University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Development and Evaluation of an OpenStack Extension to Enforce Cloud-wide, Multi-resource Fairness during VM Runtime

Stephan Mannhart
Steinach, Switzerland
Student ID: 11-917-515

Supervisor: Patrick Poullie, Dr. Thomas Bocek
Date of Submission: March 1, 2016

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

The efficient use of physical resources in compute clouds is managed by scheduling virtual machines (VMs) to suitable hypervisors. However changes in resource consumptions after scheduling can lead to resource contentions on the compute nodes on which the hypervisors run. These contentions need to be addressed, but instead of focusing on each compute node individually, a cloud-wide fair sharing of resources among all cloud users is desirable. Establishing this fair share is a subjective task that allows for many different solutions. This thesis documents the development and evaluation of a framework to enforce fair sharing of resources among users. The fairness is defined through a customizable heaviness-metric which uses resource consumption information of all VMs to determine the heaviness of each user in regard to her VM's consumptions. Based on each user's heaviness, the cloud resources are then reallocated to establish a fair share of resources among users.

Die effiziente Verwendung von physischen Ressourcen in Compute Clouds wird durch das Scheduling von Virtuellen Maschinen (VMs) auf passende Hypervisoren gewährleistet. Veränderungen des Ressourcenverbrauchs nach dem Scheduling können jedoch zu Ressourcen-Engpässen auf den Compute Nodes führen, auf welchen die Hypervisoren laufen. Diese Engpässe müssen gelöst werden, aber anstatt den Fokus auf jeden Compute Node einzeln zu legen, wäre die Betrachtung einer fairen Verteilung von Ressourcen unter Nutzern in der ganzen Cloud wünschenswert. Eine faire Aufteilung der Ressourcen unter den Benutzern zu finden ist eine sehr subjektive Aufgabe, welche auf viele verschiedene Weisen gelöst werden kann. Diese Arbeit dokumentiert die Entwicklung und Evaluierung eines Frameworks zur Etablierung einer fairen Verteilung von Ressourcen unter Nutzern. Die Fairness ist definiert durch eine anpassbare Gewichts-Metrik welche Informationen über den Ressourcenverbrauch aller VMs dazu verwendet, das Gewicht jedes Benutzers bezüglich des Ressourcenverbrauchs ihrer eigenen VMs zu bestimmen. Basierend auf diesem Gewicht werden die Cloudressourcen umverteilt, um eine faire Verteilung von Ressourcen zwischen den Benutzern zu erreichen.

Acknowledgments

I am using this opportunity to thank my advisors Patrick Poullie and Dr. Thomas Bocek for their support and guidance throughout the thesis. A special thanks goes to Patrick Poullie for being available around the clock to answer my questions and help me overcome all obstacles throughout the course of this thesis. I also want to thank Andreas Gruhler, a fellow student whose insights into the topic of resource reallocations helped me tremendously with my own work.

Last but not least I thank my family for allowing me to pursue my thesis in a supportive environment and motivating me along the way.

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Outline 1

1.3 Command-line Instructions 2

2 Design of the Framework **3**

3 Decentralized Message Exchange **5**

3.1 ZeroMQ Broker for Ubuntu 14.04 5

3.2 Setting up ZeroMQ 5

3.2.1 Controller Setup 6

3.2.2 Compute Host Setup 7

4 OpenStack Nova Extension **9**

4.1 Nova-Fairness 9

4.2 Nova API Extension 9

4.2.1 Nova API URL Mapping 10

5 Framework Evaluation **13**

5.1 OpenStack Environment 13

5.2 Setting up Nova-Fairness on Compute Hosts 14

5.3 Setting up the Controller 14

6 Summary and Conclusions **15**

6.1 Future work 15

Abbreviations **19**

List of Figures **19**

A Extending OpenStack to Adapt VM Priorities during Runtime **23**

B Contents of the CD **35**

Chapter 1

Introduction

1.1 Motivation

Virtualization allows a flexible use of data center resources and has gained renewed attention in the last few years [5]. Using virtual machines (VMs) running on physical machines (PMs) brings the advantage of being able to flexibly change machine resources. This however presents growing challenges for the field of Cloud Computing (CC) in regards to efficient resource allocations. With growing clouds and ever-changing workloads, allocating resources to VMs becomes increasingly complex. Using an orchestration layer like OpenStack does only allow to allocate resources based on VM scheduling to hypervisors with sufficient available resources. Since VMs are very dynamic and change their resource consumptions after scheduling, an approach beyond scheduling is needed. By setting the scope on the whole cloud instead of the individual hypervisor and analyzing resource consumptions of cloud users, a possible approach to the resource allocation problem can be in finding a fair resource share among users and reallocating resources on VMs based on the user's over- or underconsumption.

The goal of this thesis is to develop and document a framework based on an OpenStack extension that allows to flexibly reallocate resources among users based on their VMs resource consumptions. The extension should run on multiple compute hosts to allow cloud-wide resource reallocations. The effectiveness of the extension will be evaluated through different scenarios involving single-, as well as multi-host setups with 1 to 2 resources involved.

1.2 Thesis Outline

This thesis is largely based on the white paper "Extending OpenStack to Adapt VM Priorities during Runtime" co-authored by me [12]. The white paper is also included as Appendix A in its entirety.

This report provides additional technical information about the framework. Each chapter also contains pointers to sections in the white paper which discusses the topic from a conceptual- and implementation-specific viewpoint.

Chapter 2 discusses high-level design decisions concerning the framework.

Chapter 3 presents the advantages of using decentralized communication as well as how such a communication scheme can be set up using ZeroMQ.

Chapter 4 focuses on the extension of OpenStack through a custom nova service as well as a nova compute API extension.

Chapter 5 outlines the evaluation of the framework and talks about the chosen environment for the evaluation.

1.3 Command-line Instructions

As mentioned in Section 1.2, this report provides additional technical information in regards to the framework developed in the course of this thesis. Part of these technical information are Linux command-line instructions, e.g.,

```
# apt-get install redis-server
```

These instructions are meant to be executed with root privileges. The instructions have only been tested on Ubuntu 14.04 and may not work on other Linux distributions.

Chapter 2

Design of the Framework

The theoretical problem in regards to resource organizations in clouds and the resulting problems when defining cloud fairness and according requirements is discussed in [12, Section 3].

The design implications for a framework to enforce fairness is discussed in the theoretical problem part of the white paper, outlined in [12, Section 5].

Chapter 3

Decentralized Message Exchange

The benefits of using decentralized communication between compute hosts in the OpenStack environment of the framework instead of relying on the default setup with a centralized message broker is outlined in [12, Section 5.3.1].

3.1 ZeroMQ Broker for Ubuntu 14.04

The framework was developed for the "Juno" release of OpenStack running on Ubuntu 14.04 [11]. The individual ZeroMQ brokers on the compute hosts needed for the decentralized message exchange are not detailed in the official OpenStack installation guide for Ubuntu 14.04 [11].

Newer versions of Ubuntu already include a package that can be installed through the Ubuntu package manager, however the target version Ubuntu 14.04 "trusty-thar" lacks such a package [2]. To simplify the installation of the ZeroMQ broker on each compute host, I re-packaged the Ubuntu package to be compatible with Ubuntu 14.04 by using "debtool" [15].

The custom ZeroMQ broker package can be found on the CD as outlined in Appendix B

3.2 Setting up ZeroMQ

The setup for ZeroMQ includes the compute hosts as well as the controller node. Instead of a single broker running on the controller, brokers on each compute host as well as the controller need to be set up as discussed in [12, Section 5.3.1]

3.2.1 Controller Setup

The controller hosts the redis in-memory data store which is needed for discovery [14]. For each OpenStack nova service, an entry for compute hosts running the service is created.

Installing redis can be accomplished through the Ubuntu package manager by issuing:

```
# apt-get install redis-server
# apt-get install python-redis
```

This installs the redis server along with the python bindings necessary for OpenStack.

To configure the redis server, the line **bind 127.0.0.1** needs to be removed from the redis server configuration file located at `/etc/redis/redis-server.conf` to get redis to listen to all request, not just requests comming from the local host. To restart the redis server with the changed configuration file, issue the following command:

```
# service redis-server restart
```

The ZeroMQ broker can now be installed with the custom ZeroMQ package:

```
# dpkg -i --force-overwrite oslo-messaging-zmq-receiver_1.8.3-0ubuntu0.14.04.3_all.deb
```

The **--force-overwrite** option is used to apply a fix within the custom ZeroMQ package for a known bug in the `matchmaker_redis.py` Python source for the OpenStack integration of ZeroMQ [8].

To set ZeroMQ as the remote procedure call (RPC) backend of OpenStack nova, the following lines have to be added to the [**DEFAULT**] section of the `/etc/nova/nova.conf` configuration file:

```
1 host=HOSTNAME
2 rpc_backend=zmq
3 rpc_zmq_host=HOSTNAME
4 rpc_zmq_matchmaker=oslo.messaging._drivers.matchmaker_redis.
     MatchMakerRedis
5 rpc_zmq_ipc_dir=/var/run/zeromq
```

HOSTNAME is a placeholder for the actual hostname of the host on which the ZeroMQ broker is running. The matchmaker needs some additional configurations in a new section added to the configuration file:

```
1 [matchmaker_redis]
2 host=controller
3 port=6379
```

The line **host=controller** sets the controller node as the host on which the redis server is located. To finalize the ZeroMQ installation on the controller node, the RabbitMQ service needs to be stopped and all nova services as well as the new ZeroMQ broker service need to be restarted:

```
# service rabbitmq-server stop
# service oslo-messaging-zmq-receiver start
# service nova-api restart
# service nova-cert restart
# service nova-conductor restart
# service nova-scheduler restart
```

3.2.2 Compute Host Setup

On the compute host, only the ZeroMQ broker package as well as the Python redis bindings need to be installed:

```
# apt-get install python-redis
# dpkg -i --force-overwrite oslo-messaging-zmq-receiver_1.8.3-0ubuntu0.14.04.3_all.deb
```

The necessary changes to the nova configuration are identical to the controller configurations:

```
1 host=HOSTNAME
2 rpc_backend=zmq
3 rpc_zmq_host=HOSTNAME
4 rpc_zmq_matchmaker=oslo.messaging._drivers.matchmaker_redis.
     MatchMakerRedis
5 rpc_zmq_ipc_dir=/var/run/zeromq
6
7 [matchmaker_redis]
8 host=controller
9 port=6379
```

To finalize the installation, the ZeroMQ broker service has to be started:

```
# service oslo-messaging-zmq-receiver start
```

Chapter 4

OpenStack Nova Extension

As outlined in [12, Section 5], the framework is an extension of OpenStack nova. For the framework, two venues of extending OpenStack nova have been chosen: A custom nova service called "nova-fairness" for the main framework logic and a nova compute API extension as an interface for third party developers.

