Performance analysis of different virtualization architectures using OpenStack

Charalampos Gavriil Kominos

Institutionen för informationsteknologi

Department of Information Technology
Abstract

Performance analysis of different virtualization architectures using OpenStack

Charalampos Gavrili Kominos

Cloud computing is a modern model for having on demand access to a pool of configurable resources like CPU, storage etc. Despite its relative youth however, it has already changed the face of present-day IT. The ability to request computing power presents a whole new list of opportunities and challenges. Virtual machines, containers and bare-metal machines are the three possible computing resources which a cloud user can ask from a cloud provider.

In the context of this master thesis, we will discuss and benchmark these three different deployment methods for a private OpenStack cloud. We will compare and contrast these systems in terms of CPU, networking behavior, disk I/O and RAM performance in order to determine the performance deterioration of each subsystem. We will also try to empirically determine if private clouds based on containers and physical machines are viable alternatives to the traditional VM based scenario. To achieve these goals, a number of software suites have been selected to act as benchmarks with the aim of stressing their respective subsystem. The output of these benchmarks is collected and the results are compared against each other. Finally, the different types of overhead which take place between these three types are discussed.
Acknowledgements

• First and foremost I would like to thank my supervisor at Ericsson Research, Nicolas Seyvet, for this fantastic opportunity to work with him and develop this thesis work.

• I would also like to thank everybody affiliated with Uppsala University and Ericsson Research who directly or indirectly contributed in my decision to choose cloud computing as my field. These include: Kostantinos Vandikas, The MoNad team, Salman Toor, Edith Ngai et al.
# Contents

## Acknowledgements

### 1 Introduction

1.1 Overview of Cloud Computing ............................................. 1
1.2 Motivation and Objectives .............................................. 2
1.3 Related Work ............................................................. 3
1.4 Structure of the Report .................................................. 4
1.5 Statement of Originality .................................................. 4

### 2 Virtualization

2.1 The need for virtualization .............................................. 5
2.2 Hypervisor-based virtualization ........................................ 5
    2.2.1 KVM ................................................................. 6
    2.2.2 QEMU ............................................................... 7
    2.2.3 Libvirt ............................................................. 7
2.3 Container-based virtualization ......................................... 7
    2.3.1 Docker ............................................................. 8

### 3 OpenStack

3.1 Overview ................................................................. 9
3.2 OpenStack Architecture .................................................. 10
3.3 FUEL ................................................................. 12
    3.3.1 Rapid Deployment Tools ...................................... 12
3.3.2 Roles in FUEL ......................................................... 13
3.3.3 Networks in FUEL ............................................... 13
3.4 OpenStack networking using Neutron ........................ 15
  3.4.1 Compute node networking .................................... 16
  3.4.2 Controller node networking .................................. 17
3.5 Containers in OpenStack ......................................... 17
3.6 Bare metal in OpenStack ......................................... 18

4 System Description .................................................. 19
  4.1 Hardware .......................................................... 19
  4.2 Software suites .................................................... 21
    4.2.1 Benchmarking Tools ......................................... 21
    4.2.2 Software versions ............................................ 22
    4.2.3 Test methodology ............................................ 22

5 Results ................................................................. 23
  5.1 CPU ................................................................. 23
    5.1.1 CPU Power .................................................... 23
    5.1.2 CPU contention .............................................. 25
  5.2 Networking ........................................................ 26
    5.2.1 Throughput .................................................. 26
    5.2.2 Latency ....................................................... 27
  5.3 Memory ............................................................. 29
  5.4 Disk I/O ............................................................ 31
    5.4.1 SysBench ..................................................... 31
    5.4.2 Binary Copy .................................................. 32
  5.5 Boot-up time ...................................................... 33
6 Conclusion 34

6.1 Summary ......................................................... 34
6.2 Limitations in the current system ............................. 34
6.3 Future work ....................................................... 35

A Acronyms and Abbreviations 37

B Testing Configuration 39

B.1 PXZ ................................................................. 39
B.2 Nuttcp .............................................................. 39
B.3 netperf .............................................................. 40
B.4 Memory ............................................................. 40
B.5 Sysbench ........................................................... 40
B.6 Binary Copy ....................................................... 40

Bibliography 40
# List of Tables

4.1 Network configuration ................................................. 20
4.2 Tools Used ................................................................. 22

5.1 1 vCPU performance .................................................. 24
5.2 2 vCPU performance .................................................. 24
5.3 4 vCPU performance .................................................. 24
5.4 8 vCPU performance .................................................. 24
5.5 CPU Contention ....................................................... 26
5.6 TCP & UDP maximum performance ................................. 27
5.7 TCP latency performance ........................................... 28
5.8 UDP latency performance ........................................... 28
5.9 UDP latency performance ........................................... 29
5.10 I/O requests .............................................................. 32
5.11 Binary copy performance ........................................... 33
5.12 Boot-up times ............................................................ 33

A.1 Acronyms used throughout this report ............................. 38
# List of Figures

2.1 Hypervisor illustration[51] .................................................. 6  
2.2 Containers illustration[46] ............................................. 7  
2.3 Docker illustration[47] ..................................................... 8  
3.1 OpenStack overview[48] .................................................. 9  
3.2 OpenStack Architecture[49] .......................................... 12  
3.3 Neutron Compute Node[50] ............................................. 16  
3.4 Neutron Controller[52] .................................................. 17  
4.1 Server Rack Diagram ................................................... 19  
4.2 VLAN Network overview ............................................. 20  
4.3 Ironic Network .......................................................... 20  
5.1 CPU benchmarking for different flavors .............................. 23  
5.2 CPU contention illustration .......................................... 25  
5.3 Network Bandwidth .................................................... 26  
5.4 Network Latency ......................................................... 28  
5.5 Random Read 64bits ................................................... 29  
5.6 Random Write 64bits ................................................... 30  
5.7 Random Read example ................................................ 30  
5.8 File Test I/O ............................................................ 31  
5.9 Binary Copy ............................................................. 32
Chapter 1

Introduction

1.1 Overview of Cloud Computing

Cloud computing is a promising field within computer science. According to the latest research, cloud computing is already a multi-billion dollar industry and it is expected to grow further in the coming decades as embedded systems and the internet of things (IOT) are integrated into the modern society.

As reported by The National Institute of Standards and Technology (NIST):
Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. [1].

