University of Zurich[UZH]

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

Investigation of Resource Reallocation Capabilities of KVM and OpenStack

Andreas Gruhler
Zurich, Switzerland
Student ID: 12-708-038

Supervisor: Patrick Gwydion Poullie, Dr. Thomas Bocek
Date of Submission: August 16, 2015

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

ifi

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

Abstract

Overcommitment beschreibt die Zuweisung von mehr physikalischen Ressourcen (PRs) zu einer Gruppe virtueller Maschinen (VMs) als eigentlich verfügbar wären. Dieses Grundkonzept der Virtualisierung wird von vielen Anbietern genutzt, um die Betriebskosten zu minimieren [14]. Der Hypervisor muss dabei Ressourcen nach der erstmaligen Zuteilung so umverteilen, dass er den sich ändernden Ressourcenanforderungen der VMs gerecht wird. Aktuelle Methoden zur Rückgewinnung und Umverteilung von Ressourcen wie RAM ballooning und page sharing erleichtern diese Aufgabe [10]. Die Orchestration Ebene — welche den Start von VMs auf verschiedenen Hypervisern koordiniert — bietet nicht so viele Möglichkeiten zur Umverteilung physikalischer Ressourcen wie ein Hypervisor [14]. Physikalische Ressourcen können deshalb durch Orchestration Software wie z.B. OpenStack nicht zufriedenstellend kontrolliert werden. Diese Arbeit gibt einen Überblick über verschiedene Techniken zur Umverteilung von RAM, CPU-Zeit, Disk I/O und Netzwerkbandbreite. Es werden die Fähigkeiten von KVM und OpenStack zur Umverteilung von Ressourcen analysiert. Während der Arbeit wurde eine OpenStack Erweiterung entwickelt, welche die Umverteilung physikalischer Resourcen auf der Orchestration Ebene erleichtert.

Overcommitment, the assignment of more physical resources (PRs) than actually available to a group of virtual machines (VMs), is a core concept of virtualization used by many providers to minimize cost of ownership [14]. This requires the hypervisor to reallocate resources after initial assignment such that the changing resource demands of the VMs can be met. Current resource reclamation and reallocation methods like memory ballooning and page sharing facilitate this task [10]. The orchestration layer—which is responsible for scheduling the VMs on different hypervisors—does not provide as many options to reallocate PRs as the hypervisors [14]. This leads to a lack of control of PRs in orchestration layer software like OpenStack. This thesis reviews techniques used to reallocate memory, CPU cycles, disk I/O and network bandwidth and investigates the resource reallocation capabilites of KVM and OpenStack. During this thesis, an OpenStack extension has been developed to facilitate the task of PR reallocation on the orchestration layer.

Acknowledgments

First of all I want to thank my supervisors Patrick Gwydion Poullie and Dr. Thomas Bocek for giving me the opportunity to write a thesis in the very interesting and industry relevant field of virtualization. I am very thankful for their ongoing huge support in technical questions and for guiding me through the process of writing this thesis. I also want to thank my fellow student Simon Balkau, which was a very great help to set up a working OpenStack/KVM development environment. Thanks goes to Guilherme Sperb Machado for providing the infrastructure and giving me useful tips on how to use it efficiently. Last but not least, I would like to give thanks to my family and friends, who helped proofreading the thesis, despite its sometimes very technical character.

Contents

Chapter 1

Introduction

1.1 Motivation

Overcommitment of physical resources (PRs) is a common paradigm in todays cloud infrastructures. Because virtual machines (VMs) not always use the maximum of assigned resources simultaneously, the sum of guest resource limits can be higher than the total amount of resources actually available on a host [10]. To achieve this, resources have to be reallocated depending on the resource needs of the VMs [21].

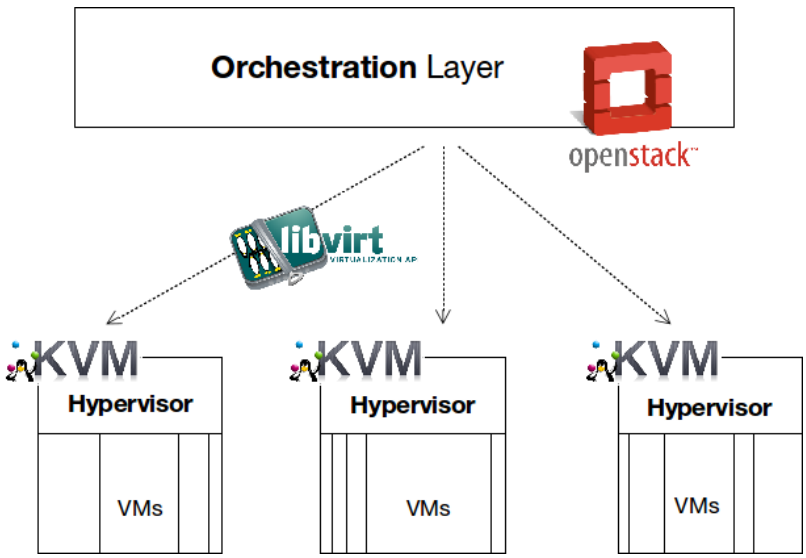


Figure 1.1: An illustration of an OpenStack/KVM cloud infrastructure. The libvirt API not only provides an interface for communication between the orchestration layer and the hypervisor, but also enables to monitor and control guests. [26]

The orchestration layer represented by current IaaS-Software (e.g. OpenStack) and its supported hypervisors (e.g. KVM, the Kernel-based Virtual Machine) allow to reallocate physical cloud resources between VMs [14]. The hypervisor provisions the resources for the VMs. The orchestration layer is responsible for the scheduling of VMs and their

deployment on a physical machine (PM). It provides unified control of multiple hypervisors [14]. In comparison to orchestration layer software, current hypervisors offer better control of PRs [14]. The overall goal of the thesis is to find, document and if not yet done implement methods used to reallocate resources of VMs running on KVM and using the open-source IaaS-Software OpenStack. KVM is a bare-metal hypervisor for the linux operating system with full virtualization support, on which guests with an original, unmodified kernel can be run [32].

1.2 Thesis Outline

The thesis is structured as follows.

Chapter 2 surveys state of the art reallocation methods for physical cloud resources. It is examined if and how KVM and OpenStack make use of these methods. Tools which support the implementation of such a method in a specific application such as KVM or OpenStack are inspected as well.

Chapter 3 presents ways to reallocate resources with the libvirt API. This commonly used API provides a comprehensive interface for KVM.

Chapter 4 discusses the current possibilities to reallocate PRs through the OpenStack API. During this thesis, an OpenStack extension has been developed to control the PRs of VMs. The decisions made in the process of writing the API extension are documented in this chapter.

Appendix A contains benchmark results from experiments with RAM bandwidth. The python code used to run the benchmarks can be found on the enclosed CD.

Appendix B holds results of disk bandwidth tests.

Appendix C documents the OpenStack API extension and its dependent parts written as part of this thesis.

Chapter 2

Related Work

2.1 Reallocation Methods

This section surveys methods suitable to reallocate PRs. The chapter concludes with a comparison of PR reallocation costs. Some of the methods are exclusively applied in virtualized environments (marked with *) whereas others are also used in the absence of virtualization.

The methods discussed are:

1. * Memory ballooning
2. * Hypervisor-/guest swapping (paging)
3. * Processor pinning
4. Page sharing/merging
5. CPU- & memory hotplug

2.1.1 Memory Ballooning

The concept of memory ballooning is well presented in the vSphere whitepaper of VMware [10]. Guests can be assigned more memory as long as there is enough RAM available on the host. Afterwards, memory has to be reallocated. In a fully virtualized environment, the guest is abstracted through the virtualization layer which translates resource requests [32]. Thus an isolated guest cannot know when the host is running out of memory.