4.1 Nova-Fairness

The *nova-fairness* service contains the main framework logic. The functionality of the service is outlined in [12, Section 5]. This section discusses the OpenStack components used for the service.

Based on the requirement to autonomously manage the resource consumptions of VMs and reallocate resources if necessary, the service functions largely independent of OpenStack nova. A service init-script located at `/etc/init/nova-fairness` makes sure the *nova-fairness* service is started at boot through the *nova-fairness* service start-script located at `/usr/bin/nova-fairness`. The start-script contains a call to the **main()** method for the service located at `nova/cmd/fairness.py`.

The **main()** method in `nova/cmd/fairness.py` registers the service with OpenStack nova and sets database access through the *nova-conductor* service running on the controller node. This starter-script defines the relevant information including the *nova-fairness* manager class. The manager class, defined through a configuration entry, handles all RPC requests of the *nova-fairness* service as well as the autonomous functionality through periodic tasks as outlined in [12, Section 5].

4.2 Nova API Extension

The OpenStack nova compute API allows third party developers to access nova functionality through a RESTful interface [10]. For this thesis, the nova compute API has been

extended to provide an interface to set the metric to be used by the framework on the different compute hosts through the central access point of the API.

The API extension for the framework outlined in this thesis uses its own WSGI controller [13].

4.2.1 Nova API URL Mapping

Since the nova compute API is a RESTful interface, all communications with the API take place over HTTP [4].

OpenStack nova uses the Routes package which itself is a clone of the Rails routes implementation for Python [1]. The main mapping of URLs for the nova compute API is done in `nova/api/openstack/compute/__init__.py` with the two methods **mapper.connect()** as well as **mapper.resource()**. URLs are directly mapped to action methods defined in WSGI controllers [7][13].

The **mapper.connect()** method is used to connect a single URL to a single action on a WSGI controller. The example below shows the mapping of / to versions.py, so calling `http://controller:8774/v2/tenant-id/v2/` calls the WSGI controller defined in versions.py and executes the **show** method:

```
1 self.resources['versions'] = versions.create_resource()
2 mapper.connect("versions", "/",
3               controller=self.resources['versions'],
4               action='show',
5               conditions={"method": ['GET']})
```

On the other hand, the **mapper.resource()** method allows the mapping of multiple URLs to actions on a controller. This method was chosen for the API extension of the framework as it allows the flexible definition of additional actions without needing to define multiple URL mappings.

The URL mapping for the framework's API extension looks as follows:

```
1 self.resources['fairness'] = fairness.create_resource()
2 mapper.resource("fairness", "fairness",
3                controller=self.resources['fairness'],
4                member={'action': 'POST'})
```

Through this mapping, the framework's API extension can directly be accessed over HTTP at the URL `http://controller:8774/v2/TENANT_ID/COMPUTE_HOST/action`.

The **TENANT_ID** specifies the "tenant" or "project" the user belongs to, who intends to access the API [9].

Specific actions are defined through a JSON payload provided with the API-call, for example:

```
1       {"set-metric":{"name":"GreedinessMetric"}}
```

To call the **set-metric** action through the API with the argument **name** being **GreedinessMetric**. A complete API-call can therefore be made in cURL as follows [3]:

```
# curl -H "X-Auth-Token:AUTH_TOKEN" -H "Content-Type:application/json" -X POST -d '"set-metric":"name":"GreedinessMetric"' http://controller:8774/v2/TENANT_ID/fairness/COMPUTE_HOST/action
```

The **AUTH_TOKEN** needs to be requested through the nova compute API beforehand for authentication purposes.

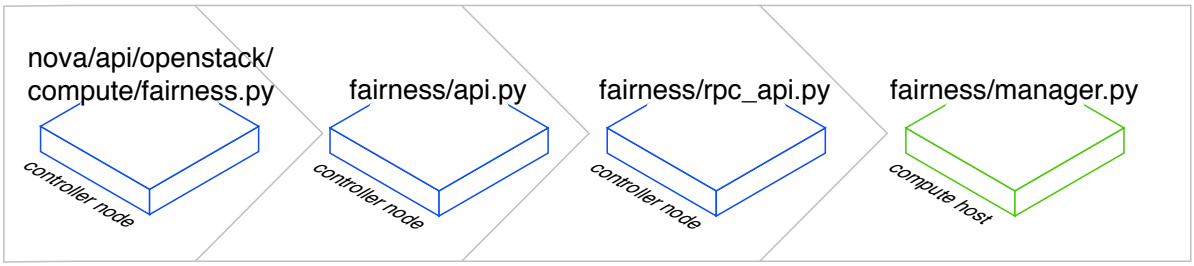

Figure 4.1: The complete call stack for a *nova-fairness* API call

Figure 4.1 shows the complete API call stack. The API call is received and analyzed by the controller node which in turn calls the appropriate compute host running the *nova-fairness* manager.

Chapter 5

Framework Evaluation

The detailed evaluation results can be found in [12, Section 6]. This chapter discusses the evaluation environment based on an OpenStack "Juno" installation running on Ubuntu 14.04.

5.1 OpenStack Environment

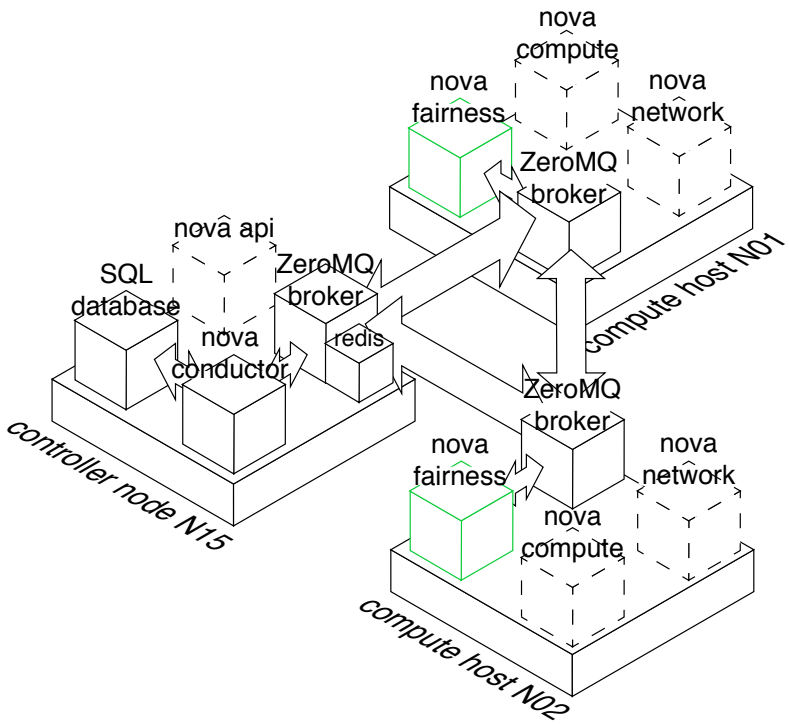

Figure 5.1: The OpenStack environment used for the framework evaluation

Figure 5.1 shows the OpenStack environment which was used for the framework evaluation. A total of 2 compute hosts and 1 controller node were set up to run the evaluations outlined in [12, Section 6.4].

The *nova-fairness* services use the ZeroMQ brokers to exchange data used for the resource reallocations. The *nova-network* as well as *nova-compute* services don't play a direct role in the framework, however they are needed to manage the VMs. On the controller node, The *nova-conductor* service provides access to the SQL database and relays database information through ZeroMQ to all compute hosts.

5.2 Setting up Nova-Fairness on Compute Hosts

After following the OpenStack installation guide [11] and setting up a minimal compute host with *nova-compute*, *nova-network* and *nova-api-metadata*, the custom *nova-fairness* service has to be installed.

The init-script is available through a custom built Ubuntu package which can be found on the CD alongside the Python code for the *nova-fairness* service as listed in Appendix B. First, the default OpenStack nova code installed as outlined in the OpenStack installation guide needs to be replaced with the Python code containing the *nova-fairness* service:

```
# rm -rf /usr/lib/python2.7/dist-packages/nova
# mv /path/to/CD/nova /usr/lib/python2.7/dist-packages/
```

Before installing the *nova-fairness* Ubuntu package, ZeroMQ needs to be installed and configured. Follow Section 3.2.2 for the ZeroMQ broker installation on the compute host.

Install the `nova-fairness_1:2015.11.1_all.deb` Ubuntu packge:

```
# dpkg -i nova-fairness_1:2015.11.1_all.deb
# apt-get -f install
```

The command **apt-get -f install** is needed to install dependencies. Included in this package is a set of `rootwrap.d` filters that allow the *nova-fairness* service to execute commands as root in a restricted manner controlled by regular expressions [6].