It is worth the effort to analyze the four (4) different deployment models as specified by NIST. The first model is the private cloud model where the cloud infrastructure is provisioned for exclusive use by a single organization. There are a number of advantages in such a scenario. Having the systems on-premise, allows greater control to enterprises since they have the ability to configure and manage everything according to the specific needs. Data locality is also an added advantage for many organizations. Having the data on-premise is often a must-have for organizations due to privacy and security concerns. The second model described is the public cloud where the cloud infrastructure is provisioned for open use by the general public. They are mostly utilized for private individuals who are less likely to need the level of infrastructure and security offered by private clouds. They also have number of advantages like greater scalability and service reliability (SLA) since the number of underlying machines is much higher than that of a private cloud. Another advantage is that these infrastructures have usually no geographical boundaries which means that they are always available wherever clients are located. Further down the list we can find community clouds where the cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns. This is combination of the previously described models and can lead to greater
collaborations between organizations with shared interests and policies while it also enables some cost-cutting techniques. The final option is the Hybrid cloud which is also a combination of public and private models. Here a single organization utilizes resources from both public and private clouds for a single purpose. Scalability, security and flexibility are only some of the possible advantages of a hybrid cloud since it combines advantages from all the previous models.

In every cloud computing platform, there are three fundamentally different types of computing resources that a user can ask for: A virtual machine (VM), a container, or a bare-metal machine. These three different types of resources define three types of deployments:

1. Hypervisor based: In this model, hypervisors are deployed which manage VMs.
2. Container based: Where a container engine is deployed and containers are spawned.
3. Bare-metal based: In this model, full physical machines can be requested.

It is evident that in order to fully grasp the capabilities of any cloud computing system, the intricacies of the aforementioned deployments models must be studied in detail.

As however institutions look into these complex systems some challenging questions arise.

- Should an organization invest in public or private infrastructure?
- Should an organization invest in the traditional VM based scenario or should it seek other solutions?
- How difficult will a migration be from current infrastructure to a cloud based one?
- How will the current applications be affected from such a migration?

This presents organizations with a dilemma because the operation of a cloud-system is a very challenging and technically intensive task. As a result organizations are sometimes unwilling to commit resources without clear understanding of the advantages and disadvantages in each solution beforehand. As a natural response to these difficult questions, benchmarking is a first step in the quest to evaluate a cloud-based setup. By using different benchmarking techniques and suites, one can get insight, on how a computer system will behave within each of the aforementioned deployment models.

### 1.2 Motivation and Objectives

Cloud computing came with a big promise. The on-demand access to only the computing resources required and the associated ease-of-deployment and cost-deduction that comes with it. Organizations however are sometimes unaware of the performance implications that the adoption of a cloud model will have in their infrastructure and their applications. In the context of this thesis, we will try to address the previous limitation for an OpenStack cloud.

In OpenStack each deployment model mentioned before is associated with different subprojects.
1. An OpenStack Hypervisor based cloud leverages the KVM hypervisor to manage the life-
cycle of VMs and the nova-libvirt driver to manage the communication between Open-
Stack and KVM.

2. An OpenStack Container based cloud uses the docker engine to manage the contain-
ers and leverages the open-source nova-docker driver to manage communication between
OpenStack and docker.

3. An OpenStack bare-metal cloud is associated with the Ironic project.

Our challenge is quantify the performance deterioration which takes place in each cloud-based
computing resource by thorough examination of the previous components. We will study each
system in detail and try to explore the performance variations for the following resources:
CPU, disk I/O, memory and networking. This will be accomplished through the use of several
open-source bench-marking tools. We will also try to empirically determine if private clouds
based on containers and/or physical machines are viable alternatives to the traditional VM-
based cloud scenario.

1.3 Related Work

As stated in [37] continuous re-evaluation of cloud solutions through benchmarking is both
inevitable and necessary. Modern benchmarking systems usually automate that procedure by
using different approaches and a plethora of software suites. In [31] the authors worked with
NGINX and video streaming. In [36] micro-benchmarking suites were used which stressed each
computer subsystem differently, like LINPACK and IOZONE while in [37] the authors build
their own engine running a number of pre-configured suites and presenting the results in a
user friendly way. This thesis uses the knowledge and expertise from those experiments but
the automation suites are never used. The reason for this, is that it would not present the
opportunity to really go for system-level understanding of the infrastructure and discovering
the real reason of the bottlenecks. We also find similar work in Felter et. al [53] where the
authors perform a comprehensive comparison between VMs and containers. The main difference
between this work and theirs is that the emphasis is placed on OpenStack and on different
kinds of hosts that can be provisioned as opposed to performing a straight-cut comparison of
hypervisors vs containers. Moreover, in the context of OpenStack there is similar work done
by Boden [45] where OpenStack Rally is used for the benchmarks.
Virtualization on the other hand has been studied more, since it exists as a subject since 1970
and as such, literature regarding both the performance variations of different hypervisors [35]
and the performance variations between containers and hypervisors [34] can be found.
Our contributions are mostly against [53] and [45]. We attempt to improve by using the most
reliable, open-source, micro-benchmarking suites found in [31],[36] and [37] to produce a more
up to date benchmark which focuses on cloud computing resources. We also improve on all the
previous work by extending that comparison with Ironic which has not been studied extensively
before.
1.4 Structure of the Report

The remaining report is organized as follows: Chapters two and three present virtualization and the OpenStack platform which are the bedrock for our project. Then in chapter four we discuss the the actual setup both from software and from a hardware perspective. Finally we close off the thesis-work with chapters five and six where we discuss and evaluate the results from the benchmarks.

1.5 Statement of Originality

This is to certify that, to the best of my knowledge, the content of this document is my own work. This project has not been submitted for any degree or other purposes. I certify that the intellectual content of this project is the product of my own work and that all the assistance received in preparing this project and sources have been acknowledged.
Chapter 2

Virtualization

2.1 The need for virtualization

Before we delve into virtualization we should take a step back and discuss four facts about computer science and software development.

- Moore’s law always drives the performance of modern computers by constantly packing more and more capability into faster, smaller and newer packages[44].

- Software development principles require a ”one server, one application” policy, which means that for each new role in a software system (DB, proxy server), typically a new machine needs to be leveraged.

- The server population has been growing rapidly and a “one physical machine- one application” approach would exhaust the resources of typical datacenter quickly.

- A typical machine rarely uses 100 % of its capabilities in production.

The aforementioned points have led computer engineers to always strive to increase use of the available resources as efficiently as possible and this was on of the driving forces which created virtualization as an idea. Virtualization as a concept can be defined as the ”abstraction of some physical component into a logical object”.

2.2 Hypervisor-based virtualization

The most common form of virtualization in computer engineering is the VM. A VM is an efficient, isolated duplicate of a real machine [30]. For example, an application could require two databases and a proxy server. With these requirements, a physical machine could spawn three virtual machines and in each machine deploy different software packages along with any libraries. This allows better machine consolidation and added security since if a service becomes
unstable or compromised, the other services on the physical host are not affected.
In order to create, run and control VMs, a new software layer is required between the application
and the hardware which is called hypervisor. The hypervisor presents the guest operating
systems with a virtual operating platform and manages the execution of the guest operating
system.