The balloon driver establishes the required communication channel between the hypervisor and the VM. In case of host memory shortcomings, the balloon driver allocates as much memory as requested by the hypervisor (requested size and actual allocation may vary slightly depending on implementation and machine states). The host communicates the

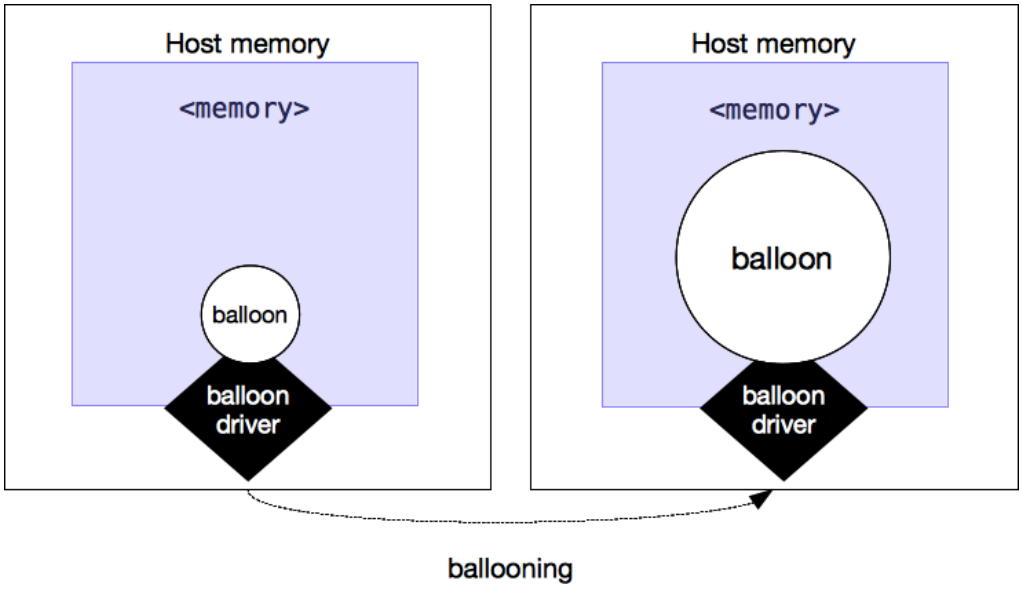


Figure 2.1: Illustration of the memory ballooning process. Host and guest communicate through the balloon driver (the black box). The violet area illustrates the amount of memory allocated to the guest at boot time. The guest can inflate the balloon up to the <memory> memory limit (labeled <memory>, because KVM makes use of this name in the VM configuration). The memory areas inside the balloon are locked for the guest and given back to the hypervisor. [10]

balloon size through the balloon driver. Memory inside the so called balloon cannot be accessed by the guest any more. The hypervisor reclaims this memory area. In the VMware implementation of the driver, the guest informs itself periodically if the host lacks any memory [10] whereas in the KVM implementation the host actively sends the memory reclamation request [11]. Choosing an appropriate dispensable memory area is up to the guest, which is the major advantage of memory ballooning. As long as the balloon size does not exceed the size of dispensable guest memory, ballooning has no impact on VM performance. Otherwise the guest moves memory pages to disk, which is called guest paging or swapping. In contrast to VMware's implementation of the balloon driver, the linux *virtio_balloon* driver will never trigger guest paging and only inflate the balloon as much as possible [11]. This way, guest memory claims (e.g. reserved memory) have a higher priority over memory reclamation requests of the host. In situations where host performance is more important than guest performance, the VMware implementation seems to be the right choice. It is unclear, if guest paging is always preferred in comparison to hypervisor paging. If the host moves memory pages of the VM to disk, this may not be a good choice. It would be better, if the host only pages memory areas not used by any VM. The paging of memory areas used by VMs may be better controlled by the VMs itself, because the host cannot know which guest pages are the most dispensable. In both cases (hypervisor- and guest swapping), paging affects the performance of the VM in a negative way.

2.1.2 Page Sharing/Merging

The idea behind page sharing, as presented in the VMware white paper [10], is to join memory pages (one memory page typically has a size of 4KiB) with identical contents.

It is not as much a method to reallocate memory, but rather a method to use memory efficiently. This technique is especially efficient if the VMs are running the same operating system or applications. The virtual memory is periodically scanned for identical pages. To reduce the scanning overhead, the memory pages are hashed or stored in a tree. VMware uses a hash table [10] whereas the linux kernel uses a red-black tree (ordered & balanced) [2]. If a guest writes to a shared page, this page gets copied (copy-on-write technique). If that page was shared between two guests, it is no longer shared.

Kernel Same-Page Merging (KSM)

Despite page sharing not being intended for virtualized environments only, the KVM is still one of its primary users [2]. This is due to the fact, that two VMs with identical operating systems can share a lot of memory. KSM can be configured on a linux system by configuring the files under `/sys/kernel/mm/ksm/`.

2.2 Reallocation Tools

The previous section gave an overview of the methods to use and reallocate memory efficiently, especially in virtualized environments. Theoretical methods have to be implemented in an utility or tool to be useful in practice. Common utilities, tools and configuration files used to reallocate resources on linux systems are:

1. * libvirt API
2. Linux control groups (cgroups)
3. Traffic control (`tc`)
4. `taskset`: retrieve or set a processes's CPU affinity
5. `cpuset`: confine processes to processor and memory node subsets
6. `nice`, `renice`: modify scheduling priority
7. `setquota`: set disk quotas
8. `ulimit`: limit the use of system-wide resources
9. `/etc/security/limits.conf`: configuration file for limiting system resources on a per user basis

Because the libvirt API is used especially in a virtualization context, it is marked by an asterix. Tools 3 to 9 do not assist one of the relevant reallocation methods listed in the previous Section 2.1. Nevertheless, they can be used to control resources in a traditional linux environment (mostly on a per user basis). Section 4.5 for example illustrates how `tc` can be used to control network bandwidth.

2.2.1 Linux Control Groups (cgroups)

Control groups are a mechanism of the linux kernel used to group processes. Subsystems (much like controllers) can be used to control resources of a cgroup. The subsystems relevant to manage RAM, CPU, disk space/speed and network bandwidth allocation are [12]:

- *memory*: Controls the amount of memory used by a cgroup.
- *cpusets*: Manages the assignment of CPUs to a cgroup.
- *cpu*: Used to set CPU access rate of a cgroup directly or relative to other tasks by using CPU shares.
- *blkio*: Allows to control block I/O speed by using weighting or throttling policies.
- *net_prio*: This subsystem allows to prioritize egressing network traffic on specific network interfaces.
- *net_cls*: The *net_cls* subsystem can be used to tag network packages. In combination with `tc` or `iptables` rate limits can be assigned to network packets of a cgroup.

While cgroups can be controlled by changing the files in the virtual file system manually it is recommended to use the *libcg* library [15].

2.2.2 libvirt

libvirt is a common API to control memory, computing time, disk space/speed and network bandwidth [16, 17, 18]. For VMs launched through libvirt, a cgroup hierarchy is created automatically for the mounted cgroup subsystems (refer to last section for an overview of the available cgroup subsystems). Most of the resources are controlled through this cgroup hierarchy. But the *net_cls* controller (see last section) is not used in libvirt [16]. Nevertheless, libvirt allows to set absolute rate limits for incoming and egressing network traffic on guest machines. Further usage examples can be found in Section 3.4. The OpenStack extension written as part of this thesis further allows to set soft limits for network bandwidth (see Section 4.5).

systemd and libvirt

systemd is a system and service manager for modern linux systems [31]. On systems running systemd, the KVM domain instances launched through libvirt are placed in a *slice*. A slice is a container for processes with specific resource limits. KVM domains with similar resource demands can be grouped in a slice. systemd uses cgroups to implement these limits but does not yet support all available subsystems [6].

2.3 Cost of Reallocation

In view of technological improvements, memory is clearly the more scarce resource than computing time. Researchers already predicted back in 1994 [34] that an increase in CPU clock speed may not be of any use when the rates of clock and memory speed improvement drift apart. While thinking about the cost of reallocating the resources compared to each other, the kernel documentation about the memory cgroup subsystem [24] makes a good point. If a program needs to allocate a lot of variables, the amount of RAM needed is constant. You may remove a CPU. The program will get slower, and eventually it will halt. The same principle applies to throttling disk I/O operations or network bandwidth. But given the characteristics of the program (i.e. the amount of variables needed at one point in time), you cannot run it with less memory unless you move memory pages to disk. Paging is much more expensive in comparison to a change in CPU time or network bandwidth. In a virtualization context the hypervisor has to take care it doesn't accidentally remove the wrong memory pages (the ones required to run the current program). Choosing the right pages takes time and a thorough knowledge of the memory content. Thus removing RAM is a difficult task which should be done carefully. Reallocating RAM is most expensive in terms of time to find the dispensable pages.

