure 7: Latency as measured for various virtualisation configurations

Figure 8 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.

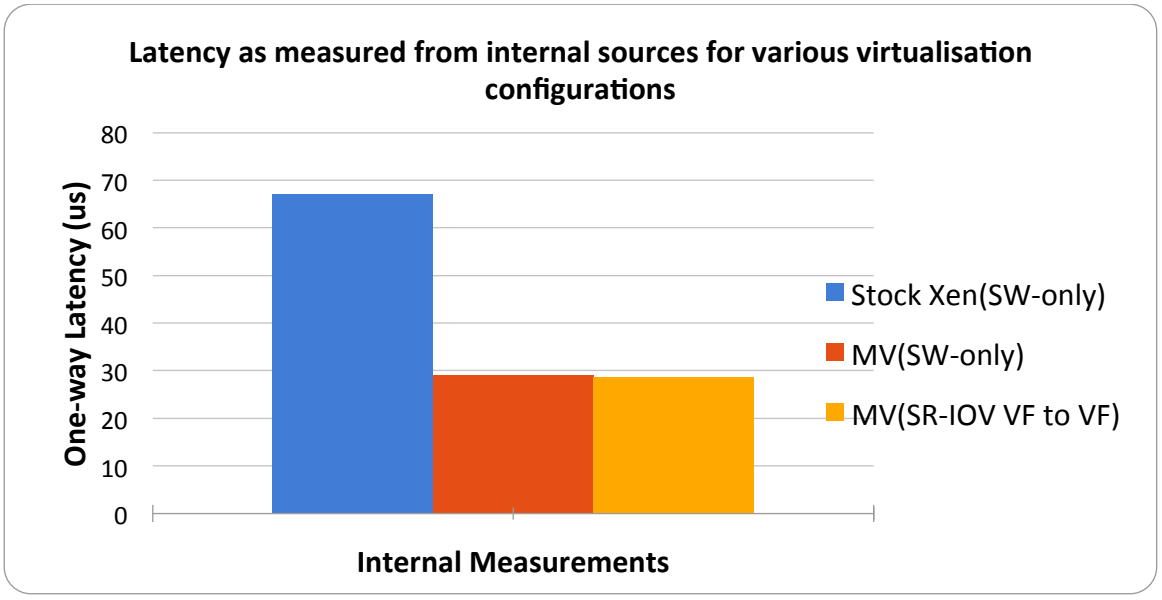

Figure 8: Latency as measured from internal sources for various virtualisation configurations

Figure 9 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.