5.3 Setting up the Controller

The controller does not run the *nova-fairness* service since it does not host any VMs. It does however still need the custom OpenStack nova Python code for the *nova-fairness* API bindings to work. Replace the default installation of OpenStack nova with:

```
# rm -rf /usr/lib/python2.7/dist-packages/nova
# mv /path/to/CD/nova /usr/lib/python2.7/dist-packages/
```

To set up the redis in-memory data store as well as the ZeroMQ broker on the controller, follow the installation instructions in Section 3.2.1.

Chapter 6

Summary and Conclusions

This framework uses the approach of enforcing fairness through runtime VM prioritization instead of using scheduling. This allows the OpenStack environment to react to resource contentions and reduce the need of VM evacuation or rescheduling.

Implementing multi-resource fairness during VM runtime in OpenStack requires a solid framework that allows seamless communications between all compute hosts involved. The multitude of resources involved and the lack of knowledge in regards to the utility functions of the different cloud users asks for a simple solution to generate a heaviness scalar that allows easy comparison between the user's resource consumptions.

The framework presented in this paper allows for a flexible implementation of different approaches towards calculating such a heaviness scalar to allow future research in regards of evaluating efficiencies of different heaviness metrics.

6.1 Future work

Performance is an important consideration for the whole framework which lead to many design decisions to further increase speed while reducing computing as well as communication overhead. There are however still performance bottlenecks present in the framework, namely the impact of network bandwidth prioritization with an increasing amount of VMs as outlined in [12, Section 6.3].

The whole installation process of the framework is still very labor-intensive as the original OpenStack nova code needs to be manually replaced with the code that includes the *nova-fairness* service as outlined in Sections 5.2 and 5.3. An automatic setup that would place the *nova-fairness* code at the right place inside the existing OpenStack nova code could be configured for a future version of the framework.

Bibliography

[1] Ben Bangert, Mike Orr. Routes Documentation. https://routes.readthedocs.org/en/latest/, 2015. Online; accessed February 18th, 2016.

[2] Canonical Ltd. Ubuntu Package Search Results - oslo-messaging-zmq-receiver. http://packages.ubuntu.com/search?keywords=oslo-messaging-zmq-receiver, 2016. Online; accessed February 17th, 2016.

[3] Daniel Stenberg. cURL. https://curl.haxx.se/, 2016. Online; accessed February 20th, 2016.

[4] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 407–416, New York, NY, USA, 2000. ACM.

[5] Greg Goth. Virtualization: Old technology offers huge new potential. *IEEE Distributed Systems Online*, (2):3, 2007.

[6] OpenStack Foundation. Rootwrap. https://wiki.openstack.org/wiki/Rootwrap, 2012. Online; accessed February 18th, 2016.

[7] OpenStack Foundation. Adding a Method to the OpenStack API. http://docs.openstack.org/developer/nova/addmethod.openstackapi.html, 2014. Online; accessed February 18th, 2016.

[8] OpenStack Foundation. Bug #1290772: Matchmaker Redis Wrong Key Value Error. https://bugs.launchpad.net/nova/+bug/1290772, 2014. Online; accessed February 17th, 2016.

[9] OpenStack Foundation. Managing Projects and Users. http://docs.openstack.org/openstack-ops/content/projects_users.html#projects_or_tenants, 2014. Online; accessed February 18th, 2016.

[10] OpenStack Foundation. OpenStack nova API. http://developer.openstack.org/api-ref-compute-v2.1.html, 2014. Online; accessed February 18th, 2016.

[11] OpenStack Foundation. OpenStack Installation Guide for Ubuntu 14.04. http://docs.openstack.org/juno/install-guide/install/apt/content/index.html, 2015. Online; accessed February 4th, 2016.

[12] Patrick Poullie, Stephan Mannhart, and Burkhard Stiller. Extending OpenStack to Adapt VM Priorities during Runtime. White paper, University of Zurich, Department of Informatics, February 2016.

[13] Python Software Foundation. PEP 0333 – Python Web Server Gateway Interface v1.0. `https://www.python.org/dev/peps/pep-0333/`, 2003. Online; accessed February 18th, 2016.

[14] Redis Labs. Redis. `http://redis.io/`, 2015. Online; accessed February 5th, 2016.

[15] Six. Debtool: Simple Tool to facilitate Downloading and Repacking Debian Archives. `https://github.com/brbsix/debtool`, 2015. Online; accessed February 17th, 2016.

Abbreviations

API	Application Programming Interface
CC	Cloud Computing
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
MQ	Message Queue
PM	Physical Machine
REST	Representational State Transfer
RPC	Remote Procedure Call
URL	Uniform Resource Locator
VM	Virtual Machine
WSGI	Web Server Gateway Interface

List of Figures

4.1 The complete call stack for a *nova-fairness* API call 11

5.1 The OpenStack environment used for the framework evaluation 13

Appendix A

Extending OpenStack to Adapt VM Priorities during Runtime

Extending OpenStack to Adapt VM Priorities during Runtime

Patrick Poullie, Stephan Mannhart, Burkhard Stiller
University of Zürich, Department of Informatics (IFI), Communication Systems Group (CSG),
Binzmühlestrasse 14, CH–8050 Zürich, Switzerland
poullie@ifi.uzh.ch, stephan.mannhart@uzh.ch, stiller@ifi.uzh.ch

ABSTRACT

In recent years fairness issues in data centers have been pointed out and mostly been addressed by job/VM scheduling. Clouds are a special case of data centers, where resources are deployed by virtual machines. Contrary to other data center types, the resource utilization of VMs in clouds changes dynamically after VMs are started (scheduled). Therefore, fairness in clouds can not solely be established by VM scheduling but also VM priorities during runtime have to be adapted. This paper develops an OpenStack framework that (i) continuously gathers information about VMs' utilization of CPU time, RAM, disk I/O, and network access during VM runtime, (ii) aggregates this information to the "heaviness" of users, and (iii) adapts VM priorities via libvirt to align the heaviness and accordingly establish fairness between users. Because fairness is an illusive and subjective concept, *i.e.*, it differs from person to person, the framework allows to flexibly define "heaviness" and to allow customizing the definition of fairness that is enforced.

1. INTRODUCTION

Cloud Computing (CC) is a computing paradigm enabled by the ever growing connectivity provided by modern communication systems combined with virtualization technology [30]. CC allows server farms to provide their combined computing power on demand to numerous customers, such as end-users and companies. These customers start Virtual Machines (VMs), which are hosted by the cloud's Physical Machines (nodes). Since CC allows numerous users to share the same physical infrastructure, resources, such as CPU, RAM, disk space, and network access, provided through CC are subject to conflicting interests [7]. While in public/commercial clouds resource allocation to VMs is prescribed by SLAs [5], private clouds, clusters, and grids are often a commodity. Therefore, it is important to allocate resources of these commodity infrastructures, such that fairness is ensured. In CC resource allocation consists of the following two steps, which are conducted continuously and in parallel.

VM scheduling During this step the cloud's orchestration layer decides which VM is started next and which node hosts the VM. This step statically multiplexes nodes and is conducted for every VM that is started or live-migrated.

Runtime Prioritization During this step a node's hypervisor, *i.e.*, the operating system of a node, allocates the node's Physical Resources (PRs), such as CPU time, RAM, disk I/O, and network access, to the VMs. This step statically multiplexes PRs of individual nodes and is conducted permanently.

While the first step decides, which node will host a VM, the second step decides how many PRs the VM will receive of this node. In particular, because CPU time, disk I/O, and network access are time-shared, the second step allows to efficiently allocate these resources by priorities. Only for RAM, which is space-shared, the (re)allocation may come with a certain cost or overhead. However, reallocating RAM in the second step still outperforms live-migrating a VM (in the first step). Therefore, the second step allows to flexibly change the performance of VMs, by changing priorities. For example, a VM scheduled to a weak node may perform better than when scheduled to a powerful node, if the weak node prioritizes the VM. Accordingly, both allocation steps are equally important for the cloud resource allocation process. Nonetheless, the first step seems to receive more attention than the second.

To close this gap, this paper designs an OpenStack framework to actively adapt the runtime prioritization of VMs in order to achieve fairness among cloud users.

2. RELATED WORK

Fairness issues in virtualization environments were first pointed out by [10]. As a solution, [10] defines Bottleneck-based Fairness (BBF). [8, 16] provide theoretical insights on this concept and [31] details how BBF fairness can be implemented between processes that share CPU time, network access and disk I/O. The most prominent definition of data center fairness is Dominant Resource Fairness (DRF) as introduced in [12]. DRF has been extended in many directions, for example by [24, 20, 11, 32, 2] to name a few. A third approach to introduce fairness in data centers is the extension of Proportional Fairness [9] to multiple resource [3, 4].

All three approaches (BBF, DRF, proportional fairness) assume that resources are required in static ratios and can, therefore, be applied during scheduling. However, during runtime resource utilization does not exhibit this simplistic dependency, wherefore, the approaches cannot be applied. The only exception is [31] which describes how to achieve BBF between host processes.

Because resources during runtime have to be allocated by assigning priorities to VMs, functions that map (multi-

resource) consumption vectors to (priority) scalars are better suited. Also, the need for assuming utility functions is avoided, because these functions take the actual utilization as input. [18, 17] present such priority functions. Although they deploy the functions only for job scheduling, these functions could also be used to calculate the priority of VMs during runtime.

The only works on data center fairness that incorporate runtime prioritization are [19, 13, 14]. [19] allows users to prioritize among their VMs during runtime. [13, 14] deploys token buckets to shape job runtime resource consumption to the jobs scheduling profile.