2.4 Overview of PR Reallocation

Figure 2.2 concludes this chapter to give an overview about existing tools and methods to reallocate PRs in virtualized environments. The libvirt API allows to reallocate all four PRs. The libvirt API makes use of cgroups and the `tc` traffic control utility. libvirt uses `tc` only to set hard limits (for more details about the methods in libvirt see Figure 3.1).

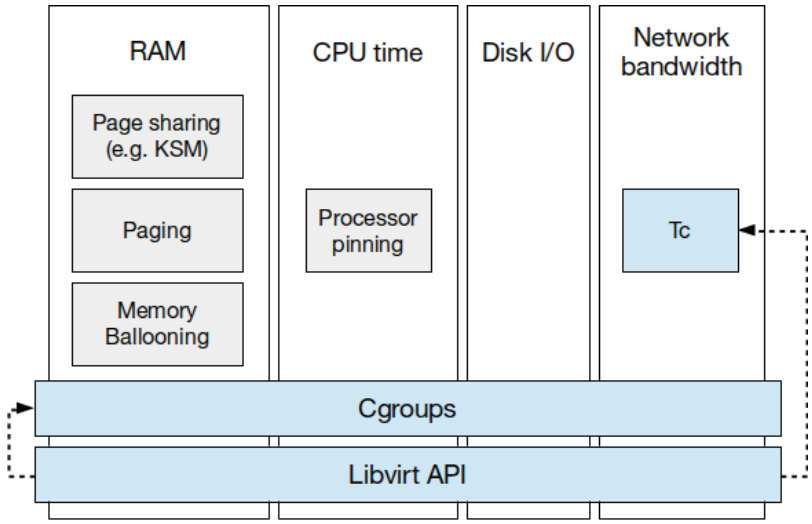


Figure 2.2: Overview of current tools and methods used to reallocate PR in virtualized environments. The dashed arrow represents a *"uses"* relation between two connected components. The grey boxes represent the theoretical methods. The blue boxes are important tools to reallocate resources.

Chapter 3

Controlling KVM Resources with libvirt

This chapter describes—by using examples for each resource—how resources of VMs running on the KVM can be reallocated using the libvirt API [33]. The tests have been conducted on a machine running Ubuntu 14.10. Figure 3.1 shows an overview of API methods and parameters discussed in this chapter, which are relevant to PR reallocation on KVM.

RAM	CPU	Disk I/O	Network bandwidth
setmem(currentMemory): set currentMemory **memtune**(soft_limit): set memory soft limit	**setvcpus**: add/remove current VCPUs **vcpupin**: pin VCPUs to physical host CPUs **schedinfo**(shares,quota): set CPU shares or CPU quota	**blkdeviotune**: control guest block devices; hard limits **blkiotune**(weight): control host block devices; soft and Hard limits	**domiftune**: set bandwidth hard limits for inbound and outbound traffic
<memory>: maximum memory allocation at boot time and maximum balloon size **<currentMemory>**: the current amount of RAM available on the guest	**<shares>**: soft limit for setting CPU priority **<quota>**: VCPU speed hard limit; determines memory bandwidth	**<weight>**: guest I/O weight on all host block devices	**<inbound>**: hard limit for ingress network traffic **<outbound>**: hard limit for egress network traffic

Figure 3.1: Overview of libvirt methods and parameters important for PR reallocation. Methods are yellow, parameters are green.

3.1 Memory (RAM)

3.1.1 Memory Ballooning with `setmem`

```
<domain>
  ...
  <memory unit='KiB'>1048576</memory>
  <currentMemory unit='KiB'>1048576</currentMemory>
  ...
</domain>
```

Listing 3.1: Memory configuration of VM1

```
<domain>
  ...
  <memory unit='KiB'>1048576</memory>
  <currentMemory unit='KiB'>524288</currentMemory>
  ...
</domain>
```

Listing 3.2: Memory configuration of VM2

As mentioned in Section 2.1.1, the host can reclaim memory from a VM (also called a *domain* in libvirt) using the ballooning technique. VM1 was started with 1024 MiB of RAM and VM2 with 512 MiB. Using the libvirt command line tool `virsh`, the RAM of VM1 can be restricted and given to VM2 as follows:

```
user@host:~$ virsh setmem VM1 524288
user@host:~$ virsh setmem VM2 1048576
```

By running `watch -n1 free` on both guests or `virsh dommemstat <DOMAIN>` or `virsh dominfo <DOMAIN>` on the host (where `<DOMAIN>` is the name of the domain), the almost instant change of free memory on VM1 and VM2 can be observed as expected. For obvious reasons, the balloon cannot be inflated by more than the `<memory>` hard limit, which is 1024 MiB in Listing 3.1 and 3.2. Setting `<memory>` in the configuration to the PM maximum memory is not possible. The guest cannot allocate that much memory at boot time and thus refuses to start, because the hypervisor needs some dedicated memory itself to run properly.

Ballooning is implemented on the guest by means of the *virtio_balloon* driver which gets inserted into the guest at startup. By using the libvirt command line tool `virt-install` to provision a new VM, the domain will have ballooning automatically enabled, if the guest OS supports this method [17]. The guest xml configuration file then contains the following snippet:

```
<devices>
  <memballoon model='virtio'/>
</devices>
```

The `virsh setmem` command inflates or deflates the balloon on running guests. The balloon cannot be inflated by more than `<memory>` (see Listing 3.1 and 3.2). If the linux system needs to allocate even more memory, a *physical memory hotplug* is needed [23].

Logical memory hotplug allows to put online the new memory blocks [23]. Then they are available for the running system. This increase of actual physical memory would make it possible to modify the balloon driver code in such a way, that it would be possible to increase the balloon size by even more than the `<memory>` ballooning limit [28]. Ballooning is of course only necessary, if the memory hotplug did not provide the necessary amount of memory required by the host. Augmenting the `<memory>` limit is reasonable if more physical memory is actually available. Memory hotplug is still under active development [23].

3.1.2 `memtune` Soft Limit

Whereas with `setmem` (see last section) it was possible to change the amount of current RAM of a VM, `virsh`'s `memtune` feature allows to specify soft limits for the guest. These limits only apply, if the host is running low on memory [17]. In my tests, the memory of a host with 4096 MiB of RAM and two guests with fifty percent of the hosts memory each was put under stress. The impact on VM1 with a soft limit of 1024 MiB and VM2 with a soft limit of 512 MiB is visible in Figure 3.2. Because of memory congestion, each VM is only allowed to use memory up to the defined soft limit.

```
user@host:~$ virsh memtune VM1 --soft-limit 1048576
user@host:~$ virsh memtune VM2 --soft-limit 524288
```

Listing 3.3: Example for setting a memory soft limit of 512 MiB on VM2 1024 MiB on VM1.

The proportional set size (PSS) is a measure used by the `smem` utility to report the current memory usage of libraries and applications [29]. It includes a proportional share of shared memory (see section 2.1.2 about page sharing).

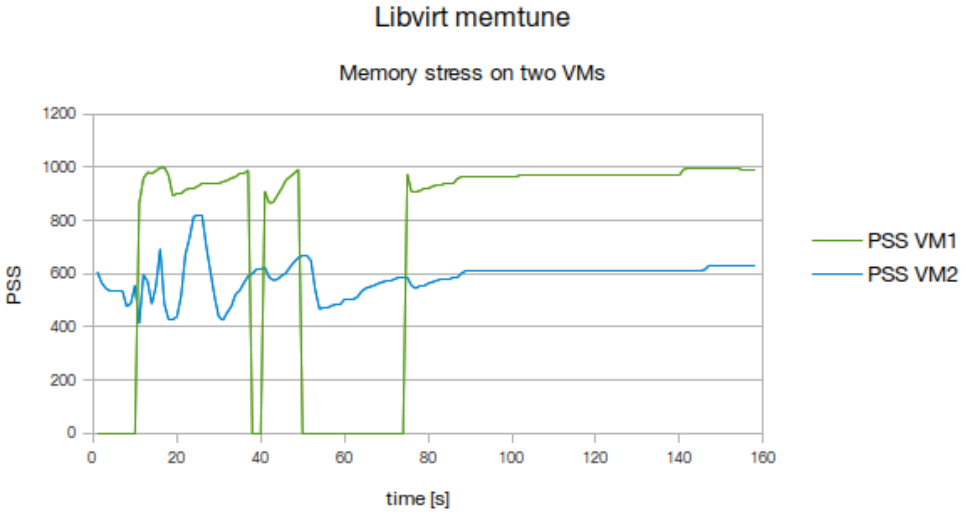


Figure 3.2: This graph shows the effect of memory soft limits in a situation of excessive memory usage. There was a memory stress test running on both VMs. VM1 has a memory soft limit of 1 GiB. VM2 has a memory soft limit of 512 MiB.