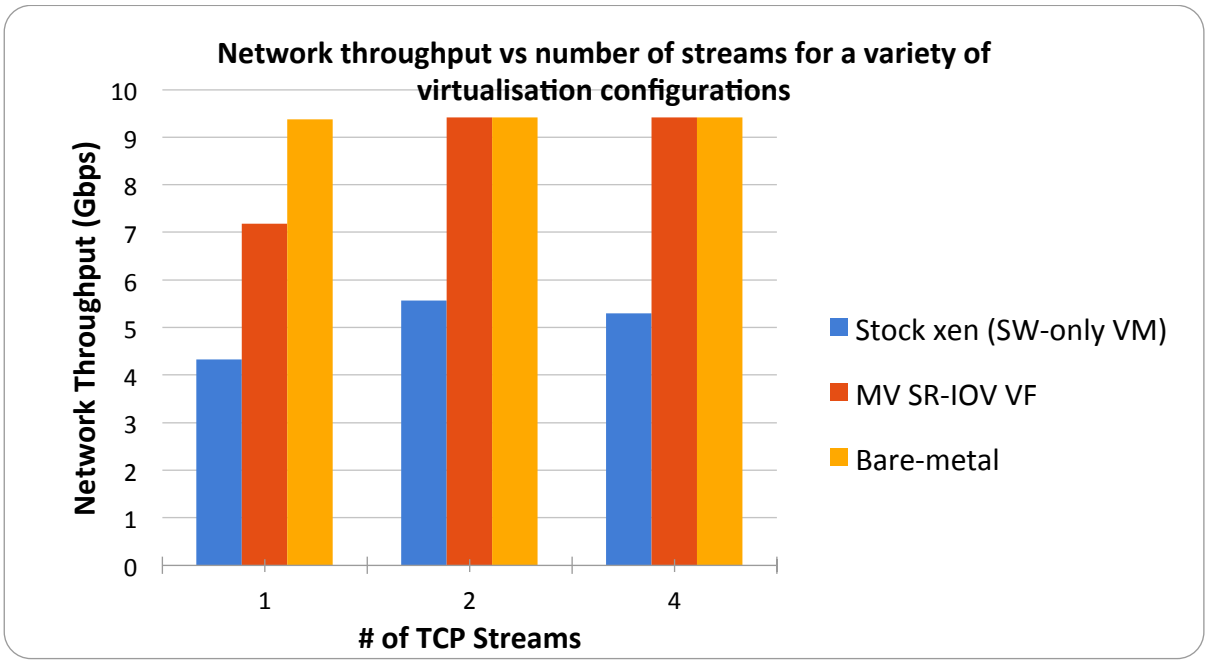

Figure 9: Network throughput vs number of streams for a variety of virtualisation configurations

3.3 Performance evaluation and tuning of Virtual Infrastructure Managers

The proposed general model of the VM instantiation process is shown in Figure 10. 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 10) 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.

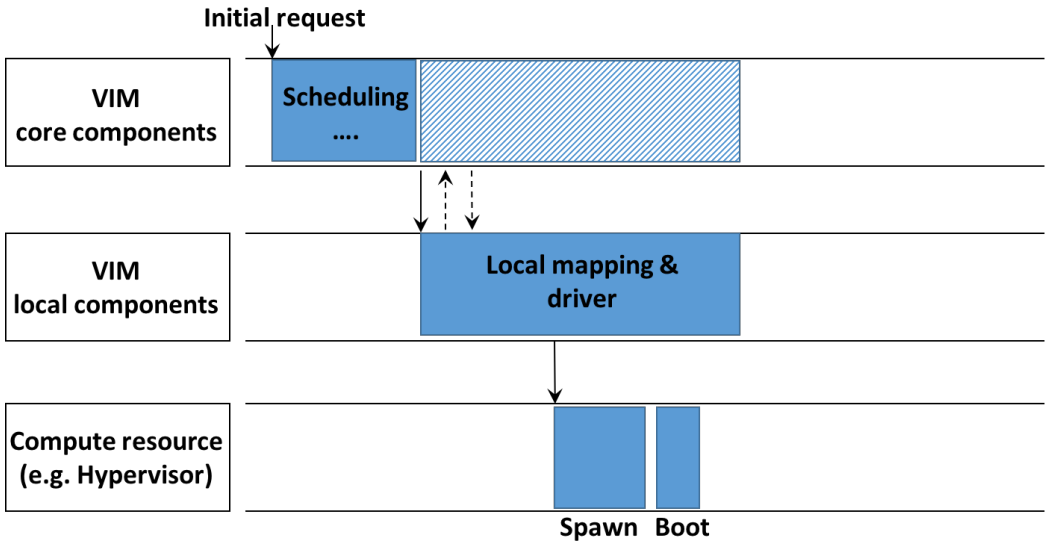

Figure 10: VIM instantiation general model

Figure 11 shows how the Nova and Nomad components can be mapped on the proposed reference model. In the next subsections, we provide a detailed analysis of the operations of the two VIMs. As for the Compute resource/hypervisor, in this work we focus on Xen [2], an open-source hypervisor commonly used in production clouds, as the resource manager. Xen can be configured to use differents toolstacks (tools to manage guests 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.