3. THEORETICAL PROBLEM

This section describes how cloud resources are organized to highlight problems when defining cloud fairness and according requirements.

A cloud efficiently processes varying workloads by VMs, which are dynamically started on the clouds nodes. VMs are defined by Virtual Resources (VRs), *e.g.*, virtual CPU (VCPU) and virtual RAM (VRAM), often chosen from a range of different *flavors*, *i.e.*, a VM flavor is a set of VRs that a VM of that flavor has. Especially in private clouds, resources may be managed by quotas, *i.e.*, each user has a quota that defines a maximum of VRs that the user's VMs may have in total.

While VMs compete for resources, fairness is to be enforced among users, because users are the entities that are actually entitled to the cloud resources. This complicates finding a fair allocation: Fairness has to be enforced among users by allocating PRs to "intermediaries", *i.e.*, VMs. Furthermore, the PRs that can be allocated to a VM are constrained by the PRs of the node that hosts the VM.

3.1 Multi-resource Fairness

Cloud runtime fairness cannot be uniquely defined. The reason is that heterogenous resources, such as CPU time, RAM, disk I/O, and network access, are shared, while users have different demands. For example, some users may require more CPU for their workloads while others require more RAM. A third user may deploy the cloud for backups and therefore mostly requires disk-space and bandwidth. Thus, resource shares are not objectively comparable.

In economics, the problem of defining fairness is solved by definitions based on consumer's utility functions (a consumer's *utility function* maps each bundle to a number quantifying the consumer's valuation for the bundle). However, because cloud user demands (and, therefore, utility functions) change frequently during runtime, such definitions can neither be applied nor enforced.

The only information accessible in clouds about the user's resource demands, are the resources that have been and are currently utilized. Therefore, a cloud applicable fairness definition, needs to be based on the utilization so far. Given these constraints, cloud fairness has to be defined as *prioritizing cloud users inversely to their heaviness.* That is, users that put little load on the cloud are prioritized at the expense of heavy users. This defers the problem of defining cloud fairness to the problem of defining cloud user heaviness.

3.2 Defining User Heaviness

It seems obvious to define heaviness of a user to include the sum of heavinesses of the user's VMs. Therefore, the heaviness of a VM has to be defined by a metric that maps the VM's (multi-resource) consumption vector to a (heaviness) scalar. Applicable metrics can be found in [18, 17, 25]. Next, desirable characteristics of VM and user heaviness metrics are discussed in Section 3.2.1 and 3.2.2. Based on this discussion, Section 3.3 concludes on information that is essential to achieve these characteristics.

3.2.1 Rewarding Correct VM Configuration

In order to reward the correct configuration of VMs, the heaviness metric must take the flavor/VRs of a VM into account. Consider a small VM vm_s, a medium VM vm_m, and a large VM vm_l, where the size of a VM refers to its VRs. Now assume that all VMs execute the same workload and, therefore, utilize the same resources, which are in conformity with the VRs of vm_m. Because the size of a VM determines, how much resources a VM is expected to utilize, the size of a VM determines how many resources have to be reserved for a VM and, accordingly, which nodes qualify to host the VM. Therefore, the set of nodes that can accommodate vm_s will be larger than the set that can accommodate vm_m or vm_l. Accordingly, executing the workload on vm_s will put a higher load than expected on the VM's host node and executing the workload on vm_l will lead to a node that reserves more resources for the VM than needed. In contrast, executing the workload on vm_m will result in the "anticipated" workload and, therefore, not waste or overuse resources.

Accordingly, the metric should quantify the heaviness of the three VMs, such that it is most advantageous for a user to execute the workload on vm_m and in general, the metric should result in lower user heaviness, if a user configures the VMs in accordance with the subsequent workload.

3.2.2 Discouraging Idle VMs

In order to reward economic behavior, the heaviness metric should give incentive to users to keep the number of instantiated VMs low, if possible. Consider two users u_a and u_b, that utilize the same amount of cloud resources. u_a utilizes this resource amount by one busy VM, which utilizes 50 times the amount of resources compared to its idle state. In contrast, u_b has instantiated 50 VMs of the same flavor, which are idle. In order to be able to provide for u_b's VMs if necessary, the cloud is forced to reserve a large amount of resources. Although over-provisioning will make this factor of reserved resources smaller than 50, B stresses the cloud resources significantly more than u_a.

3.3 Information to Be Collected

The most important factor that determines the heaviness of a user are the resources the user's VMs utilize. Therefore, the resources VMs utilize during runtime must be collected for every VM. This information is referred to as a VM's Runtime Utilization Information (RUI).

Section 3.2.1 highlights that it should be rewarded, when a VM's resource utilization is in harmony with its virtual resources and Section 3.2.2 highlights that it should be rewarded to instantiate VMs economically. To determine these characteristics for each user, the VMs' VRs need be input into the metric.

Resources are heterogenous and measurable in different units. Therefore, resources need to be normalized to calculate a scalar that represents all resources equally. This nor-

malization has to happen with respect to the overall cloud resource supply, to have a cloud-wide unique resource utilization. Accordingly, the framework must collect the resource capacities of each node, referred to as Node Resource Information (NRI), to calculate the overall cloud resource supply.

In summary, the heaviness metric has to receive each VM's (i) RUI, (ii) VRs/flavor, (iii) node name, and (iv) owner as well as each node's NRI as input to the metric.

4. METRIC DESIGN CONSIDERATIONS

This section describes practical considerations, when designing the fairness metric.

4.1 VM endowment

Resource overcommit factors critically determine, how much resources are budgeted for VMs. For example, let c_5 be a cloud that overcommits CPU with factor 5, let c_{15} be a cloud that overcommits CPU with factor 15, and let vm be a VM with a fixed set of VRs. When vm is hosted in c_5 the amount of CPU cycles that is budgeted for vm is on average three times higher, than when vm is hosted in c_{15}. Accordingly, the amount of resources a cloud budgets for a VM not only depends on the VM's VR but also on the clouds overcommit factors.

This makes it hard to define how much resources are budgeted for a VM and, accordingly, the degree to which a VM's resource utilization is in harmony with its VRs (cf. Section 3.2.1) is hard to quantify. In order to tackle this problem, the *endowment* of a VM is defined as the share of the VM's node's PRs that are proportional to the VM's VR. For example, when a VM has three VCPUs and runs on a node, which hosts VMs with a total of 15 VCPUs, then the VMs CPU endowment is 1/5 of the nodes CPU cycles.

4.2 Fairness Quota

Just as VRs cannot be translated directly to the amount of PRs that a cloud budgets for a VM, the quota of a user cannot be directly translated to the amount of PRs a cloud budgets for a user. For example, a user's quota in OpenStack, referred to as the *real quota* subsequently, does not change with the cloud size or the number of users (although it can be customized).

However, in order to define the degree to which a user deploys its real quota, and thereby tackle the problem outlined in Section 3.2.2, a user quota, that represents the amount of PRs a cloud can allocate to each user, is needed. Therefore, the *fairness quota* of a user is defined as the overall cloud resource supply divided by the number of users. In case real quotas are different, also the fairness quotas have to be defined according to this inequality. This may, for example, be done by dividing the overall cloud supply in proportion to the real quotas to calculate the fairness quota.

4.3 Implications

The fairness quota and the VM endowments are on the same scale, for the following reason: the endowments partition a node's PRs among the hosted VMs and the sum of node PRs is the overall cloud supply, which is equally shared among all users to calculate the user's fairness quota.

Accordingly, a user's heaviness can generally be defined as the sum of endowments of the user's VMs (to discourage instantiating idle VMs) plus the heaviness of each VM. This heaviness is defined by a function that takes the VM's endowment and RUI as input and returns a scalar. As discussed in Section 4.1, the endowments also represent the VM's VR in proportion to the VM's node and, therefore, also accounts for the cloud's overcommitment factors. Lastly, the fairness quota has to be substracted from the user heaviness. While this subtraction can be disregarded, when all consumers have the same quota, it is important, when users have different quota (wherefore, also the fairness quota should be different).

5. THE FAIRNESS FRAMEWORK

Out of the box, OpenStack performs scheduling but does not leverage runtime prioritization, to steer the resource allocation in any direction. Two essential types of nodes in the OpenStack architecture, are the controller node and compute nodes. The controller node coordinates the cloud, *e.g.*, by scheduling VMs, and stores essential information to be accessed by other nodes or administrators. The compute nodes perform the actual processing of workloads, *i.e.*, hosting VMs. For this purpose, every compute node runs the OpenStack service *nova-compute* which is part of the OpenStack compute project called nova.

Compute nodes are the nodes that have direct access to the VMs and, therefore, can gather information about their RUI and adapt their priorities. Thus, the framework is implemented as an additional nova service called *nova-fairness*.

OpenStack nova knows three kinds of managing components: Services, managers and drivers [23]. Services are generic containers that group compute node functionalities. Each service has its own managers to control a specific aspect of the node and execute tasks. Managers encapsulate functionalities of the service by providing methods and periodic tasks to deploy the functionality. The manager of the nova-fairness service runs autonomously on each compute node and executes the following steps periodically:

1. Announcing of the node's NRI to the cloud and collecting other node's NRI.
2. Gathering of RUI for all VMs hosted by the node.
3. Applying a heaviness metric to map this RUI to scalars and distributing the scalars and the VM's endowments to all reachable compute nodes that run the fairness-service.
4. When the scalars from all reachable compute nodes are received, the scalars are aggregated for each user to assign priorities to the hosted VMs based on the aggregates of their owners.

The gathering of RUI is accomplished with the third main managing component of OpenStack, the driver. The libvirt virtualization API provides methods to access a multitude of different libvirt drivers communicating with the hypervisors which allow to collect detailed RUI as outlined in Section 5.1.2 [26].