3.2 Computing Time (CPU)

Through the `virsh nodeinfo` command, the host's physical CPU information is displayed:

```
user@host:~$ virsh nodeinfo
CPU model:           x86_64
CPU(s):              4
CPU frequency:       808 MHz
CPU socket(s):       1
Core(s) per socket:  2
Thread(s) per core:  2
NUMA cell(s):        1
Memory size:         8086220 KiB
```

On the example host with 4 cores a sample VM is set up to boot with 4 VCPUs and restricted to a maximum of 16 VCPUs:

```
<domain>
  ...
  <vcpu placement='static' current='4'>16</vcpu>
  ...
</domain>
```

The 4 cores can be monitored with the `virsh vcpuinfo <DOMAIN>` (see Section 3.2.2) or the `virsh vcpucount <DOMAIN>` command:

```
user@host:~$ virsh vcpucount VM
maximum      config        16
maximum      live          16
current      config         4
current      live           4
```

Values marked with the `config` flag are specified in the xml configuration file of the domain. Values with the `live` flag describe the current state of the domain. These values can differ, if a dynamic adjustment on the running host (e.g. `virsh setvcpus`, see next section, or `virsh setmem`) has been made.

3.2.1 setvcpus

Adding more VCPUs is possible with `virsh setvcpus <DOMAIN><COUNT>`, where `<COUNT>` is the new number of VCPUs available to the VM:

```
user@host:~$ virsh setvcpus VM 8
user@host:~$ virsh vcpucount VM
maximum      config        16
maximum      live          16
current      config         4
current      live           8
```

<COUNT> cannot be bigger than the maximum amount of VCPUs. The new VCPUs are not used without proper configuration, which is not a behavior specific to CPU reallocation but can be observed reallocating other resources too (see Section 3.1 about *logical memory hotplug* for example). A look at the lscpu output on the guest shows them as offline CPUs. They can be made available for linux, if the QEMU guest agent is configured properly [19]. QEMU (quick emulator) is the emulator used by KVM. The <devices> section of the VM configuration has to include:

```
<channel type='unix'>
   <source mode='bind'/>
   <target type='virtio' name='org.qemu.guest_agent.0'/>
</channel>
```

Further, the QEMU guest agent has to be installed on the guest. On Ubuntu the package is called qemu-guest-agent. Then the guest CPU can be made available (online) using the guest flag:

```
user@host:~$ virsh setvcpus VM --guest 8
```

Completely removing a VCPU is not possible, but the VCPU can be marked offline (made unavailable for the guest) again:

```
user@host:~$ virsh setvcpus VM --guest 4
```

This scenario enforces the assumption, that removing resources is generally harder than adding them, because the guest operating system (OS) expects the resources it was provisioned with to be present [14].

3.2.2 vcpupin

The virsh vcpupin <DOMAIN><VCPU><CPU_LIST> command allows to pin virtual CPUs (VCPUs) to the host's physical CPUs. <VCPU> is the VCPU which gets pinned to a list of physical CPUs (<CPU_LIST>):

```
user@host:~$ virsh vcpupin VM 4 0,2-3
```

This forbids VCPU number 4 of the VM to use physical host CPU 1 for its calculations. The new CPU affinity can be verified with the virsh vcpuinfo <DOMAIN> command afterwards:

```
user@host:~$ virsh vcpuinfo VM
VCPU:           4
CPU:            3
State:          running
CPU time:       0.1s
CPU Affinity:   y-yy

[...]
```

3.2.3 CPU <shares>

If two or more VMs make use of the same CPU, the relative amount of computing time a VCPU is allowed to consume can be changed at runtime through the `virsh schedinfo` command. A single VM always uses the whole CPU capacity. The initial configuration can be found in the XML file of the domain:

```
<domain>
  ...
  <cputune>
    ...
    <shares>5</shares>
    ...
  </cputune>
  ...
</domain>
```

VMs configured with a `<shares>` value lower than 5 will get less computing time assigned and the opposite is true for VMs configured with a higher `<shares>` value [17]. The VM with the highest `shares` value on the host receives most of the processor cycles, all other VMs receive their proportional share. This means that it does not matter, if two hosts have shares 10 and 20 or if they have shares 50 and 100 respectively. The behavior should be the same in both scenarios. This behavior can be changed at runtime as follows:

```
user@host:~$ virsh schedinfo VM --set cpu_shares=<NEW_SHARE_VALUE>
```

`<NEW_SHARE_VALUE>` is the new share value. By issuing `top` on the host, the relative priorities for VCPUs of different VMs pinned to the same host CPU can then be observed by finding the QEMU instance processes of the two (or more) VMs. This is visible in Figure 3.3.

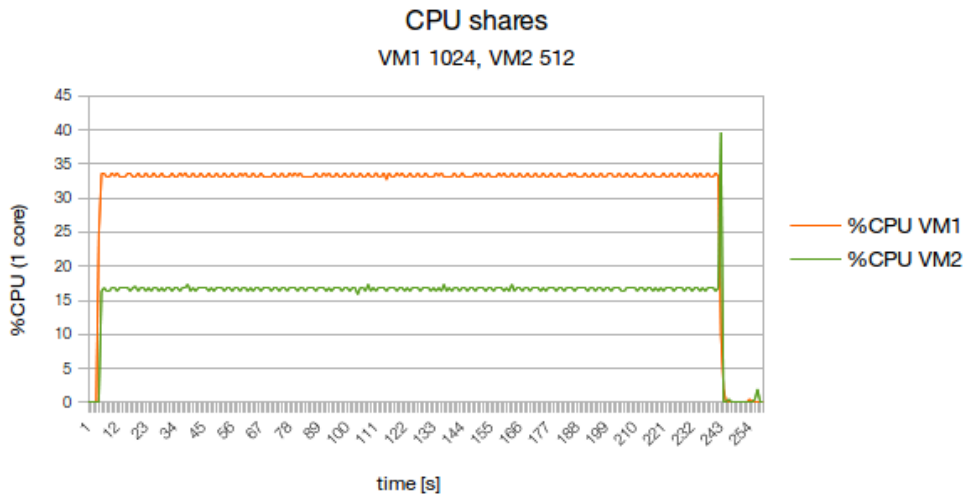


Figure 3.3: VM1 has a CPU share value of 1024, whereas VM2 only gets 512 shares. This means that VM1 gets twice as much computing time on the physical test CPU. The data was gathered using `virt-top`.

3.2.4 CPU <quota>

A CPU quota limits the bandwidth between main memory and the processor [17] and sets all VCPUs of the VM to run at the same speed. Therefore it can also be effectively applied to an environment with just a single VM. The <quota> is a hard limit, which does not depend on the configuration of another VM. The <shares> parameter from last section in contrast does depend on the configuration of other machines. The <quota> is configured in the same place as the <shares> parameter (see last section). It can also be changed at runtime:

```
user@host:~$ virsh schedinfo VM --set vcpu_quota=<NEW_QUOTA_VALUE>
```

<NEW_QUOTA_VALUE> is the new quota value. According to the libvirt documentation, the parameter value can vary in the range [1000, 18446744073709551] in microseconds [17]. To show the impact of the libvirt quota parameter on memory bandwidth in Mbit/s, the *stream* benchmark can be used [22]. The python script `schedinfo.py` (see enclosed CD contents) adjusts the quota parameter several times:

```
# create a data series of length 20
for quota in range(1000, 100000, 100):
    # adjust quota
    call(['virsh', 'schedinfo', 'VM1', '--set', 'vcpu_quota=%s' % quota])
```

Listing 3.4: `schedinfo.py`: the while loop for adjusting the guest quota on the host system.

Afterwards, the benchmark is run on the guest system and the results are saved in a database:

```
    # run the benchmark on the remote host
    # and process the output
    out = check_output(['ssh', 'user@192.168.122.101', '"./stream/stream
        .6176K"'])
    for line in out.split('\n'):
        m = re.search(r'(\w+):\s+(\d+\.\d+)\s+(\d+\.\d+)\s+(\d+\.\d+)\s+(\d
            +\.\d+)$', line)
        if m:
            query = "insert into stream_bench (quota, rate, avg_time,
                min_time, max_time, benchmark_type) values (%i, %s, %s, %s,
                %s, '%s')" % (quota, m.group(2), m.group(3), m.group(4), m.
                group(5), m.group(1))
            cur.execute(query)
            cnx.commit()
```

Listing 3.5: `schedinfo.py`: the benchmark executable is run on the guest system via `ssh` and the benchmark results are processed and saved into a MySQL database.