There are two types of hypervisors:

1. Type one hypervisor or bare-metal: VMs lie directly on top of the hypervisor (like ESXi
   or Xen)
2. Type two hypervisor: VMs lie on top of a physical host which is running its own operating
   system. Our focus in this document will be on this category and more specifically in the
   KVM-QEMU pair.

![Hypervisor illustration](image)

Figure 2.1: Hypervisor illustration[51]

2.2.1 KVM

KVM (Kernel-based Virtual Machine) is a full virtualization solution for Linux on x86 hardware,
containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel
module which allows a Linux host to act as type two hypervisor and run a guest operating
system inside a Linux process. [20].
QEMU [6] is a generic and open source machine emulator and virtualizer. It is a user space tool whose aim is to emulate guest hardware devices and instantiate virtual machines. When used in conjunction with KVM, QEMU emulates the hardware, but the execution of the guest is done by KVM as requested by QEMU.

Libvirt [25] is software suite that provides a convenient way to manage virtual machines and other virtualization functionality, such as storage and networking. It includes a C API, a daemon (libvirtd), and a command line utility (virsh). A primary goal of libvirt is to provide a single way to manage multiple different hypervisors, such as the KVM/QEMU, Xen and others. Virtual Machines managed by virsh are created by describing the virtual machine in a XML file.

2.3 Container-based virtualization

Container-based virtualization or operating-system-level virtualization, is a lightweight virtualization approach using the host kernel to run multiple virtual environments. These virtual environments are often referred to as containers[21]. Linux-Vserver, OpenVZ and LXC are just a few of the possible systems which leverage containers.

From the outside, a process which is running inside a container is not different than a process running on the host machine. However from the inside, containers use the Linux Kernel namespaces feature, cgroups and a copy-on-write filesystem to achieve complete filesystem, process
and network stack isolation. By using the aforementioned technologies containers can achieve complete isolation. Generally this involves adding a container ID to every process and adding new access control checks to every system call. This is based on the kernel namespaces feature. This feature allows the creation of completely separate instances of previously global namespaces (PID, filesystem, network et al.) [53]. As a result a process running inside a container has no visibility or access to objects outside the container and process isolation is achieved.

2.3.1 Docker

Docker is a container management tool which simplifies the use of containers. Docker also made good use of the concept of software images. An image is a layered format using a Union file system package, which is built step-by-step using a series of instructions. Images are highly portable and easy to manipulate. They are a portable binary that programmers can use in order to share software. The following diagram is a good illustration of Docker’s architecture.

![Docker illustration](image)

Docker consists of the following parts.

1. Registry: The docker registry is where docker images are stored. The Docker Hub [38] is the largest public registry available hosted by Docker Inc.

2. Client: The part that the user interacts with in order to issue commands.

3. Daemon: The central component of docker. It receives commands from the Client in order to manage Docker’s images and containers.

Docker’s main advantages are the simple APIs to safely create, control and package containers, to create portable images and to deploy them in another host.
3.1 Overview

OpenStack is a free and open-source software platform for cloud computing, mostly used as an infrastructure-as-a-service (IaaS). It is a cloud operating system that controls large pools of compute, storage, and networking resources within a datacenter[8].

The following diagram, which is taken from the OpenStack website, gives a high level view of the different OpenStack components.[8]

![OpenStack Overview Diagram](image)

Figure 3.1: OpenStack overview[48]

Everything in OpenStack is accessible through APIs. Users can interact with the platform in the following ways[2].

- **HTTP**: Tools like curl can be leveraged to make http requests to OpenStack.
- **CLI**: Each OpenStack component comes with its own CLI reference in order to access the API.
• REST clients: Browsers like Chrome or other programming languages can be used to send REST messages to OpenStack.

• Python: All OpenStack command-line tools are implemented by using the Python SDK. The SDK implements Python bindings to the OpenStack API, which enables users to perform calls on Python objects rather than making REST calls directly.

• Horizon: A Javascript implementation of a user interface accessible through a web browser.

3.2 OpenStack Architecture

OpenStack is designed with a modular architecture. Every component in OpenStack is designed to be installed and operated without using any other component. It is therefore in the hands of the system administrator to judge which components are required and to deploy accordingly.

Figure 3.2 shows the relationships among the OpenStack services in the Kilo release. At the time of this writing (15/8/2016) figure 3.2 is the latest diagram provided by the OpenStack foundation, which describes interactions between the components. Each service has a specific role and inter-service communication is done through REST API calls. The components and their use is presented below [9][11].

• **Keystone:** The core project that provides a central directory of users mapped to the OpenStack services they can access. It also registers endpoints for OpenStack services and acts as a common authentication system.

• **Ceilometer:** The project provides metering and measuring facilities for OpenStack.

• **Horizon:** The project provides a dashboard, which is a web interface for managing the cloud.

• **Glance:** The core project provides discovery, registration, and delivery services for disk and server images.

• **Nova:** The core project provides compute services.

• **Sahara:** OpenStack project that provides a scalable data-processing stack and associated management interfaces.

• **Swift:** The core project provides object storage and is responsible for eventually consistent and redundant storage and retrieval of fixed digital content.

• **Neutron:** The core project provides networking services.

• **Cinder:** The core project provides block storage services for VMs.

• **Ironic:** The project provides bare metal, as opposed to virtual, machines.

• **Trove:** The project provides scalable and reliable Cloud Database-as-a-Service functionality for both relational and non-relational database engines.

• **Heat:** The project aims to orchestrate multiple cloud applications for OpenStack.
Also available but not listed in the diagram are:

- **Zaqar**: A project that aims to produce an OpenStack messaging service that affords a variety of distributed application patterns in an efficient, scalable and highly-available manner.

- **Manila**: The project provides a set of services for management of shared file systems in a multi-tenant cloud environment. Users can create remote file systems and mount them on their instances.

- **Designate**: The project provides scalable, on demand, self service access to authoritative DNS services, in a technology-agnostic manner.

- **Barbican**: The project produces a secret storage and generation system capable of providing key management for services wishing to enable encryption features.

- **Magnum**: The project provides containers Service.

- **Murano**: The project provides an application catalog service so that users can compose and deploy composite environments on an application abstraction level while also managing the application lifecycle.

- **Congress**: The project provides Governance-as-a-Service across any collection of cloud services in order to monitor, enforce, and audit policy over dynamic infrastructure.
3.3 FUEL

3.3.1 Rapid Deployment Tools

In order to accelerate the deployment of OpenStack clouds, a number of tools have been developed. The aim of these tools is to automate OpenStack deployment while hiding some of the complexity from the user.