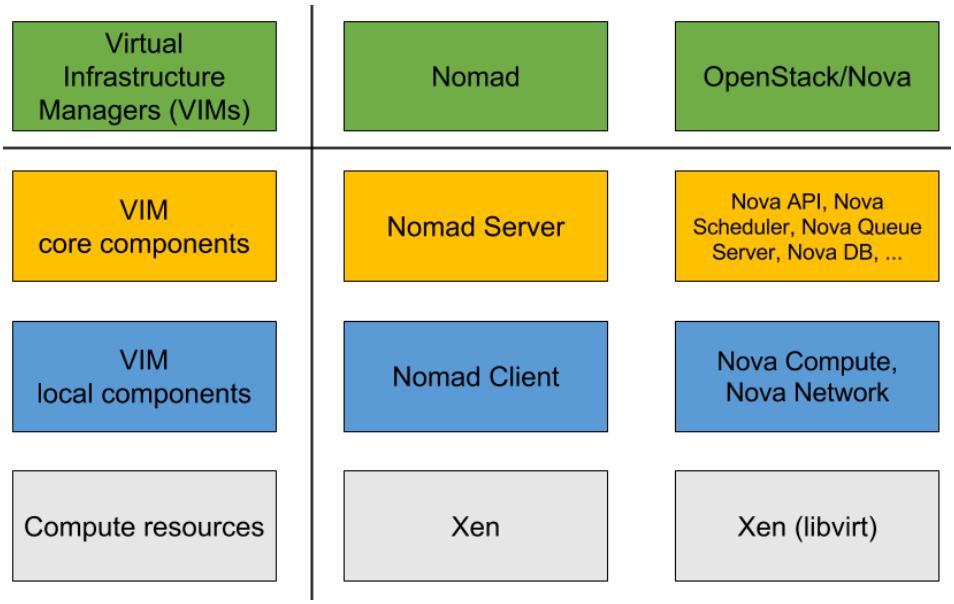

Figure 11: Mapping the reference model onto the considered orchestrators

3.3.1 Modelling OpenStack Nova

Figure 12 shows the scheduling and instantiation process for OpenStack Nova. 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 12, 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).

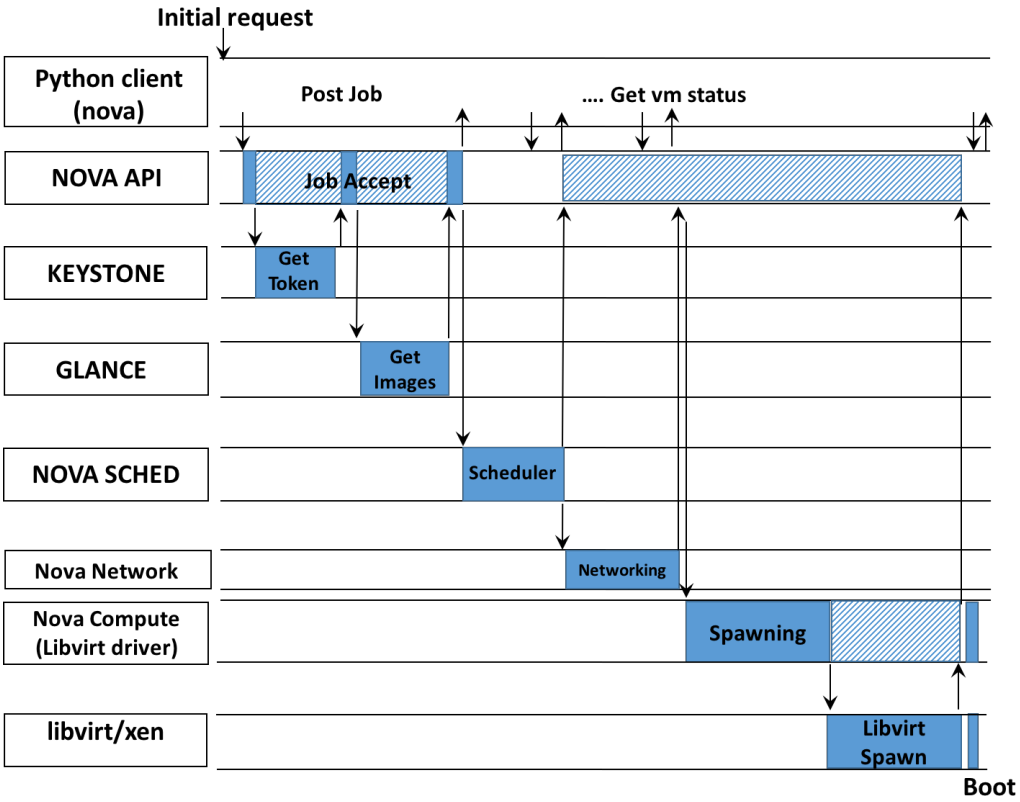

Figure 12: VIM instantiation model for OpenStack Nova

3.3.2 Modelling Nomad

The scheduling and instantiation process for Nomad is shown in Figure 13. 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 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.