Only *triad* operations are being evaluated [30]. In this benchmark mode, *stream* fetches three values from memory, performs a multiplication on one value and writes the result of an addition back to memory[30]:

$$a(i) = b(i) + q * c(i)$$

The arrays used by *stream* must be at least 4 times larger than the combined size of all last level caches of the guest. Otherwise the benchmark includes cache bandwidth. This array size constant has to be set at compile time. Because the cache size of the guest was not accessible with `dmidecode`, the combined size of the last level caches on the host (12352 KB) has been used. The calculation has been made according to the documentation [30]:

$$12352\ KB \times 4 = 49408\ KB$$
$$\frac{49408\ KB}{8\ bytes} = 6176000\ bytes$$

Listing 3.6: Compiling the stream benchmark.

```
gcc -O -DSTREAM_ARRAY_SIZE=6176000 stream.c -o stream.6176K
```

After the results were saved to the database, these results of the triad operations were plotted against the quota parameter value using the `plot-results.py` script (see CD contents). The script reads the values from the database and plots them using the python `matplotlib`. This benchmark result can now be used to approach a reasonable quota value range, where the impact on the guest is most useful.

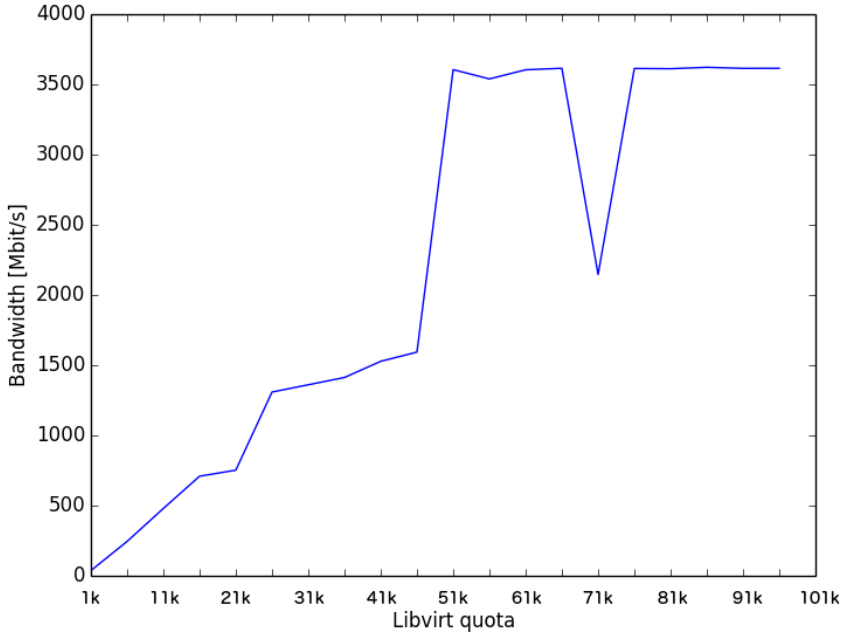


Figure 3.4: Plotting the results of the stream triad benchmark. This test was performed with the libvirt quota parameter in range [1000, 100'000] microseconds.

The host running the VM had two CPUs and 4GB of RAM assigned. The guest was configured with one VCPU and 1GB of RAM. The resulting graph in Figure 3.4 shows clearly, that a reasonable quota value for the specified testing environment has to lie within range [1000, 100k] or smaller. Values above 100k for the quota parameter do not change memory bandwidth any further.

This finding—the quota paremter being most useful within range [1000, 100k]—could be confirmed by various redundant performance benchmarks on another host machine.

The resulting graphs are given in Appendix A. They show the implications of the quota parameter on VM performance. This analysis has been performed using the *phoronix* test suite to see when performance of the benchmarks in the suite is degraded by the quota.

3.3 Disk I/O Speed

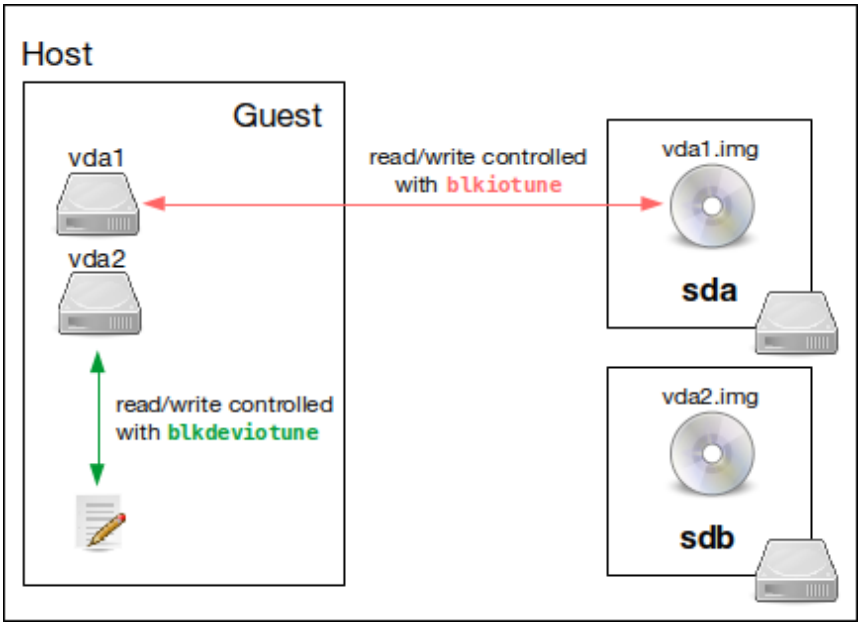


Figure 3.5: Virtual disk images are located on physical host drives. `virsh blkdeviotune` is used to limit the speed to/from the guest block device (guest block devices *vda1* and *vda2*). `virsh blkiotune` is used to manage host block devices (*sda* and *sdb*).

The VM used in this section only contains one virtualized hard disk. More than one disk per guest may be useful in different use cases, e.g. if another guest disk needs to be on a dedicated physical host disk for security reasons or if more space on a dedicated device is needed (see illustration in Figure 3.5). The disk file (see Listing 3.7) is located on the physical drive of the host. Therefore, there exist two ways to control disk I/O speed for the VM. On the guest side, the disk I/O speed can be controlled with `virsh blkdeviotune` (see Section 3.3.1). libvirt makes use of QEMU to enforce the `blkdeviotune` limits. The access speed to/from the host disk is controlled using `virsh blkiotune` (see Section 3.3.2), which makes use of cgroups. `blkdeviotune` (the guest side) only allows to set hard limits, whereas the `blkiotune` (the host side, controlled with cgroups) allows to set soft and hard limits, even for individual host block devices.

```
user@host:~$ virsh domblklist VM
Target     Source
------------------------------------------------
vda        /home/user/VM.img
hda        -
```

Listing 3.7: Listing the block devices of the VM.

3.3.1 Guest Block Device Speed

I/O throttling with hard limits for guest block devices is possible with the `virsh blkdeviotune <DOMAIN><DEVICE><PARAMS>` command. Current settings can be queried as follows (where *vda* is the `<DEVICE>` we are interested in):

```
user@host:~$ virsh blkdeviotune VM vda
total_bytes_sec: 0
read_bytes_sec : 0
write_bytes_sec: 0
total_iops_sec : 0
read_iops_sec  : 0
write_iops_sec : 0
```

Listing 3.8: The `blkdeviotune` hard limits to control guest I/O speed on guest block devices.

This also displays all the guest block device paramaters `<PARAMS>`, which can be controlled with libvirt. I/O operations as well as the I/O speed (in bytes per second) of the guest block devices can be configured separately (read/write) or combined (total).

Without changing these parameters, a 200 MiB file gets created on the guest in about 18 seconds:

```
user@guest:~$ dd if=/dev/zero of=file.out bs=200M count=1
1+0 records in
1+0 records out
209715200 bytes (210 MB) copied, 18.4686 s, 11.4 MB/s
```

After throttling the total I/O speed of the VMs *vda* block device to 5 MiB/s, it takes about twice as long to create a file of the same size:

```
user@host:~$ virsh blkdeviotune VM vda --total_bytes_sec 5242880

user@guest:~$ dd if=/dev/zero of=file.out bs=200M count=1
1+0 records in
1+0 records out
209715200 bytes (210 MB) copied, 36.8286 s, 5.7 MB/s
```

Similar examples with different combination of I/O parameters could be constructed. In this test I've chosen to set the rate limit in MB/s and not in I/O operations per second (IOPS), because `dd` did not provide any information about the number of IOPS of the task.