The most common tools are:

- Devstack: An open-source community driven tool.[3]
- RDO: Driven by Red Hat. [4]
- PackStack: A utility that uses Puppet modules to deploy various parts of OpenStack on multiple pre-installed servers over SSH automatically.[5]
3.3. FUEL: A rapid deployment tool which is developed by Mirantis. [7]

For this thesis, the choice was made to use mirantis fuel mostly because of the active community. However any other tool would work as the results are irrespective of the tool used to deploy OpenStack.

3.3.2 Roles in FUEL

Each OpenStack tool deploys these services in a different way. Some tools will deploy an all-in-one single-node test environment while others will focus on a fully distributed environment. In a fully distributed environment multiple machines are leveraged to take part in the deployment. In the context of this thesis the focus will be on a distributed environment with the use of the FUEL project. In order to better grasp the operations in OpenStack, the concept of roles must be understood. In mirantis-fuel there are four different types of roles that a physical machine can have. [10]

- **Fuel Master:** A server with the Fuel application installed, also referred to as the Fuel server. The Fuel Master runs on its own machine, or VM and it is used to deploy and manage OpenStack environments. It assigns IP addresses to the nodes, performs PXE boot and initial configuration, and provisions the nodes according to their roles in the environment.

- **Controller:** The controller node is the server which manages all activities in an OpenStack environment. In Fuel it runs the following openstack services: Keystone, Glance, Cinder, Heat and Neutron. It also runs some extra services, namely RabbitMQ, HAProxy, MySQL, Galera, Pacemaker and Cluster, which are required for fuel to operate.

- **Compute Node:** A Compute node is a node to which the Compute role is assigned. Compute nodes are the workhorses of any installation. They are the servers on which virtual machines are created. Nova-compute controls the life cycle of these VMs. The Neutron Open Vswitch agent also runs on Compute nodes, while the other Neutron agents run only on Controller nodes. The Ceilometer Compute Agent may also run on Compute nodes.

- **Ironic Node:** The Ironic node is responsible for orchestrating, provisioning and managing the lifecycle of a physical machine according to the user’s needs.

3.3.3 Networks in FUEL

It is also equally important to understand the networking infrastructure of FUEL, which deploys the following networks.[12]

- **Public network:** Public network provides connectivity to the globally routed address space for VMs. That means that the VMs use this network to communicate with each other and to communicate with machines outside the OpenStack cloud. The IP network
range is used by Source-NAT to enable the out-going traffic from VM across the Internet. It also provides the range from which floating IP address are drawn and assigned to instances.

- **Private:** A private network is an internal Vlan seperated network so that VMs which reside in the same project can share a common address space for communication purposes.

- **Internal:** All other networks are classified as internal and are designed by fuel to separate traffic. These networks are:

  1. **Admin:** The admin, or PXE network, is used to provision a new node to OpenStack environment. It is also used to orchestrate and configure that node according to the role that the node will get.

  2. **Management:** Management network serves all other internal traffic such as database queries, AMQP messaging, and high-availability services.

  3. **Storage:** Storage network handles replication traffic from Ceph or Swift.

  4. **Baremetal:** Baremetal is the overlay network in which all the BM machines must be connected to in order to be deployed and configured on demand.

Networking is very important in distributed systems for multiple reasons like security and performance.

In Openstack, traffic isolation can be achieved in three ways:

1. **VLAN tagging:** In each data-link-layer frame, a 12-bit field specifying the VLAN ID is attached. Devices with same VLAN ID are members of same broadcast domain and receive all traffic. All other traffic that is not part of the same broadcast domain is filtered out from the switches in the network. It employs up to 4096 VLAN tags.

2. **VxLAN:** VxLAN is variation of the VLAN tagging protocol. It extends address space by adding a 24-bit segment ID and increasing the number of available IDs to 16 million. As before only machines within the same logical network can communicate with each other.

3. **GRE:** Generic routing encapsulation (GRE) encapsulates packets into IP packets and redirects them to an intermediate host, where they are de-encapsulated and routed to their final destination. Because the route to the intermediate host appears to the inner datagrams as one hop, switches can operate as if they have a virtual point-to-point connection configuration.

For this project we work with VLAN tag isolated networks.
Neutron is the OpenStack project which provides network connectivity as a service between interface devices (e.g., virtual NICs) and provides an API which allows users to define network connectivity and addressing in the cloud. It also provides an API to configure and manage a variety of network services ranging from L3 forwarding and NAT to load balancing, firewalls a, and IPsec VPN.

Neutron setup consists of multiple services and agents running on one or multiple nodes. Each of those services provides some of the networking or API services. The most important services are:

- **Neutron-server**: Provides API endpoints and serves as a single point of access to the database. It usually runs on nodes called Controllers.

- **Layer2 agent**: Can utilize Open vSwitch, Linuxbridge or other vendor-specific technology to provide network segmentation and isolation for project networks. The L2 agent runs on every node where it is deemed responsible for wiring and securing virtual interfaces (usually both compute and controller nodes).

- **Layer3 agent**: Runs on Controller nodes and provides routing services and optionally Firewall as a service FWaaS or Virtual Private Networking as a service (VPNaaS).
3.4.1 Compute node networking

On the compute node, a packet which comes in from the NIC will take the following path:

br-eth → phy-br-eth (virtual ethernet pair) → br-int (OVS integration bridge) → qvbXXX (virtual ethernet pair) → into the qbrXXX linux bridge which OpenStack uses to control access to the machines and finally in the tap port (vnet). From there, the VM is connected to the port and receives the traffic.

When working with nova docker, a new networking namespace is created on the host machine for each container. This new namespace is then connected to the tap port. As a concept it is similar to the ”host” capability of a container. Therefore we can still use the existing networking fiber as provided by neutron. The only downside is that the container has very limited control over its networking capabilities.

Finally, when working with Ironic the previous diagram does not apply since a physical machine is just connected to ETHx. Therefore currently there is no way to isolate the tenants in Baremetal Network.
3.4.2 Controller node networking

The above diagram represents the networking stack on the controller. For every private network, the controller has a DHCP server and a virtual router in a separate network namespace. Irrespective of the system tested (Ironic, Nova-Docker, VM) the controller remains the same. Therefore during implementation, the only thing which was changing according to the needs of the experiment was the compute node.

3.5 Containers in OpenStack

There are numerous usages of containers in OpenStack. Currently the most popular are:

- **Kolla**: Its mission is to provide production-ready containers and deployment tools for operating OpenStack clouds [40].

- **Magnum**: Is an API service developed making container orchestration engines such as Docker Swarm, Kubernetes, and Apache Mesos available as first class resources in OpenStack [39].

- **Murano**: Is an application catalog which includes a Kubernetes application for deploying containers [41].
• **Nova-Docker**: Enables users to spawn containers inside compute hosts instead of VMs [28].