5.1 Resources

Currently the framework accounts for the following six resources: (i) CPU time in seconds, (ii) memory used in kilobytes, number of bytes (iii) read from disk and (iv) written to disk, and number of bytes (v) received and (vi) sent

through the network interface. Accordingly, these resources are represented in the RUI and endowments of VMs. Accounting for other resources, such as software licenses, GPUs, or disk space is future work.

5.1.1 Weighted CPU Time

CPU time, *i.e.*, the amount of time used for a specific CPU task to complete [29], measures CPU usage. Unfortunately, the CPU time provided by different nodes is not directly comparable. The reason is that cloud nodes are rarely homogeneous [15] and, therefore, equipped with different CPUs. Accordingly, one second of CPU time on a powerful node is more valuable than one second of CPU time on a less powerfull node. To compare CPU time across nodes the framework normalizes CPU time by the nodes BogoMIPS. "BogoMIPS" are a metric provided by the linux operating system to capture the performance of different CPUs. However, BogoMIPS are not a scientifically reliable measure to compare different CPUs, wherefore other normalization references, such as the SPEC value [28] will be implemented in the future.

5.1.2 Compiling RUI

Each node collects the RUI of the hosted VMs by a periodic task of the nova-fairness service. Each iteration, the task requests a list of VMs that are running on its node by an RPC to the nova-conductor service. This list contains only VMs that have been successfully scheduled by OpenStack on the node. Therefore, the list indicates for which VMs RUI have to be collected.

The libvirt API provides access to the RUI for each VM and ensures the framework's compatibility with numerous hypervisors.

For time-shared resources (CPU time, disk I/O, network access), the libvirt API provides the accumulated resource consumption since boot time. Therefore, the framework calculates the RUI for the current measurement period by subtracting the accumulated consumption at the beginning of the period from the accumulated consumption at the end of the period. For space-shared resources (RAM) the libvirt API provides the current usage, which the framework uses to represent RAM consumption in the RUI vector.

5.1.3 NRI

The NRI of a node specifies its hardware resource capacities. Therefore, the NRI per node is stable, which allows to capture NRI without a temporal component. The different NRI data-points considered by the framework are (i) Amount of CPU cores on the compute node, weighted with the node's BogoMIPS (ii) Combined disk read speeds of all disks in bytes/s (iii) Network throughput in bytes/s (iv) Total amount of installed memory in kilobytes

5.2 Heaviness Metric

The framework allows to customize the metric that determines the heaviness of users and VMs. To this end a generic map-function is inherited and overwritten by the specific metric to be used and gets called by the framework. The metric to be used is specified in the nova configuration. The configuration entry contains the complete class path of the metric. This class path is checked by the framework for correctness, *i.e.*, whether a correct metric-class with the needed mapping-functions exists at the specified path. In particular, the mapper has to map the input discussed in Section 5.1 to a heaviness scalar for each VM.

5.2.1 Calculating Endowments

The endowments calculation is not part of the heaviness-metric, but the VM endowments can be used by the metric. In particular, the framework calculates the CPU endowments of VMs as discussed in Section 4.1. The memory endowment is defined by a VMs VRAM. This is possible as long as RAM is not overcommitted, which is the case for the used OpenStack configuration. However, when the overcommit factor of RAM is greater than 1, the RAM endowment should be calculated by dividing a node's RAM among hosted VMs proportional to the VMs' VRAM. Because VM flavors do not specify virtual disk I/O or network access, the endowment of these resources is calculated by dividing node capacity equally among the hosted VMs.

5.3 Message Exchange

This section discusses how the framework is able to exchange information between asynchronized nodes in a decentralized manner.

5.3.1 Decentralization

The nova services, which run on every compute node, have a server counterpart on the controller node. The server hosts a centralized message broker that handles all messaging tasks between the client and server parts of the nova services. Therefore, by default, all information exchanged between compute nodes has to traverse the controller node.

The outline of the frameworks's steps shows that exchange of information (NRI, VM scalars, and endowments) between compute nodes is required periodically. Accordingly, all information required to be exchanged by the framework has to traverse the controller node. This is not scalable and introduces a single point of failure (although, arguably, the controller node is always a single point of failure). To overcome this weakness the message exchange needs to be decentralized.

OpenStack enables message exchange between nodes by message queues. OpenStack's default message queuing protocol is the AMQP implementation "RabbitMQ" [6]. AMQP is centralized, wherefore using RabbitMQ implies that all inter-node communication traverses the RabbitMQ broker on the controller node.

To allow for decentralized information exchange between compute nodes, the ZeroMQ [21] message queuing system is deployed for inter-node information exchange. In particular, ZeroMQ allows all nodes to exchange information directly, by running a decentralized ZeroMQ message broker on each node. The only mechanism of ZeroMQ that works centralized, is the discovery of nodes to communicate with. For this purpose, a *redis data structure store* [27] that hosts identification information for all ZeroMQ brokers has to be installed on the controller node. Because redis is an in-memory data store, it is highly performant and accordingly scalable.

To set up the ZeroMQ broker on a compute node, the broker binaries have to be installed, the nova configuration changed to use ZeroMQ, and the correct ports and node names for discovery and communication configured. No other changes are necessary, because all RPCs needed for communication are agnostic to the underlying messaging protocol and ZeroMQ is directly integrated into OpenStack.

Therefore, this solution allows to achieve decentralization of the framework in a highly convenient and OpenStack compatible manner.

5.3.2 Asynchronization

To achieve independence of compute nodes, the nova-fairness services are not synchronized among nodes. Moreover, messages between nodes are sent by RPC casts, as these, in contrast to calls, do not invoke a response. Accordingly, information about other nodes (VM scalars, VM endowments, and NRI) reach compute nodes at different times.

5.3.3 Collecting the Cloud Supply

As discussed in Section 3.3 the overall cloud resource supply is required for normalization and is the sum of NRI of compute nodes in the cloud. The NRI depend on the hardware configuration of nodes and are, therefore, rather stable. Accordingly, it is sufficient, when a node stores the NRI of every node and overwrites it, when new NRI of that node (which will likely be the same) is received.

To distribute the NRI in the cloud, every node that runs the nova-fairness service announces its NRI to all other nodes. This announcement is done by a periodic task with a default interval of 10 seconds. In particular, this task sends the local NRI to all compute nodes which have not yet received the node's NRI in the current period. Vice-versa, a node replies with its NRI upon receipt of the NRI of another node.

5.3.4 Collecting Scalars and Endowments

Unlike the NRI, VM endowments and, particularly, VM heaviness scalars change constantly and are associated with a certain period of time. Accordingly, this information needs to be synchronized, when it reaches a node. To do so, every compute node n_a maintains a FIFO queue for every other node n_b in which scalars received from n_b are queued. When all of these queues on n_a contain at least one element, n_a dequeues the first element from every queue. A single compute node could however stall this whole process if it sends its scalars with a delay or fails to send them at all. This problem will be addressed in future work. The dequeued scalars are aggregated to the heaviness of users by adding up all scalars of a user's VMs. The user heaviness together with the VM's heaviness is then used to set priorities of each user's VMs hosted on n_a. By adding the VM-specific heaviness to the user heaviness instead of only using the user heaviness to set priorities, the weight of the individual VM is emphasized.

To ensure that scalars are only sent to compute nodes which are currently reachable and running the fairness service, a list of available nodes is first requested from the nova-conductor service. The nova-conductor queries the SQL database located on the controller which contains all status information about services currently running.

5.4 Calculating and applying priorities

The framework allocates node resources by weights. Weights allow to achieve a proportional and flexible allocation of resources. For example, when a node hosts VMs vm_1 with a CPU weight of 1 and vm_2 with a CPU weight 2, vm_2 will receive twice the amount of CPU time from the node, given that both VMs' resource demands are not satisfied. However, if vm_2 does not attempt to utilize all CPU time, which it is entitled to, the remainder can be utilized by vm_1 (if there are no other VMs on the node). In particular, this highlights the most important feature of weights compared to hard limits. Weights always allow to use the full resource capacity, *i.e.*, even if a VM has a small weight, it is allowed to utilize the entire resource, when no other VM requests it.

The framework implements a very basic algorithm to map user heaviness to weights in a generic fashion. Defining more elaborated mappings are future work.

CPU shares are prioritized based on the ratios of shares. A CPU-share distribution of 100 shares for CPU 1 and 200 shares for CPU 2 has the same effect as 512 shares for CPU 1 and 1024 shares for CPU 2. The framework maps priorities to CPU shares in the range [1,100].

RAM soft-limits are assigned from 10 MiB to the maximum amount of RAM available to the VM.

Disk weights range from 100 to 1000 with 100 representing the lowest and 1000 the highest weight. These limits are defined by libvirt and have to be adhered to.

Network priorities are translated into HTB qdisc classes ranging from 1:0 to 1:98 with 1:0 being the lowest and 1:98 the highest priority. The range is defined by tc and has to be adhered to.

All priorities, except network, can be set directly by libvirt. Although libvirt allows to change network access of VMs, this can only be done by hard limits. However, this is not desirable, as hard limits potentially waste desired resources. Therefore, the framework regulates network access by calling the corresponding command line interfaces (CLIs) for the unix application "tc" to assign weights to the VMs' traffic.

6. EVALUATION

For the evaluation, the framework was integrated into an OpenStack environment which was set up according to the official OpenStack installation guide for Ubuntu 14.04 [22]. Accordingly, the OpenStack installation includes the following nodes:

1. The controller node which runs OpenStack management components for the compute service (nova) as well as the networking service (nova-network). It also hosts the SQL server where OpenStack stores all operation-relevant data and a redis server which is used for discovery by the message queue.