To measure the effect of a `total_iops_sec` limit (set with `virsh`, for the parameter refer to Listing 3.8) on transfers per second (TPS), I've created the `tps-iops-relation.py` script (see CD contents). According to the `iostat` manpage, TPS should be identical to IOPS [13]. The script is run from the host and adjusts the virsh IOPS limit periodically. For this reason it basically consists of one while loop. Every time the virsh IOPS parameter is adjusted, an `iostat` session (Listing 3.9) and a `dd` benchmark (Listing 3.10) on the guest are started.

```
# start iostat in background;
# refresh every second;
# save output in log file
p.call(['ssh', '-i', '/home/cloud/id_rsa_cloud@n02', 'ubuntu@192.168.99.4',
        'rm', '-f', 'iostat.out'])
pid_iostat = sp.Popen(['ssh', '-i', '/home/cloud/id_rsa_cloud@n02',
                       'ubuntu@192.168.99.4', 'iostat 1 > iostat.out'])
```

Listing 3.9: Deleting the old log file first and then starting a guest `iostat` session in the background.

```
# run dd benchmark on vm;
# wait for it to finish
sp.call(['ssh', '-i', '/home/cloud/id_rsa_cloud@n02',
         'ubuntu@192.168.99.4', 'dd', 'if=/dev/zero', 'of=file.out',
         'bs=100M', 'count=10'],
         stdout=open(os.devnull, 'wb'), stderr=open(os.devnull, 'wb'))
```

Listing 3.10: Running a `dd` benchmark on the VM.

The last part of the main loop (Listing 3.11) consists of parsing and printing the IOPS and TPS results, such that they can be read as CSV.

```
# download log file
sp.call(['scp', '-i', '/home/cloud/id_rsa_cloud@n02',
                 'ubuntu@192.168.99.4:/home/ubuntu/iostat.out', '.'],
                 stdout=open(os.devnull, 'wb'), stderr=open(os.devnull, 'wb
                    '))

# parse iostat log
f = open('iostat.out', 'r')
for l in f:
        m = re.match(r'^vda\s+(\d+\.\d+)', l)
        if m:
                tps = m.group(1)
                print "%i, %f" % (iops, float(tps))
```

Listing 3.11: Parsing and printing the result.

Tests with IOPS limits above 50 showed no significant effect on the number of transfers per second of a disk I/O task. Limits under 50 IOPS also lead to a decrease in TPS. The exact results—i.e. the absolute TPS numbers—depend heavily on the configuration of the `dd` benchmark. With a higher block size (`bs`) or a higher `count` the benchmark tends to produce more transfers per second. This result is not precise, because it has not been possible to make out a particular relation (in terms of a constant factor or multiplicator) between the IOPS and the TPS. It has been concluded that `dd` is not quite the right benchmark to achieve exact results in this experiment.

3.3.2 Host Block Device Speed

`virsh blkiotune <DOMAIN>` allows us to query and modify the host disk priority of different guests. `blkiotune` only allows to set priorities for host block devices, not for guest devices. To control guest disk access speed, refer to the last section. The `<weight>`

parameter can be used to set host disk priority in the domain configuration file. The weight for host disks can also be set with `virsh blockiotune`. To set different weights for individual host disks, use the `device-weights` flag followed by a list of device/weight pairs. Weight is always in range [100, 1000] [33].

```
user@host:~$ virsh blkiotune VM --weight 300
user@host:~$ virsh blkiotune VM --device-weights /dev/sda,100,dev/sdb,250
```

Listing 3.12: Example for changing the I/O priority on all host disks to 300 or to set different priorities for the *sda* and *sdb* host devices. *sdb* could be an external host disk or another internal host partition. In both examples, the weights always apply to the guest named `VM` as a whole.

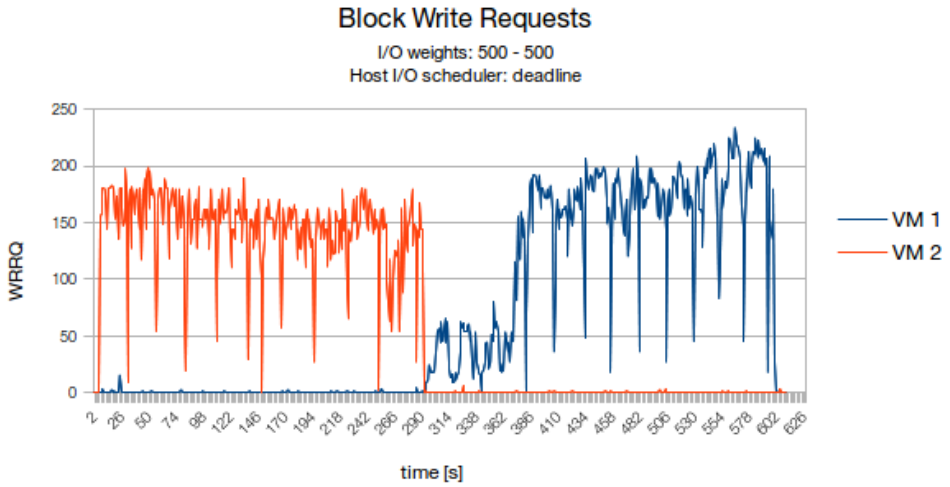


Figure 3.6: Block write requests of two VMs running a `dd` test synchronously. The host performs the writes on disk in batches, because the *deadline* scheduler is active. The VMs both have the same libvirt I/O weights (500).

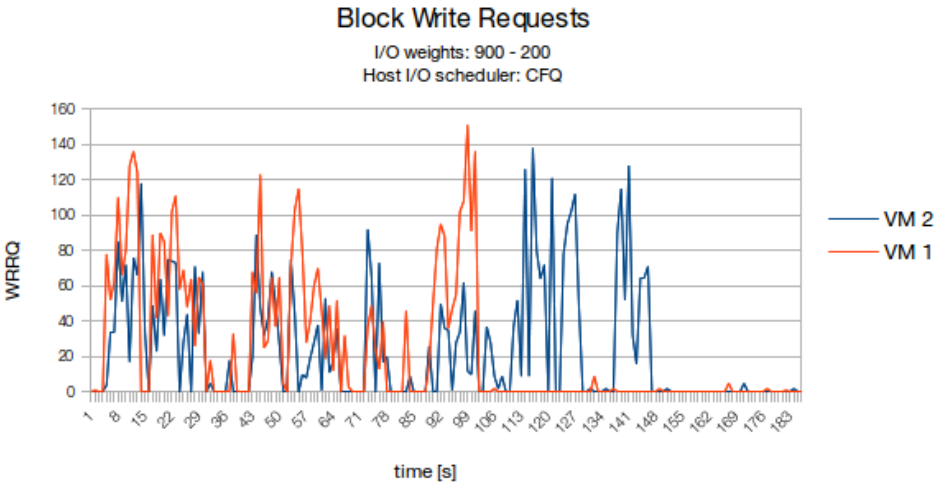


Figure 3.7: Block write requests of two VMs running a *dd* test synchronously. The host prioritizes the writes on disk correctly, because the *CFQ* scheduler is active. The VMs have the libvirt IO weights 900 (VM 1) and 200 (VM 2).

Thorough analysis of host disk IOs with `dd` tests lead to the conclusion, that the I/O scheduler type of the host disk is an important key for using virshs `blkiotune` feature efficiently. First tests with the *deadline* scheduler showed a delay between the writes of two guests executing the same `dd` benchmark started at the same time. The *deadline* I/O scheduler in linux executes the IO tasks in batches [7], which is visible in Figure 3.6. This is undesirable to observe the effect of `virsh`'s soft limits with `blkiotune`. Figure 3.6 does not allow to draw conclusions about the I/O *weight* of the VMs, because on the host, the *deadline* scheduler is in use.

```
user@host:~# echo cfq > /sys/block/sda/queue/scheduler
user@host:~# cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
```

Listing 3.13: Changing the I/O scheduler to CFQ [5].