For our work we focus on the Nova-Docker driver. A standard compute node leverages the nova-libvirt driver to spawn VMs on the KVM hypervisor. The task was to modify nova on the compute node so that it could use the nova-docker driver instead of the nova-libvirt driver. The driver integrates with neutron with minimal configuration. As a result, the networking diagrams 3.3 and 3.4 still apply. In addition the nova-docker driver can be used as the default driver in a compute node, enabling clusters where VMs and containers coexist. For our tests we use the standard Ubuntu container as it is found in docker hub.

### 3.6 Bare metal in OpenStack

By bare metal, we mean a physical machine which has no hypervisor running. In OpenStack the ability to control bare metal machines is achieved through the Ironic Project [29]. Ironic integrates out of the box with FUEL 8.0 and little configuration was required. For our tests we had to create our own Ubuntu images to be used with ironic which are approximate 1.6 GB.
Chapter 4

System Description

4.1 Hardware

Diagram 4.1 represents the system on which we take our measurements. The physical servers which were used were Hewlett Packard G6 servers. They include 2 x Intel(R) Xeon(R) CPU E5540 @ 2.53GHz processors with 72 GB of RAM and two 160 GB hard disks in RAID0 configuration.

These machines are connected with the aid of two NICs to a 1000/1000 network with the aid of Extreme Networks X450a-48t Switch. Compute node number two was used for all the experiments. Compute node number one was used to test that all the deployment models in node number two can communicate with VMs or containers hosted in other nodes.

Diagrams 4.2 and table 4.1 show the networking configuration.
In order to use Ironic the network configuration has to be expanded. The extra network, which is visible in the diagram, is another overlay network which allows the Ironic conductor to power on/off the machines.
The ironic conductor achieves that through the Integrated Lights Out (ILO) interface. ILO is HP’s version of the Lights Out Management (LOM) protocol. A LOM allows a system administrator to monitor and manage servers and other network-attached equipment by remote control regardless of whether the machine is powered on, or if an operating system is installed or functional.

4.2 Software suites

4.2.1 Benchmarking Tools

CPU, memory, disk I/O and network are the fundamental computing resources that comprise a computer system. Each workload stresses these resources in a different way.

In order to get valuable insight on how each deployment model behaves, open-source benchmarks have been selected to stress in each resource.

- **CPU:** In order to stress the CPU component, the PXZ program has been selected.[14] This tool, which is widely available in Ubuntu and Fedora, is an implementation of the classic Lempel-Ziv-Markov algorithm.[13] PXZ is a parallel lossless compression tool which can be easily configured to run in any number of cores, therefore making it easy to run in machines with different CPU capabilities. As input for PXZ a one (1) GB wikipedia datadump has been selected.[15] The file is fed into the PXZ algorithm while varying the number of cores. The wall time that PXZ takes to compress the file is then recorded.

- **Network:** As far as the network is concerned, network throughput and network latency were tested.
  - Nuttcp [17] is used to measure the network throughput. The tool works in a client-server model. The system to be tested acts as client and a physical machine external to the cloud acts as a server. Traffic is generated between client and server. The throughput is a variable in Nuttcp, so tests are made between 650 and 1100 Mbps in order to observe the performance at different rates. We measure both inbound and outbound throughput.
  - Netperf [16] is used to measure the network latency. It also works in a client-server model although in this case the machine to be tested acts as a server. The client is then used to generate a packet of fixed length (200 bytes). When that packet reaches the server, the server sends back a response of fixed length (200 bytes). Therefore only one packet is in flight at any point in time. We measure how many packets are send within a fixed period of time thus allowing the use of a simple model to make a good approximation of the latency in the system.

- **Memory:** In the tested system four memory levels are available (L1 cache, L2 cache, L3 cache, RAM). In order to test all these resources the open-source bandwidth[18] tool was selected. In this tool memory chunks of different sizes are copied into memory. The
tools starts from 256 bit and continues until the chunks are 512 MB in size. For different chunk sizes a different memory level is utilized.

- **Disk I/O:** In order to test system I/O two different tools were used.
  
  
  - Binary copy (dd) utility of linux.

  The tools operate, by copying and reading from and to big chunks of data. It is important for the I/O benchmarking to limit RAM involvement. A big RAM would mean that caching would be a predominant factor and therefore would cloud the real results. A way to bypass this problem is to use files and data which are larger than the amount of system memory. So for these experiments the machines are set to 8 GB of RAM and the files that they are working with have a size of 16 GB (twice the size of the RAM).

### 4.2.2 Software versions

<table>
<thead>
<tr>
<th>System</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Openstack Libery</td>
<td>Fuel 8.0</td>
</tr>
<tr>
<td>ParallelPXZ</td>
<td>4.999.9beta</td>
</tr>
<tr>
<td>Nuttcp</td>
<td>6.1.2</td>
</tr>
<tr>
<td>Netperf</td>
<td>2.6.0</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>1.3.1</td>
</tr>
<tr>
<td>SysBench</td>
<td>0.4.12</td>
</tr>
<tr>
<td>dd</td>
<td>8.25</td>
</tr>
</tbody>
</table>

Table 4.2: Tools Used

### 4.2.3 Test methodology

Each experiment is repeated ten times and the mean and standard deviation are recorded. The standard deviation is displayed as error bar in bar diagrams. If the error bar is not visible it should be assumed that the deviation was less than 1%.

Furthermore it is important that the CPUs which are visible during virtualization, are the same as the CPUs which exist on the host machine. This is accomplished through editing the XML file of the VM and using the \texttt{cpu <cpu mode='host-passthrough'>} configuration [43].

Detailed list of commands and configuration can be found in Appendix B.
Chapter 5

Results

5.1 CPU

5.1.1 CPU Power

For the CPU tests, four different OpenStack flavors are created with variable number of vCPUS. A flavor describes the characteristics of a machine to be created. In our test we operate with flavor with 1, 2, 4, 8 vCPUs. In order to make useful observations PXZ’s ability configure the number of cores to be leveraged is used. For example a VM with 2 vCPUS is compared with a baremetal machine where the PXZ algorithm is forced to use only 2 cores.

The following diagram and tables present the findings.