2. The individual compute nodes run OpenStack compute services (nova) which communicate with the hypervisor. KVM, the default hypervisor is used to manage the VMs running on the nodes.

The framework was evaluated with respect to the following criteria:

1. Performance overhead of the framework

2. Impact on VM performance in non-congested clouds

3. Performance isolation for VMs in congested clouds

6.1 Impact on the OpenStack environment

The performance overhead was evaluated by measuring CPU times for OpenStack services with and without the nova-fairness service running. A compute node in the framework environment runs the *nova-compute*, *nova-network* and *nova-api-metadata* services as well as VMs which can all be directly analyzed through their corresponding processes.

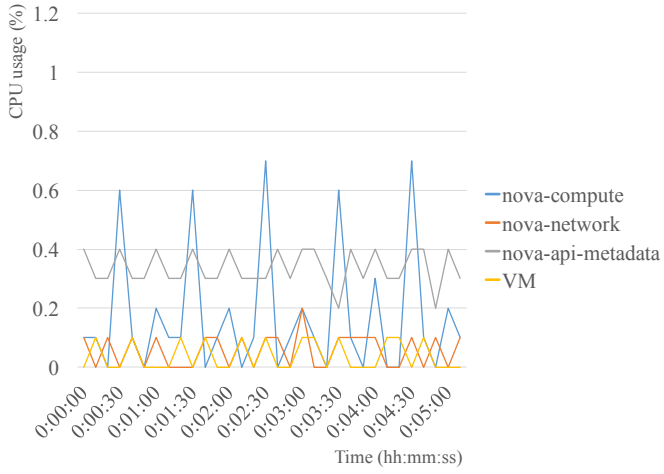

Figure 1: CPU usage of OpenStack services without nova-fairness running

The overall CPU usage of the idle services is very low with CPU usages of 0.1% to 0.8% as can be seen in Figure 1. The individual spikes in CPU usage for the nova-compute service can be directly attributed to the periodic tasks which have a default interval of 60 seconds.

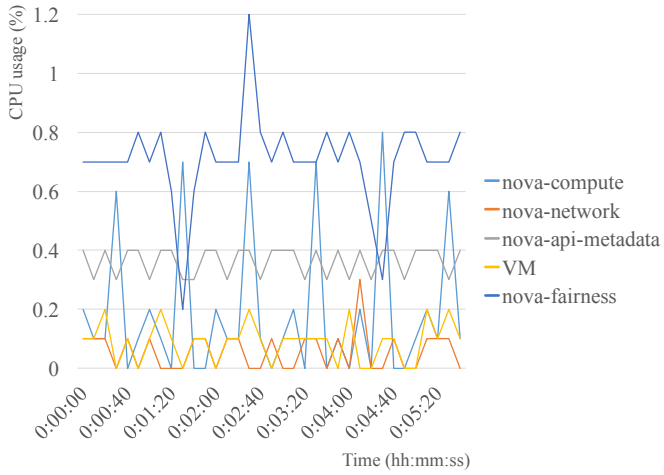

Figure 2: CPU usage of OpenStack services with nova-fairness running

Figure 2 shows the nova-fairness service with a modest CPU usage of about 0.7% to 1.2%. It is consuming more CPU time compared to the other services but does not impact the overall CPU usage of the OpenStack environment significantly. This shows the isolation of the framework from the rest of the OpenStack environment: The framework is a standalone service which only communicates with the nova-conductor service on the controller node and passively inspects VM resource consumption.

6.2 VM performance without contention

It is important that the framework's resource needs do not impede the performance of VMs. The framework should only limit the VMs through the soft-limits it sets, not through excessive resource consumption from running the service. To test this behavior, a single VM has been scheduled on a compute node running the nova-fairness service. The node is equipped with a dual-core CPU on which a maximal CPU load is provoked by the VM. To achieve this, the VM is configured with two VCPUS and runs the "stress" workload generator [1]. This workload generator can be used for CPU, memory and I/O stress tests by configuring a specific amount of workers that put the specified load on the hardware. Different intervals for the RUI collection task are tested to analyze whether a difference in frequency for RUI collection changes the performance impact of the service.

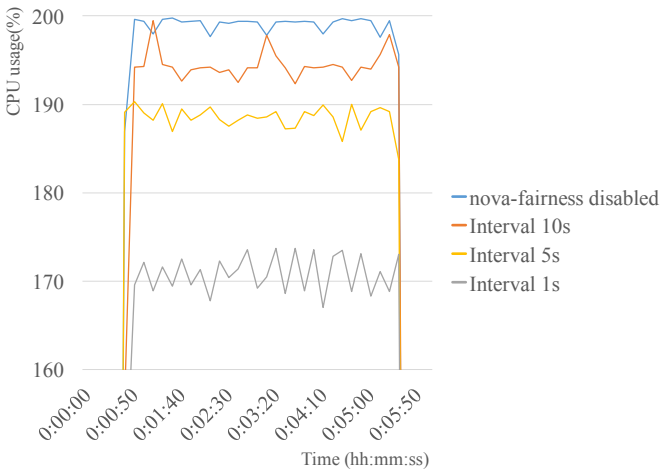

Figure 3: CPU usage of a single VM under maximum load with nova-fairness disabled or enabled with different RUI collection intervals

Running the framework with the default RUI collection interval of 10 seconds results in only a minor impact on VM performance compared to running the stress workload with the nova-fairness service disabled.

A notable impact on performance occurs, as soon as the interval for the RUI collection task is decreased below 5 seconds as can be seen in Figure 3.

6.3 Analyzing framework runtime

Figure 4 shows the average amount of time needed for a single run of the framework for 1 to 4 VMs. The runtimes have been averaged over a period of 5 Minutes, running the framework every 10 Seconds.

The bar for each run is divided into the different framework-steps for each VM. The initial step of collecting RUI takes longest for the first VM but is done much faster for subsequent VMs as the first RUI collection run also includes setting up data structures and requesting the list of instances from the nova-conductor service.

The total runtime increases linearly with the amount of

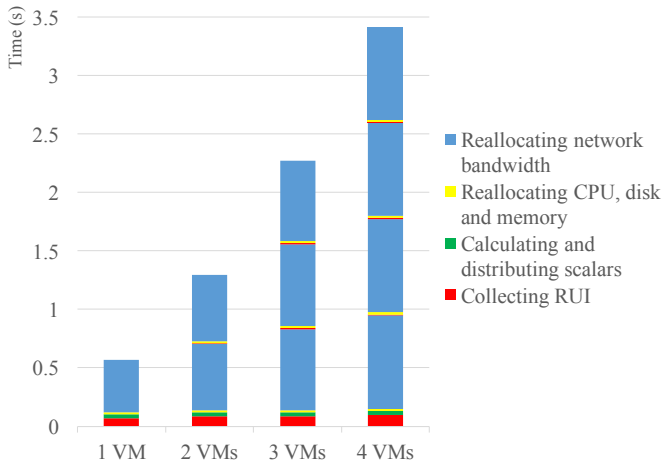

Figure 4: Average runtime for the framework with 1,2,3 or 4 VMs

VMs as each VM requires a complete run of the framework. Setting network bandwidth priorities takes longest. This runtime impact can be identified in Figure 4 with the blue bars which represent the time needed for network bandwidth prioritization.

The main reason for this increased time consumption is the need for multiple CLI calls to set the priorities. As outlined in Section 5.4, the unix application "tc" is used to apply network priorities which needs to be called multiple times.

6.3.1 *Optimal Interval Length*

Based on the runtime of the framework, an informed decision towards an optimal interval length can be made to minimize the impact of the framework and thereafter increase the performance of the compute node. If the framework runtime is higher than the interval length, a delay is produced that adds to the delay discussed in Section 5.3.2.

This analysis together with the results from Section 6.2 lead to the conclusion that a RUI collection interval of less than 5 seconds is not advisable.

The default interval-length of 10 seconds has been chosen as a result. This interval-length allows for delays caused by network latency as well as framework runtime while still providing enough time between framework runs to keep the framework impact on VM performance to a minimum.

6.4 User performance isolation

The ability of the framework to reallocate resources and create a fair distribution of resources among users is paramount.

Different setups have been considered to cover possible situations where resource contention might occur. Single node as well as multi-node scenarios with multiple users and single as well as multi-resource contention have been prepared to evaluate the framework. The default interval of 10 seconds was chosen for all evaluations.

CPU and disk usage have been chosen as evaluation-resources. The choice for these two resources is based on the ease of monitoring. Both nodes used for the evaluation are equipped with a dual-core CPU, 4096 MiB RAM, a 150 MiB/s harddrive as well as a 1 Gbit/s network connection.

Memory and network usage have not been used for these evaluations to simplify the setups and provide easily readable evaluation outputs.

The greediness metric was used as the heaviness-metric for the evaluation [25].

6.4.1 *Single node, single resource*

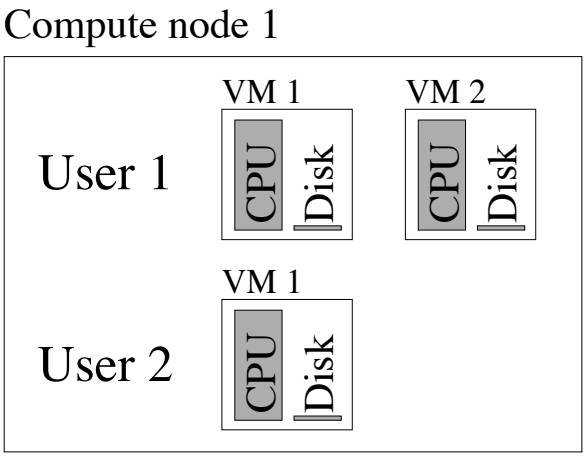

Figure 5: Evaluation setup with 1 node and 2 users competing for a single resource

In this evaluation setup, two users are running VMs on a single compute node as shown in Figure 5. They are both creating a maximal CPU load on all their VMs. All other resources are idle as can be seen in Figure 5 with the second resource "disk".