The *complete fairness queueing (CFQ)* scheduler is designed to support priority based I/O activities [4]. After changing the scheduler on the host (see Listing 3.13), the disk write IOs behaved as expected. Figure 3.7 shows that the host prioritizes the write requests of the two VMs correctly. The complete set of test results is given in Appendix B.

3.4 Network Bandwidth

```
user@host:~$ brctl show
bridge name ... interfaces
virbr0      ... vnet0
                vnet1
```

Listing 3.14: The default libvirt network setup with the *virbr0* virtual switch and two VM network interfaces.

libvirt creates a virtual network for the VMs by default [20]. All VM traffic is directed through a virtual switch (interface labeled *virbr0* by default). This is shown in Listing 3.14 or graphically in Figure 3.8. The bridge runs in network address translation mode (NAT), which means that the virtual network with its VMs is not visible from computers on the host network (besides the host itself), because the VMs operate in a dedicated subnet. An instance of the `dnsmasq` DHCP server listens on the bridge interface and assigns IPs to VMs. On newer versions of libvirt an additional network interface card (NIC) named *virbr0-nic* ensures the static assignment of the MAC adress for the lifetime of the bridge [27].

Each VM interface has two names (see the yellow boxes in Figure 3.8). On the guest side, the interface will be most likely labeled as *eth0*. On the host side, the interfaces of the guest machines are labeled as *vnet0, vnet1, ..., vnetn*. It is recommended to change the default host labels of the NICs created for running instances (*vnet0, vnet1, ..., vnetn*), because the machines may be started in different order. The host interface label of a VM can be changed in the `<interface>` section of the domain configuration xml file:

```
<interface type='network'>
  ...
  <target dev='<IFACE_NAME>'/>
```

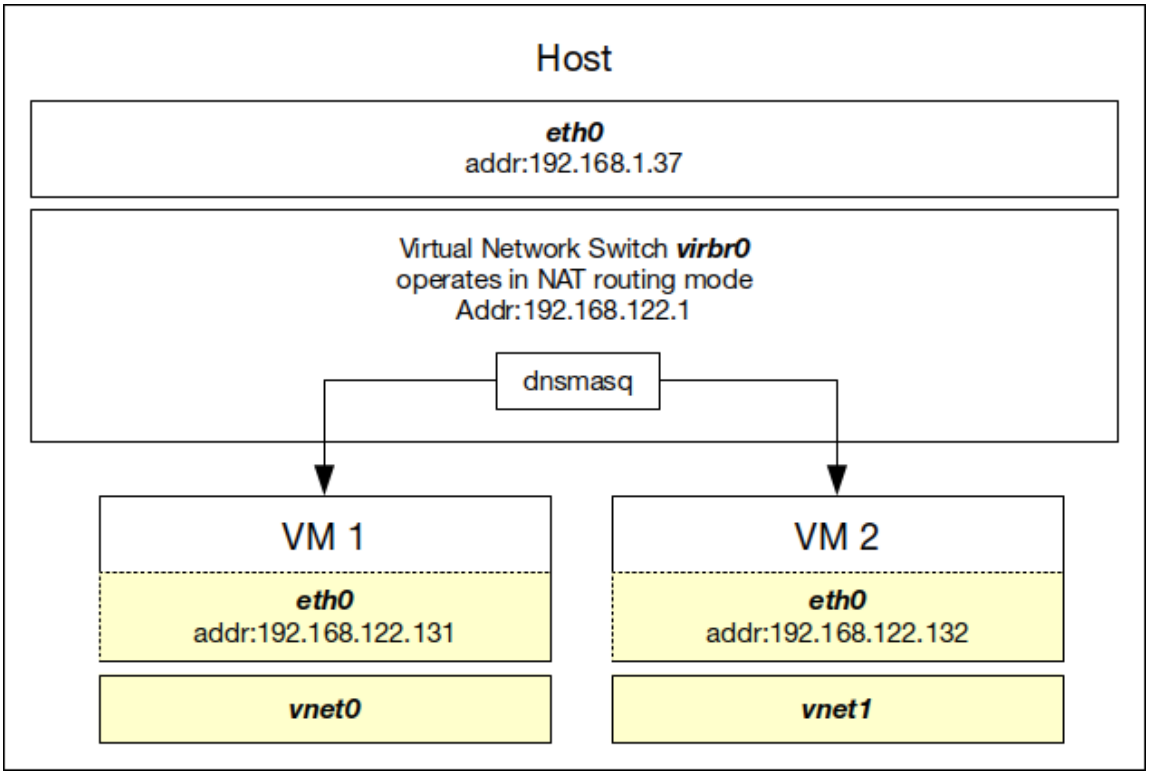


Figure 3.8: An illustrative network setup on hosts using (and guests launched through) the libvirt API [20].

```
    ...
  </interface>
```

Listing 3.15: Assigning a more descriptive label to the host interface of a domain.

<IFACE_NAME> is the new host interface name. By explicitly labeling the host interfaces (*vnet0, vnet1, ..., vnetn*), they still show up in the corresponding guest as the ethernet interface *eth0* with its IP assigned but don't accidentally change their host labels on reboot. The mapping from the VM to the host interface can be made clearer with a more descriptive label, so you can instantly tell which host interface belongs to which VM interface. This is especially useful in scenarios as in Listing 3.17, where VM name and interface name are required to map to the same VM.

Network rate hard limits for the whole network can be set via the libvirt network configuration file. This file can be modified for the `default` network with `virsh net-edit default` [18]:

```
<network>
  ...
  <bandwidth>
    <inbound average='1500' peak='3000' burst='4000'/>
    <outbound average='128' burst='100'/>
  </bandwidth>
  ...
</network>
```

Listing 3.16: Exemplary rate hard limit settings in the network configuration

Incoming and outgoing traffic can be controlled separately. According to the XML format documentation for libvirt networking [18], the average and maximum (`peak`) rate in kilobytes/second can be set. The `burst` attribute in kilobytes sets limit to the amount of data transmitted at `peak` speed. Only the `average` limit is mandatory. `peak` is not yet implemented for outbound traffic. It can be specified but will be ignored [18], because the outbound traffic on the VMs interface represents the inbound traffic on the corresponding host interface (see yellow boxes in Figure 3.8) and `tc` cannot be used to control the `peak` rate on ingress traffic. These rate hard limits can also be applied in the same way to dedicated interfaces of a domain.

The virtual network must be one of type *routed*, *NAT* or *isolated* [18]. As explained in the beginning of this section, NAT is the default mode. All VMs communicate inside a subnet and the virtual switch *virbr0* acts as a router between the subnet and the host LAN. In routed mode, the VMs communicate directly over the same LAN as the host. In isolated mode, the VMs communicate on a subnet as in NAT mode, but the subnet is completely isolated (i.e. the VMs cannot reach the host LAN and vice versa).

libvirt itself does also allow to change the network rate hard limits at runtime with the `virsh domiftune <DOMAIN><IFACE> [ <MODE> <AVERAGE>,<PEAK>,<BURST> ]` command, where `<AVERAGE>,<PEAK>` and `<BURST>` are the limits in kilobytes/sec. `<MODE>` stands for the `--outbound` or `--inbound` flag to adjust outbound or inbound traffic. The dynamic adjustment at runtime has the same effect on the guests rate limits as setting the parameters in the network configuration xml file (see Listing 3.16).

```
user@host:~$ virsh domiftune VM vnet0 --inbound 2000 --outbound 2000
```

Listing 3.17: Example for changing rate hard limits of a domain at runtime. Interface label and VM name have to match the same VM (see Listing 3.15 on how to change the host interface label of a domain).