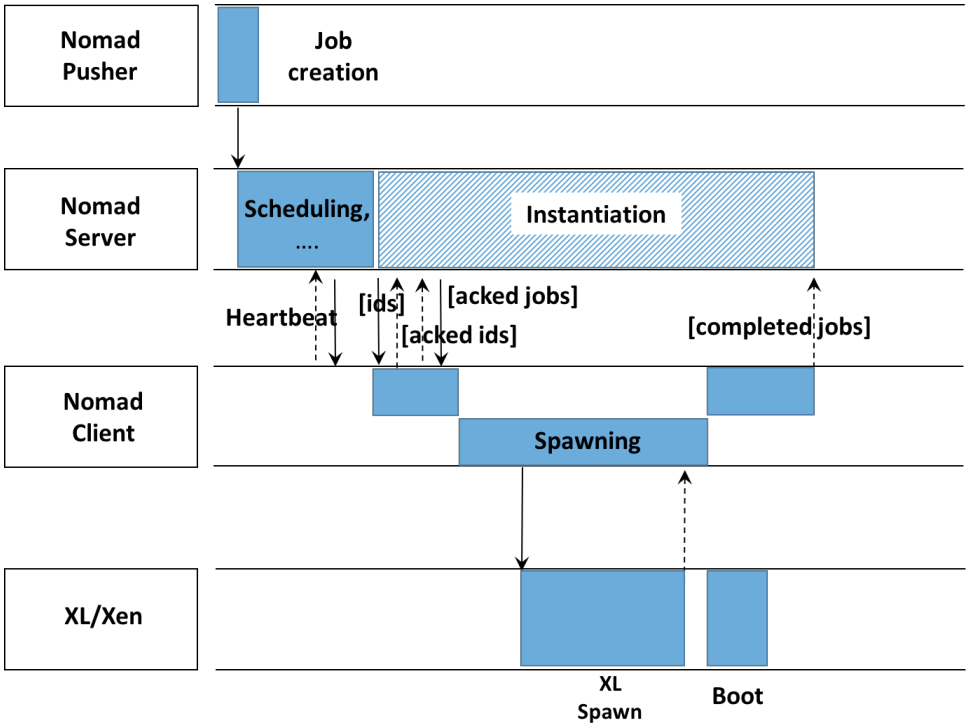

Figure 13: VIM instantiation model for Nomad

3.3.3 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 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 [3] library. The VIM Message Analyzer (available at [4]) 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 and Nova Compute nodes. We have generated the workload for OpenStack using Rally [5], a well known benchmarking tool. For the generation of the the Nomad workload, instead, we have developed the Nomad Pusher tool. It is an utility written in the GO language, which can be employed to programmatically submit jobs to the Nomad Server.

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 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.

3.3.3.1 OpenStack Nova Experimental results

For what concerns OpenStack 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 12, we report in Figure 14 the measurements of the instantiation process in OpenStack, 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. Analyzing 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 compo-nents account for around 0.5 seconds. In Figure 15 (topmost horizontal bar), we show the details of the spawning phase which is split in tree 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.