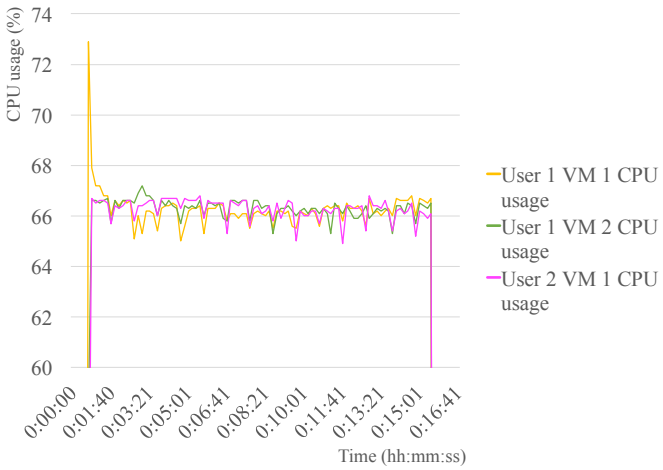

Figure 6: CPU usage for 3 VMs on a single node competing for a single resource without the nova-fairness service running

Running multiple VMs on a single node without the nova-fairness service results in identical resource consumptions for all VMs as can be seen in Figure 6. The fact that user 2 only runs 1 VM compared to user 1 with 2 VMs plays no role in the amount of CPU time available to the VMs, as

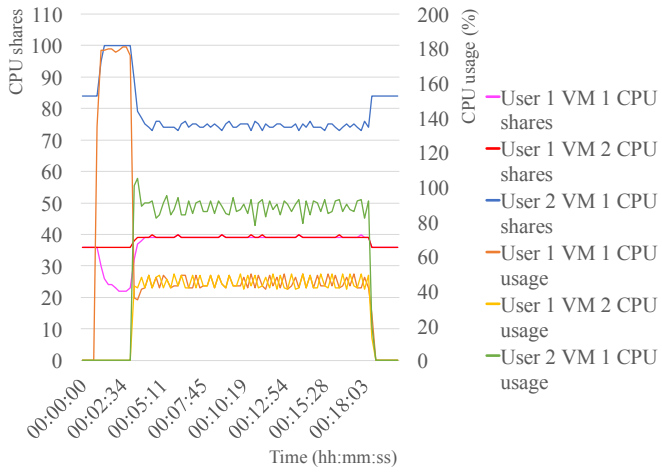

Figure 7: CPU shares and CPU usage for 3 VMs on a single node competing for a single resource

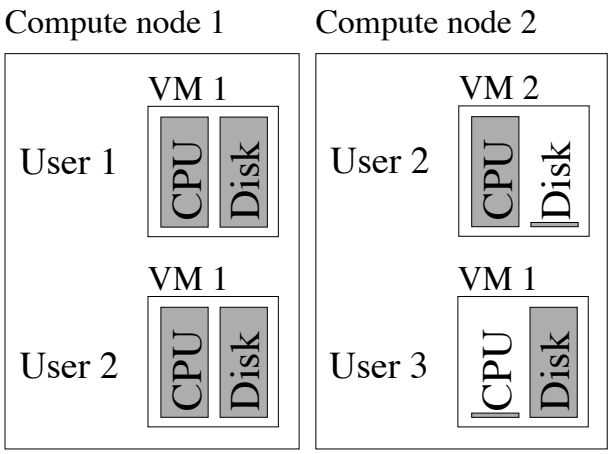

Figure 8: Evaluation setup with 2 nodes and 3 users competing for multiple resources

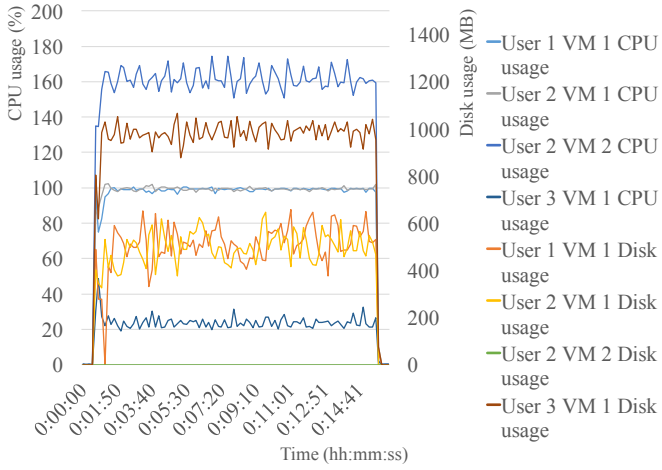

Figure 9: CPU and disk usage for 4 VMs on multiple nodes competing for multiple resources without the nova-fairness service running.

could be expected without the framework prioritizing the VMs of user 2.

Figure 7 shows the successful performance isolation caused by the framework for user 2: Since user 2 only runs 1 VM compared to the 2 VMs of user 1, she gets about twice as much CPU shares as user 1 and consequently also shows higher CPU usage throughout the whole runtime.

The initial spike in CPU usage for VM 1 of user 1 in Figure 7 serves to demonstrate the flexibility of the framework in regards to setting CPU shares as well as the need for contention on the node in order for the CPU shares, which are soft-limits, to take effect. Since only VM 1 is using CPU time and both VM 2 of user 1 and VM 1 of user 2 are idle, the node is not under contention in regards to CPU usage. This allows VM 1 of user 1 to consume a large amount of CPU time despite its CPU shares which are clearly sinking to favor user 2 whose CPU shares are rising throughout the CPU spike.

6.4.2 *Multiple nodes, multiple resources*

The effects of the framework on multiple nodes are much more critical than single-node results since clouds generally consist of many compute nodes. This is also true for multiple resources.

Based on this, two compute nodes have been chosen for the second evaluation setup. Users are competing for both CPU time as well as disk I/O to evaluate the ability of the framework to consider multiple resources for resource allocations.

Without the framework, VMs on compute node 2 can use all the resources they want since no resource contention exists as can be seen in Figure 9 for VMs 1 and 2 of users 3 and 2. This situation is comparable to the resource allocations for the same setup with the framework in place. However on compute node 1, the situation is not that simple: Since two VMs are trying to use CPU time as well as disk I/O, they both get the same amount of CPU time and are limited to the same disk speeds as can be seen in Figure 9.

With the framework in place, the resource contention on compute node 1 is addressed in a fair way: User 1 gets more CPU shares than user 2 on compute node 1 because she only runs a single VM on a single compute node compared to user 2 who runs 2 VMs on different compute nodes.

In Figure 11, the easiest observation is the big amount of disk usage by VM 1 of user 3. This is a result of a lack of contention in regards to disk usage on compute node 2.

Disk usage for VM 1 of user 1 as well as the disk weights applied to the same VM are much higher than the disk usage of VM 1 of user 2 since user 2 is also running a second VM on compute node 2, which reduces her overall priority. This shows the clear difference in disk speeds available to both VMs compared to the situation without the framework in place as shown in Figure 9.

6.5 Findings

All evaluation setups show the successful use of soft-limits to ensure that no unnecessary resource limitations occur if the compute node in question is not under contention. The evaluations also show that the framework functions properly

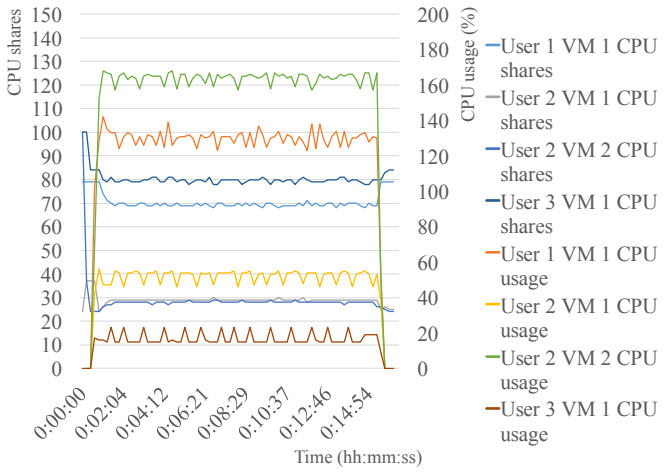

Figure 10: CPU shares and CPU usage for 4 VMs on multiple nodes competing for multiple resources

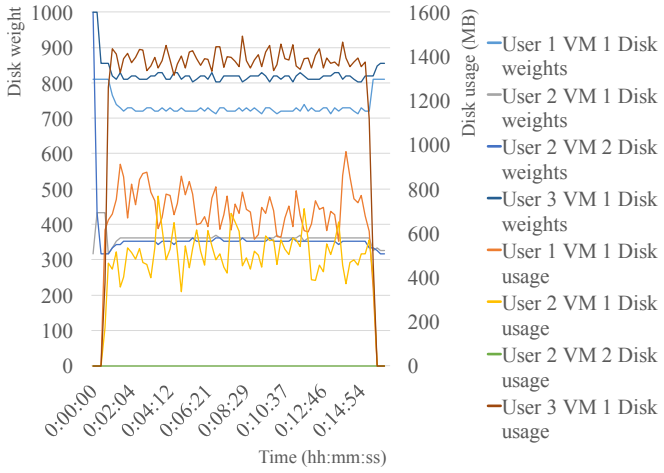

Figure 11: Disk weights and disk usage for 4 VMs on multiple nodes competing for multiple resources

over multiple nodes, successfully exchanging scalars and applying the correct priorities based on user's resource consumptions.