![CPU Benchmarking Diagram](image)

Figure 5.1: CPU benchmarking for different flavors
## 1 vCPU performance

<table>
<thead>
<tr>
<th>System</th>
<th>time (s)</th>
<th>std</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ironic</td>
<td>670.8</td>
<td>6.9</td>
<td>-2 %</td>
</tr>
<tr>
<td>Host</td>
<td>683.8</td>
<td>2.8</td>
<td>0 %</td>
</tr>
<tr>
<td>Docker</td>
<td>685.2</td>
<td>3.9</td>
<td>0 %</td>
</tr>
<tr>
<td>VM</td>
<td>709.6</td>
<td>7.8</td>
<td>+3 %</td>
</tr>
</tbody>
</table>

Table 5.1: 1 vCPU performance

## 2 vCPU performance

<table>
<thead>
<tr>
<th>System</th>
<th>time (s)</th>
<th>std (s)</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ironic</td>
<td>360.4</td>
<td>4.3</td>
<td>-7 %</td>
</tr>
<tr>
<td>Host</td>
<td>388.9</td>
<td>3.8</td>
<td>0 %</td>
</tr>
<tr>
<td>Docker</td>
<td>387.0</td>
<td>7.6</td>
<td>0 %</td>
</tr>
<tr>
<td>VM</td>
<td>393.6</td>
<td>3.45</td>
<td>+1 %</td>
</tr>
</tbody>
</table>

Table 5.2: 2 vCPU performance

## 4 vCPU performance

<table>
<thead>
<tr>
<th>System</th>
<th>time (s)</th>
<th>std</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ironic</td>
<td>192.0</td>
<td>0.8</td>
<td>-9 %</td>
</tr>
<tr>
<td>Host</td>
<td>211.3</td>
<td>2.1</td>
<td>0 %</td>
</tr>
<tr>
<td>Docker</td>
<td>211.7</td>
<td>2.5</td>
<td>0 %</td>
</tr>
<tr>
<td>VM</td>
<td>223.9</td>
<td>1.2</td>
<td>+5 %</td>
</tr>
</tbody>
</table>

Table 5.3: 4 vCPU performance

## 8 vCPU performance

<table>
<thead>
<tr>
<th>System</th>
<th>time (s)</th>
<th>std</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ironic</td>
<td>109.5</td>
<td>1.0</td>
<td>-16 %</td>
</tr>
<tr>
<td>Host</td>
<td>131.4</td>
<td>2.1</td>
<td>0 %</td>
</tr>
<tr>
<td>Docker</td>
<td>131.2</td>
<td>1.4</td>
<td>0 %</td>
</tr>
<tr>
<td>VM</td>
<td>141.1</td>
<td>1.0</td>
<td>+7 %</td>
</tr>
</tbody>
</table>

Table 5.4: 8 vCPU performance
5.1. CPU

From the diagram we can draw some interesting conclusions.

- The bare-metal machine has by far the best performance. This is expected since there is no virtualization overhead and there are no OpenStack services running.
- When the workload is run directly on the compute host, there is scheduling overhead because all the OpenStack services are still running.
- When running inside a VM the hypervisor’s main task is to schedule time in the physical CPU for the guests processes. The hypervisor requests for example 4 cpu cores for a limited amount of time and waits until these resources become available. From the experiment we infer that the number of CPUs of the VM is of major importance when trying to approximate hypervisor overhead. In our test the performance drop for a single or dual cpu, is about 2 %, but with 4 CPUs it rises to 5% and with 8 it rises even more to 7 %. We assume that this trend continues as more CPU resources are requested from physical machines. Therefore we conclude that the more resources a hypervisor must schedule for, the more the performance drops percentage-wise.
- The Docker container is, in terms of performance, on par with the compute host in every test.

5.1.2 CPU contention

Contention is the situation where different parties compete for a single resource. For our test we focus on CPU contention. We create up to 4 VMs and Containers with 4vCPUs and then we run PXZ on them on the same host. The results are presented bellow.

![CPU Contention](image.png)

Figure 5.2: CPU contention illustration
Table 5.5: CPU Contention

<table>
<thead>
<tr>
<th>System</th>
<th>VM</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Instance</td>
<td>216</td>
<td>203</td>
</tr>
<tr>
<td>2 Instances</td>
<td>252</td>
<td>236</td>
</tr>
<tr>
<td>3 Instances</td>
<td>367</td>
<td>339</td>
</tr>
<tr>
<td>4 Instances</td>
<td>483</td>
<td>459</td>
</tr>
</tbody>
</table>

We observe that the absence of the hypervisor also affects the performance in high contention cases. We observe that when the systems “fit” into the host (two 4vCPU systems fit in an 8 core machine) the systems show similar performance. When the consolidation of resources however forces us to pack more VMs or containers into the same host (up to four in our experiment), containers outperform hypervisor based systems.

5.2 Networking

5.2.1 Throughput

We employ nuttcp to determine the network throughput for different data rates. The tool works in a client server model where the system to be tested acts like a client and another machine external to the cloud acts as a server. The client is configured to generate traffic at different rates ranging from 500 to 1100 Mbps. The diagram present our findings for inbound TCP traffic and is indicative of our findings both for inbound and outbound traffic in TCP and UDP.

![Network Bandwidth](image.png)

Figure 5.3: Network Bandwidth
5.2. Networking

The following table displays the findings for maximum throughput achieved in TCP and UDP traffic in both directions. Reverse TCP and UDP traffic was not measured successfully in the time set and has been omitted.

<table>
<thead>
<tr>
<th>System</th>
<th>TCP</th>
<th>Reverse TCP</th>
<th>UDP</th>
<th>Reverse UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ironic</td>
<td>923</td>
<td>929</td>
<td>941</td>
<td>931</td>
</tr>
<tr>
<td>Host</td>
<td>923</td>
<td>–</td>
<td>937</td>
<td>—</td>
</tr>
<tr>
<td>Docker</td>
<td>899</td>
<td>327</td>
<td>932</td>
<td>939</td>
</tr>
<tr>
<td>VM</td>
<td>912</td>
<td>931</td>
<td>939</td>
<td>939</td>
</tr>
</tbody>
</table>

Table 5.6: TCP & UDP maximum performance

From the collected data the following observations can be made.

- In terms of maximum throughput, the bare metal machine achieved the best performance in every case. All the other systems performed similarly and the maximums achieved were just below the ironic case.

- OpenStack internal communication is based on TCP as a protocol. As a result a small part of the network traffic in all cases except Ironic is part of the OpenStack internal messaging system. That is why all the systems except ironic achieved speed which were very close to the maximum but not maximum.

- In terms of UDP traffic all the tests show that all machines were able to reach the maximum. This is expected since internal OpenStack communication is done through TCP and not UDP.

- The Docker container did not perform as well as the host in nearly every case despite several reconfiguration attempts. Further research and experimentation is advised.