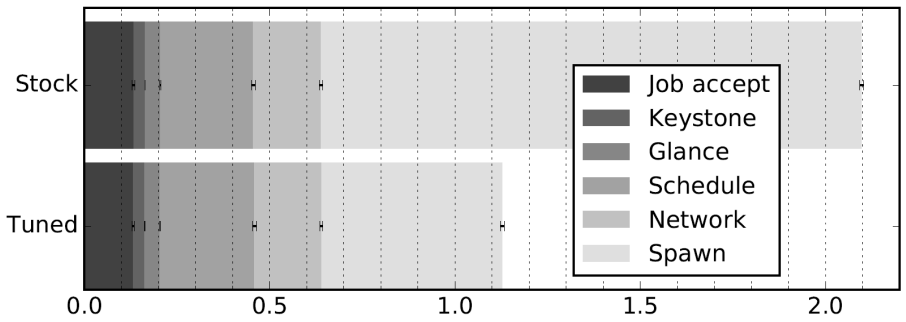

Figure 14: ClickOS (micro-NFV) instantiation time breakdown on OpenStack

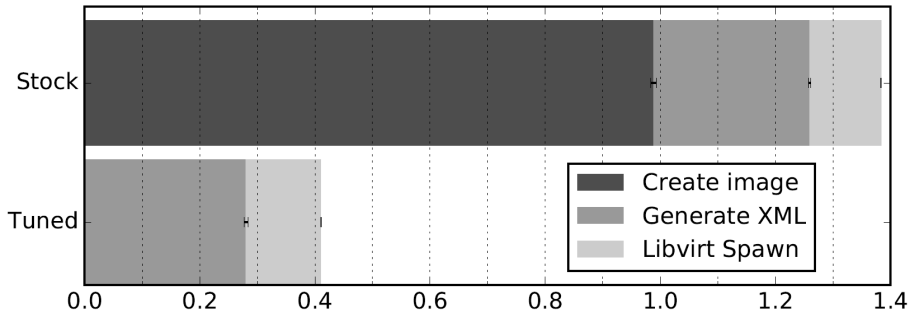

Figure 15: ClickOS spawning time breakdown on Nova-compute

3.3.3.2 Nomad Experimental results

According to the Nomad architecture, the minimal deploy-ment 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 16 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 17. 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 16. 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 16.

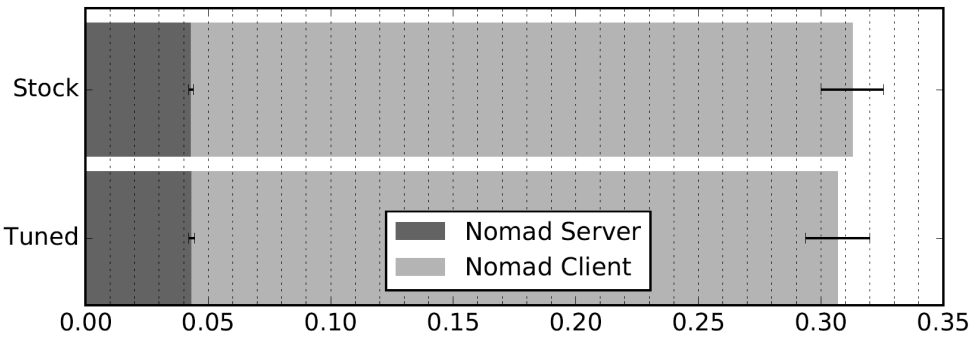

Figure 16: ClickOS (micro-NFV) instantiation time breakdown on Nomad

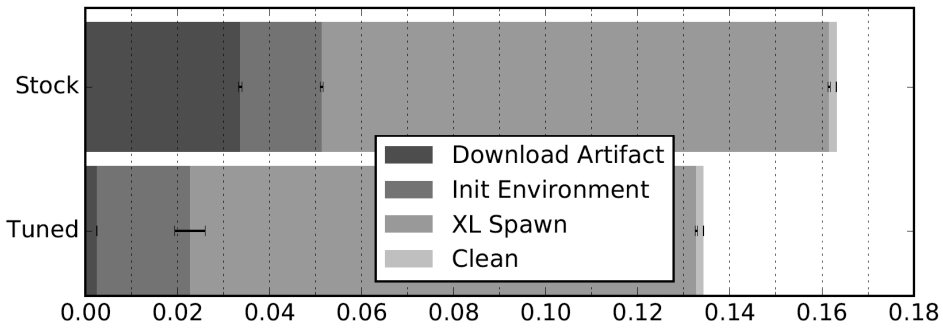

Figure 17: ClickOS spawning time breakdown on Nomad

3.4 Cloud RAN and software network functions

The final deliverable will include a report on the CRAN optimisation efforts.

4 Results

Currently there are no results to describe in I5.2

5 Conclusion

The final conclusions will be written at the same time as the complete introduction, due for D5.2 delivery date in June 2017.

6 References

[1] J. Martins et al., “Clickos and the art of network function virtualization,”in Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. USENIX Association, 2014, pp. 459–473

[2] “Xen project.” [Online]. Available: http://www.xenproject.org

[3] “Scapy.” [Online]. Available: http://www.secdev.org/projects/scapy/

[4] “VIM tuning and evaluation tools.” [Online]. Available: https: //github.com/netgroup/vim-tuning-and-eval-tools

[5] “Openstack rally.” [Online]. Available: https://wiki.openstack.org/wiki/ Rally