All evaluations have been conducted over a timespan of 15 minutes, successfully showing that the framework can isolate a user's performance without any degradation over that timeframe.

The influence of multiple resources on the user's priority can be directly observed in section 6.4.2 where the second resource helps to differentiate the priorities and clearly set them apart in terms of resources they are allowed to consume.

7. SUMMARY AND FUTURE WORK

This framework uses the approach of enforcing fairness through runtime VM prioritization instead of using scheduling. This allows the OpenStack environment to react to resource contentions and reduce the need of VM evacuation or rescheduling.

Implementing multi-resource fairness during VM runtime in OpenStack requires a solid framework that allows seamless communications between all compute nodes involved. The multitude of resources involved and the lack of knowledge in regards to the utility functions of the different cloud users asks for a simple solution to generate a scalar that allows easy comparison between the user's resource consumptions.

The framework presented in this paper allows for a flexible implementation of different approaches towards calculating such a heaviness scalar to allow future research in regards to evaluating efficiencies of different heaviness metrics.

Performance is an important consideration for the whole framework which lead to many design decisions to further increase speed while reducing computing as well as communication overhead. There are however still performance bottlenecks present in the framework, namely the impact of network bandwidth prioritization with an increasing amount of VMs.

An additional venue for future research presents itself with the search for an effective translation of scalar aggregates to priorities as discussed in Section 5.3.4. Throughout the evaluation of the framework, it became clear that changes in resource usage only result in small changes in the heaviness aggregates which makes it harder to clearly calculate a priority that reflects the initial change in resource consumption. This could be addressed with smart applications of factors to amplify certain changes in aggregates more than others, to further increase the significance of the heaviness-scalars calculated by the chosen metric.

8. REFERENCES

[1] Amos Waterland. stress. `http://people.seas.harvard.edu/~apw/stress/`, 2014. Online; accessed February 10th, 2016.

[2] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. Hierarchical Scheduling for Diverse Datacenter Workloads. *4th Annual Symposium on Cloud Computing*, SOCC '13, pp 1–15, Santa Clara, CA, USA, October 2013.

[3] Thomas Bonald and James Roberts. Enhanced Cluster Computing Performance through Proportional Fairness. *Performance Evaluation*, 79:134–145, April 2014.

[4] Thomas Bonald and James Roberts. Multi-Resource Fairness: Objectives, Algorithms and Performance. *2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, pp 31–42, Portland, Oregon, USA, 2015.

[5] David Breitgand, Zvi Dubitzky, Amir Epstein, Alex Glikson, and Inbar Shapira. SLA-aware Resource Over-commit in an IaaS Cloud. *8th International Conference on Network and Service Management (CNSM) and 2012 Workshop on Systems Virtualization Management (SVM)*, pp 73–81, Las Vegas, NV, USA, October 2012.

[6] Citrix Systems, Inc. AMQP and Nova. `http://docs.openstack.org/developer/nova/rpc.html`, 2010. Online; accessed February 5th, 2016.

[7] D. Clark, J. Wroclawski, K. Sollins, and R. Braden. Tussle in Cyberspace: Defining Tomorrow's Internet.

IEEE/ACM Transactions on Networking, 13(3):462–475, June 2005.

[8] Danny Dolev, Dror G. Feitelson, Joseph Y. Halpern, Raz Kupferman, and Nathan Linial. No Justified Complaints: On Fair Sharing of Multiple Resources. *3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pp 68–75, Cambridge, MA, USA, January 2012.

[9] Frank Kelly et al. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. *The Journal of the Operational Research Society*, 49(3):237–252, March 1998.

[10] Yoav Etsion, Tal Ben-Nun, and Dror G. Feitelson. A Global Scheduling Framework for Virtualization Environments. *2009 IEEE International Symposium on Parallel Distributed Processing*, IPDPS 2009, pp 1–8, Rome, Italy, May 2009.

[11] Eric Friedman, Ali Ghodsi, and Christos-Alexandros Psomas. Strategyproof Allocation of Discrete Jobs on Multiple Machines. *15th ACM Conference on Economics and Computation*, EC '14, pp 529–546, Palo Alto, CA, USA, June 2014.

[12] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Heterogeneous Resources in Datacenters. Technical Report UCB/EECS-2010-55, EECS Department, University of California, Berkeley, CA, USA, May 2010.

[13] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource Packing for Cluster Schedulers. *2014 ACM Conference on Special Interest Group on Data Communications*, SIGCOMM '14, pp 455–466, Chicago, Illinois, USA, August 2014.

[14] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource Packing for Cluster Schedulers. *SIGCOMM Computer Communication Review*, 44(4):455–466, August 2014.

[15] Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. Navigating Heterogeneous Processors with Market Mechanisms. *IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, pp 95–106, Shenzhen, China, February 2013.

[16] Avital Gutman and Noam Nisan. Fair Allocation without Trade. *11th International Conference on Autonomous Agents and Multiagent Systems*, Vol. 2 of *AAMAS '12*, pp 719–728, Valencia, Spain, June 2012.

[17] Dalibor Klusáček and Hana Rudová. Multi-resource Aware Fairsharing for Heterogeneous Systems. In: Walfredo Cirne and Narayan Desai (Editors), *18th International Workshop on Job Scheduling Strategies for Parallel Processing, Revised Selected Papers*, JSSPP 2014, pp 53–69. Springer International Publishing, May 2015.

[18] Dalibor Klusáček, Hana Rudová, and Michal Jaroš. Multi Resource Fairness: Problems and Challenges. In: Narayan Desai and Walfredo Cirne (Editors), *Job Scheduling Strategies for Parallel Processing*, Vol. 8429 of *Lecture Notes in Computer Science*, pp 81–95. Springer, Berlin/Heidelberg, Germany, 2014.

[19] Haikun Liu and Bingsheng He. Reciprocal Resource Fairness: Towards Cooperative Multiple-Resource Fair Sharing in IaaS Clouds. *2014 IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pp 970–981, New Orleans, LA, USA, November 2014.

[20] Xi Liu, Xiaolu Zhang, Xuejie Zhang, and Weidong Li. Dynamic Fair Division of Multiple Resources with Satiable Agents in Cloud Computing Systems. *IEEE 5th International Conference on Big Data and Cloud Computing*, BDCloud '15, pp 131–136, Dalian, China, August 2015.

[21] OpenStack Foundation. OpenStack Wiki: ZeroMQ. `https://wiki.openstack.org/wiki/ZeroMQ`, 2014. Online; accessed February 5th, 2016.

[22] OpenStack Foundation. OpenStack Installation Guide for Ubuntu 14.04. `http://docs.openstack.org/juno/install-guide/install/apt/content/index.html`, 2015. Online; accessed February 4th, 2016.

[23] OpenStack Foundation. Services, Managers and Drivers. `http://docs.openstack.org/developer/nova/services.html`, 2016. Online; accessed February 5th, 2016.

[24] David C. Parkes, Ariel D. Procaccia, and Nisarg Shah. Beyond Dominant Resource Fairness: Extensions, Limitations, and Indivisibilities. *13th ACM Conference on Electronic Commerce*, EC 2012, pp 808–825, Valencia, Spain, June 2012.

[25] Patrick Poullie and Burkhard Stiller. Cloud Flat Rates Enabled via Fair Multi-resource Consumption. Technical Report IFI-2015.03, `https://files.ifi.uzh.ch/CSG/staff/poullie/extern/publications/IFI-2015.03.pdf`, Universität Zürich, Zurich, Switzerland, October 2015.

[26] Red Hat, Inc. Libvirt: Internal Drivers. `https://libvirt.org/drivers.html`, 2016. Online; accessed February 5th, 2016.

[27] Redis Labs. Redis. `http://redis.io/`, 2015. Online; accessed February 5th, 2016.

[28] Standard Performance Evaluation Corporation. Benchmarks. `http://spec.org/benchmarks.html`, 2015. Online; accessed February 20th, 2016.

[29] Chandra Thimmannagari. *CPU Design: Answers to Frequently Asked Questions*. Springer, Berlin/Heidelberg Germany, 2005.

[30] G. Wei, A. Vasilakos, Y. Zheng, and N. Xiong. A Game-theoretic Method of Fair Resource Allocation for Cloud Computing Services. *The Journal of Supercomputing*, 54(2):252–269, November 2010.

[31] Yoel Zeldes and Dror G. Feitelson. On-line Fair Allocations Based on Bottlenecks and Global Priorities. *4th ACM/SPEC International Conference on Performance Engineering*, ICPE 2013, pp 229–240, Prague, Czech Republic, April 2013.

[32] Qinyun Zhu and Jae C. Oh. An Approach to Dominant Resource Fairness in Distributed Environment. In: Moonis Ali, Young Sig Kwon, Chang-Hwan Lee, Juntae Kim, and Yongdai Kim (Editors), *Current Approaches in Applied Artificial Intelligence*, Vol. 9101 of *Lecture Notes in Computer Science*, pp 141–150. Springer International Publishing, 2015.

Appendix B

Contents of the CD

The contents of the CD are structured as follows:

Abstract contains the abstract in English.

Bachelorarbeit.pdf contains the complete thesis report.

evaluation contains raw evaluation data for the framework evaluations outlined in [12, Section 6] stored as Microsoft Excel spreadsheets. It also contains spreadsheets for additional evaluation setups not presented in the whitepaper or report. *evaluation_helper* contains Python scripts which were used to gather the data used for the evaluations.

openstack_nova_with_fairness contains the complete customized OpenStack nova code with the nova-fairness service integrated.

shortcuts contains symbolic links to the nova-fairness Python code.

ubuntu_packages contains the Ubuntu packages for the nova-fairness service as well as the ZeroMQ broker outlined in Sections 5.2, 5.3 and 3.1.

Zusfsg contains the abstract in German.