5.2.2 Latency

Netperf was the tool of choice in testing the systems latency. For this tool, the machine that is to be tested acts as a server and another machine outside the cloud acts as the client. In order to make a good approximation of system latency we do a fixed time test (30 sec). In this time frame, the tool calculates how many packets were sent and received. Therefore we can approximate the time it takes each packet to reach the destination. We use the formula below from [42] to approximate the system latency.

\[
\frac{1}{\text{TransferRate(Req/s)}} \times \frac{10^6 \mu s}{1s}
\]

The tables below show the results for TCP and UDP round trip times.
Figure 5.4: Network Latency

<table>
<thead>
<tr>
<th>System</th>
<th>Requests / (s)</th>
<th>std</th>
<th>latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ironic</td>
<td>1917</td>
<td>100</td>
<td>260</td>
</tr>
<tr>
<td>Host</td>
<td>2016</td>
<td>59</td>
<td>250</td>
</tr>
<tr>
<td>Docker</td>
<td>1673</td>
<td>67</td>
<td>298</td>
</tr>
<tr>
<td>VM</td>
<td>1380</td>
<td>64</td>
<td>362</td>
</tr>
</tbody>
</table>

Table 5.7: TCP latency performance

<table>
<thead>
<tr>
<th>System</th>
<th>Requests / (s)</th>
<th>std</th>
<th>latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ironic</td>
<td>1944</td>
<td>102</td>
<td>257</td>
</tr>
<tr>
<td>Host</td>
<td>1992</td>
<td>87</td>
<td>250</td>
</tr>
<tr>
<td>Docker</td>
<td>1724</td>
<td>93</td>
<td>289</td>
</tr>
<tr>
<td>VM</td>
<td>1430</td>
<td>43</td>
<td>349</td>
</tr>
</tbody>
</table>

Table 5.8: UDP latency performance

It can be observed from the diagrams that both Ironic and the host show similar performance. However, the addition of the extra layers of networking infrastructure by Neutron(3.3) add some latency to the docker and the VM of approximately 40µs. In the VM’s case, there is also extra latency which is inferred by the hypervisor and as a result a VM has an additional 60 µs added to the previous level for a total of 100 µs compared to the host.
5.3 Memory

The Bandwidth tool which was used performs a thorough test on the memory subsystem. It uses a random data batch, of variable size (from 256bit to 512 MB) to check the system memory. It performs a number of different types of benchmarks but for this thesis we present only the random read and random write speeds. These are more typical characteristics for real applications.

The system under test has the following CPU characteristics.

<table>
<thead>
<tr>
<th>CPU name</th>
<th>set associative</th>
<th>cache blocks</th>
<th>size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Data Cache</td>
<td>8-ways</td>
<td>64 sets</td>
<td>32</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>4-ways</td>
<td>128 sets</td>
<td>32</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>8-ways</td>
<td>512 sets</td>
<td>256</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>16-ways</td>
<td>8192 sets</td>
<td>8192</td>
</tr>
</tbody>
</table>

Table 5.9: UDP latency performance

From the diagrams below (5.5,5.6,5.7) we can see that different types of memory subsystems have vastly different results. As expected, each cache level outperforms the previous level by a considerable factor. Also the CPU caches are approximately ten times faster than RAM.
In order to get a better view of our data set, we present an indicative diagram (5.7) that presents only 4 data-points (256b, 48KB, 768KB, 512MB) from our experiment. Each column represents the data rate at which memory was accessed. We observe the following:

- We observe that in terms of performance there is very little performance deterioration when the memory access happens only at cache level.
• In testing main memory, read performance does not show significant fluctuations regardless of the system tested.

• When testing RAM, write performance of the VM shows considerable drop of approximately 30%.

• Finally we also note that the absence of OpenStack services allows ironic to be faster in most tests by at least 5%.

5.4 Disk I/O

For these tests the nova-docker driver could not be leveraged. The reason for this is that when using a standard container, access is not granted to the Hard drive but instead to the copy-on-write file-system that containers use (AUFS). Therefore, in order to get access to the hard drive a standard Ubuntu container was spawned from dockers CLI and a host file was mounted. All the subsequent testing was performed in that mounted folder.

5.4.1 SysBench

We use Sysbench to create a 16GB set of files and the suite will use an internal 70%/30% read/write automated test to test the I/O subsystem. We measure the I/O requests which are made within 20 minutes by the suite. Because Sysbench cannot force data to be always in the hard driver, the length of the test was extended in order to minimize the effects of caching data in RAM.

![File Test I/O](image)
Every system except the VM presents a similar behavior. It should be noted that for this test the nova-docker driver was not leveraged. Instead a standard ubuntu container was spawned which had a host folder attached.

### 5.4.2 Binary Copy

Finally, a very simple binary copy with the linux dd utility is made. We create a 4.3GB file filled with zeros and then copy it in a single file. The results from the test are presented below.
As in the previous case Ironic and the host showed the same performance. A 6% performance degradation was experienced while using containers and a 30% performance drop was visible in the VM case. We therefore note that in cases of full load, THE KVM/QEMU hypervizor slows down the data transfer considerably. However we speculate that with further tuning higher speeds can be achieved. Nonetheless the I/O result presents us with a more interesting finding. As mentioned in [53] because each request must go through QEMU, the system requires more IOPS and CPU instructions per second for the same absolute performance, leaving fewer resources for useful application work.

### 5.5 Boot-up time

Finally we measure the boot-up time of each system. In order to do that the log in horizon is observed. When the user login state is achieved in the log we report the time. Although this method is not overly accurate, it is a good estimation of the actual time that it takes for a resource to become usable.

<table>
<thead>
<tr>
<th>System</th>
<th>Seconds (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ironic</td>
<td>600</td>
</tr>
<tr>
<td>Docker</td>
<td>5</td>
</tr>
<tr>
<td>LargeVM</td>
<td>105</td>
</tr>
</tbody>
</table>

Table 5.12: Boot-up times

We observe that there are considerable differences in the time it takes for a computing resource to become available to a user. This information is crucial when designing computer systems and should always be taken intro consideration.
Chapter 6

Conclusion

6.1 Summary

In this thesis we benchmarked three distinct OpenStack deployment models. One based on VMs, one based on Docker Containers and one based on bare-metal machines. We highlighted the performance fluctuations for CPU, I/O, memory and networking which take place within each deployment and observe these:

- The performance deterioration which takes places when using VMs should not be viewed as a constant. In our view it is a variable that depends on VM size and in some cases on the specific workload.

- The bare-metal machine displayed the best performance on every metric. Therefore in applications with a protracted full system load Ironic should be considered.

- While an OpenStack cloud which focuses on spawning containers was better on most metrics it requires extra configuration before it can be seriously considered for production environments.

- Containers have only one kernel and multiple user-spaces defined. Therefore Clouds which spawn different operating system kernels (f.g. Windows machines on Ubuntu Hosts) are not possible and that is an important reason for which VMs will continue to be part of modern cloud deployments.

- An OpenStack cloud which can control full physical machines was realized with Ironic. As a result a cloud which is able to control VMs, containers and bare metal machines presents an interesting solution to real deployments.

6.2 Limitations in the current system

At the time of this writing "January 16, 2017" the nova docker driver comes with a series of limitations some of which are quite serious.
• There is a hard-coded limitation in the driver which only allows compute hosts up with with hard drivers up to 26GB. This is obviously not enough for production use, so further configuration is required.

• In OpenStack the nova-scheduler is responsible for deciding in which node the VMs will be spawned. In order to decide the scheduler takes into account a nodes available resources (e.g. the disk space available). As a biproduct of the limitation in the previous point is that the better consolidation of resources with containers theory, cannot be tested because the 26GB limit will be reached before the scheduler limit is reached. So with the driver in the current form we can only spawn the same, or fewer number of containers in a node than VMs.

• By default a container can see some of the internals of its host (for example the CPU) even if it can’t use them. This may or may not be an acceptable behavior.

• In order to get the full bandwidth, the MTU of the docker host must be changed to 1420. This is not possible from within the container and in general one cannot manipulate some of the hardware characteristics of the container which can be troublesome.

• We should also mention that AUFS presents another serious limitation. When trying to install applications from inside the container to the host filesystem the hosts ability to manage application specific privileges (Apparmor) cannot be tampered. Therefore further configuration is needed before running full scale applications on containers with the use of OpenStack because developers do not have access to host to make these changes.

• Currently there is no multi-tenant network isolation for the baremetal nodes. All nodes must reside in the Admin subnet and as result they all belong in the same IP range. This limitation is mainly due to the fact that the bare-metal machine has no internal neutron infrastructure since the node is connected on the physical NIC. Therefore the infrastructure that allows isolation in cases of VMs and containers is not present. This is an interesting problem and there are some efforts in that direction but none are part of the FUEL ecosystem.

6.3 Future work

As mentioned in the document, four distinct parts of the system are benchmarked independently. These are CPU, Memory, IO, Network. From these, both the CPU and Network parts are quite accurate. For memory however we observe that the tool (and every tool that was tested) can only infer and approximate the speed at which the memory access occurred. Therefore, and specially in high speed cache cases, the accuracy of the measurement can be debated. As a result further experimentation is required with better engineered tools. For I/O the container system is solid in our opinion but heavy configuration is always needed to achieve production use.

For the Memory subsystem, Stream bandwidth [22], phonorix suite [27] and RandomAccess [23] where tried but were found to be unreliable.
For the IO subsystem, FIO [24] and IOzone [26] have been used in similar work. However in our experiment they were found to be inaccurate and showing huge fluctuations in their results between individual runs.
Appendix A

Acronyms and Abbreviations

The acronyms and abbreviations presented in table are used in this project report.
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMQP</td>
<td>Advanced Message Queuing Protocol</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AUFS</td>
<td>Another Union File System</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DHCP</td>
<td>Dynamic Host Configuration Protocol</td>
</tr>
<tr>
<td>FwaaS</td>
<td>Firewall as a Service</td>
</tr>
<tr>
<td>ILO</td>
<td>Integrated Lights Out</td>
</tr>
<tr>
<td>IOT</td>
<td>Internet Of Things</td>
</tr>
<tr>
<td>I/O</td>
<td>Input / Output</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel-based Virtual Machine</td>
</tr>
<tr>
<td>LOM</td>
<td>Lights Out Management</td>
</tr>
<tr>
<td>MTU</td>
<td>Maximum Transfer Unit</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>PXE</td>
<td>Preboot Execution Environment</td>
</tr>
<tr>
<td>QEMU</td>
<td>Quick Emulator</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RDO</td>
<td>Rapid Deployment tool</td>
</tr>
<tr>
<td>REST</td>
<td>Representational state transfer</td>
</tr>
<tr>
<td>SDK</td>
<td>Sun Development Kit</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual private network</td>
</tr>
</tbody>
</table>

Table A.1: Acronyms used throughout this report
Appendix B

Testing Configuration

Below we specify indicative testing configurations and instructions used. There is of course a plethora of different configurations but these instructions serve as examples.

B.1 PXZ

In order to use pxz the following command is issued through the terminal.

```bash
/usr/bin/time -p -o times.txt -a /usr/bin/pxz -k -T4 enwik9≫/dev/null
```

- `time` in order to use the Linux time command.
- `times.txt` is the file to output the results of the time command
- `k` to keep the original file.
- `T4` specifies the number of cores to leverage (4).
- `filename` is the file to be used as input.
- `output` is piped to `/dev/null` for destruction.

B.2 Nuttcp

In order to start the server we issue:

```
Nuttcp -S
```

On the client side we issue:

```
nuttcp -l8972 -T30 -u -r -Ri300m/100 -i1 ServerIP
```

- `-l8972` sets the buffer length
- `-T30` sets the time in seconds to run the test
- `-u` sets UDP as the running protocol
- `-r` reverses the data flow
- `-Ri300m/100` limits the rate to 300 Mbps with bursts of 100 packets
- `-i` server_hostname is the destination of our test.
B.3 netperf

On the server side we issue:

```
netperf
```

and on the client side:

```
netperf -H ServerIP -l 30 -t UDP_RR -P 1 -r 200,200
```

- `ServerIP` is the IP of the server already started
- `-l 30` sets the tests at 30 seconds
- `-t UDP_RR` specifies UDP round trip as the test protocol
- `-P 1` suppresses the test banner
- `-r 200,200` sets the packet size at 200 bytes

B.4 Memory

The tool operates in autopilot for every test. Simply install the tool and run:

```
nice -n -2 ./bandwidth
```

B.5 Sysbench

For the SysBench IO test we run:

```
sysbench –test=fileio –file-total-size=16G prepare
to create the files.
```

```
```

```
–max-time=300 –max-requests=0 –num-threads=12 run >>results.txt
```

and the tool handles the rest.

To cleanup after our test we issue:

```
sysbench –test=fileio –file-total-size=16G cleanup
```

B.6 Binary Copy

For a simple binary copy with input /dev/zero and output /home/testfile we issue

```
dd if=/dev/zero of=/home/testfile bs=4G count=2 oflag=direct
```
Bibliography


[31] Tapti Palit; Yongming Shen; Michael Ferdman Demystifying cloud benchmarking Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium
[34] Moritz Raho, Alexander Spyridakis, Michele Paolino, Daniel Raho KVM, XEN and Docker: a performance analysis for ARM based NFV and Cloud computing Virtual Open Systems, Grenoble France
[37] Joel Scheuner, Jrgen Cito, Philipp Leitner, Harald Gall, Cloud WorkBench: Benchmarking IaaS Providers based on Infrastructure-as-Code University of Zurich, Zurich, Switzerland


[49] Openstack Kilo Graph http://docs.openstack.org/kilo/install-guide/install/apt/content/figures/1/a/common/figures/openstack_kilo_conceptual_arch.png, accessed 8/12/2016