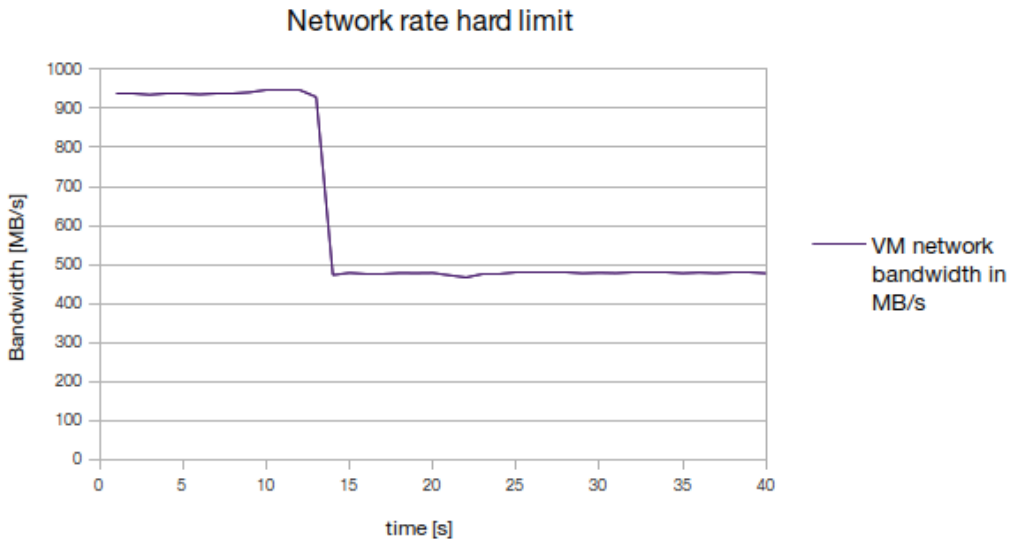


Figure 3.9: The result of limiting inbound network bandwidth to 500 MB/s (average) with `virsh domiftune VM vnet0 --inbound 500000`.

Chapter 4

Resource Reallocation in OpenStack

In the OpenStack framework, flavors are used to manage physical resources. A flavor is a hardware template, which specifies the RAM size, disk size, number of cores and other VM parameters [9].

The current version of OpenStack already allows to change the libvirt parameters relevant in this thesis, which are memory soft and hard limits, CPU `<shares>`, the IO `<weight>` and network bandwidth. The flavor *extra_specs*, a list of key/value pairs, can be populated with the according parameters. This includes soft limits like IO `<weight>` or CPU `<shares>` and hard limits like CPU `<quota>`, disk writes/reads per second and the in- and outbound bandwidth. Currently for such a constraint to be active, a VM has to be booted with a flavor which holds the desired *extra_specs*. Therefore the flavor is not the solution of choice for dynamic resource reallocation, or the change of the constraints during runtime. Also, by creating a new flavor for every possible combination of resource limit and virtual machine, the administrative overhead of the flavors would become too big.

Reallocation of resources is also possible through live migration, which allows to change the flavor of a VM with minimum downtime. The live (or current) configuration of KVM hosts can be changed dynamically at runtime with the libvirt API. The extension written in this thesis is not an attempt to expose the whole libvirt API through the nova API (nova is the part of OpenStack that provides acces to the compute nodes). Rather than that, the focus lies on the libvirt parameters which allow to set *soft limits*. Those limits only apply if resources get scarce and are applied in relation to the usage of other machines. These are especially the CPU `<shares>`, the IO `<weight>`, and the memory soft limits. libvirt's network bandwidth parameters hitherto do not offer any comparable soft limits. Absolute *hard limits* have to be used to reallocate the bandwidth. In Section 4.5 it is shown how to manage network bandwidth priorities with `tc`.

The remainder of this chapter serves the purpose to document the process of writing an extension for dynamic resource reallocation with OpenStack in detail.

4.1 Using the libvirt Interface in OpenStack

To develop a nova API, which allows to access libvirt functionality to perform resource reallocation, the devstack *All-In-One Single VM* approach was used [8]. For this setup, a VM in the testbed served as devstack developer machine. Devstack makes it possible to install a complete OpenStack environment on a VM or physical host.

Even though V2 API of nova should not be extended anymore, the code written in this thesis relies on the V2 extension mechanism. The V3 microversion framework as documented in the nova developer API [1] was not applicable. In the very first iteration of the extensions, the microversion framework has been adopted as documented. Unfortunately, the plugin was not recognized by OpenStack. After a discussion with the supervisor, the plugin has been edited to use the older fallback V2 extension mechanism. This approach is much simpler, because it only requires to place the new controllers in the right `contrib` directory. Afterwards, the extension shows up in OpenStack.

The nova API can be extended by placing custom controllers in the `contrib` folder of the directory tree. The `LibvirtResourcesController` written in this thesis extends the API with *actions* on the *servers* resource to control the PR of a server (see Appendix C.1 for detailed documentation).

To test the new functionality of the API, the python command-line interface `nova` has been extended accordingly. Refer to Appendix C.2 on how to use the CLI.

4.2 Reallocate Memory (RAM)

To integrate the control of ballooning capabilities into the OpenStack API, one has to decide if the memory defined in the flavor of an instance represents the usage maximum or if the guest is allowed to acquire more RAM than specified in the flavor. Figure 4.1 shows the difference between the two possible approaches to create a new instance capable to balloon up and down. Approach (1) leads to a discrepancy between the flavors memory size defined in the database and the actual size of the VM. By modifing the flavor object at run-time and expanding its memory size before creating an instance, the actual template as defined in the database is not affected. Further, the dashboards in horizon (web GUI of OpenStack) rely heavily on the flavors memory size to provide an overview of memory used. Horizon uses the VM flavor to display aggregate information. If instances were allowed to use more RAM than defined by the flavor, the aggregated information in the dashboards would not reflect the actual aggregate resource consumption of all VMs on the host. Because of these inconsistencies, approach (2) where the flavor represents the maximum memory size allowed to be used by the machine was used in the implementation. At the end of instance creation the instance will ballon down with factor r which allows for an expansion later on if needed. An end user will have to keep in mind to choose a flavor with a bigger memory size than in a default OpenStack environment without the extension. Otherwise problems could arise while booting the VM. The constant factor r can be modifed by setting the `ballooning_ram_ratio` option

in the [libvirt] section of the nova configuration at /etc/nova/nova.conf. For transparency reasons, the parameter—and therefore also the post-creation ballooning—is disabled by default. The behaviour has to be explicitly enabled by setting an appropriate ballooning_ram_ratio (e.g. 2).

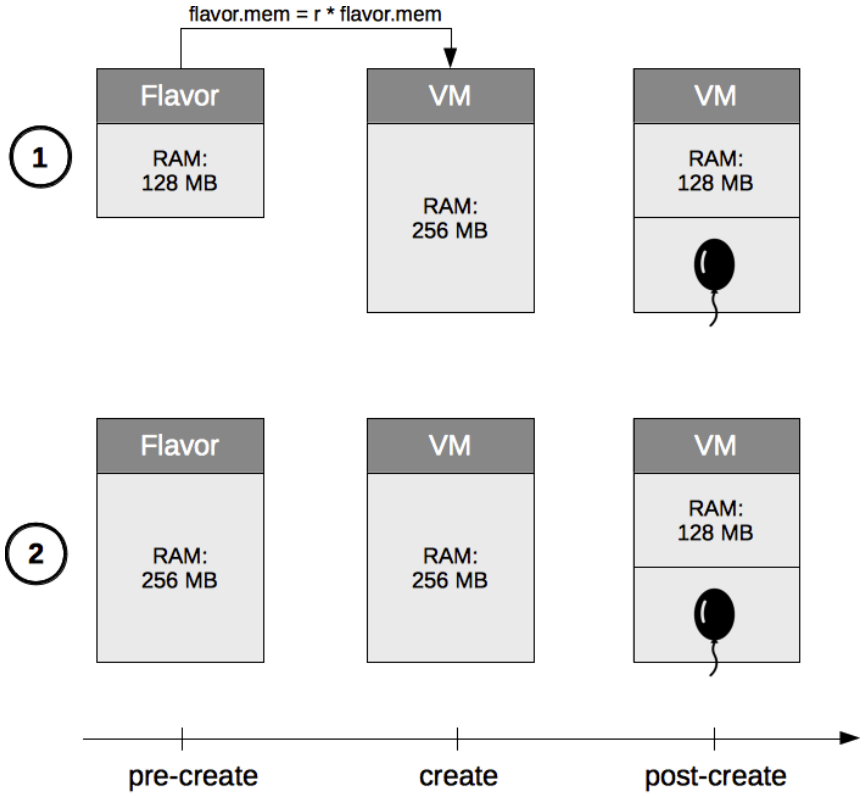


Figure 4.1: In (1), the final allowed maximum memory size is larger than the size of the flavor template, as defined in the database. In (2), the flavors memory size represents the maximum memory.

It was possible to write a hook to perform the ballooning after instance creation (see Appendix C.3 for details). Unfortunately there exists **no such hook for the start of a VM!** So once the instance is rebooted, it will be started with the maximum memory configuration.

4.3 Reallocate Computing Time (CPU <shares>)

During the course of this thesis, the importance of so called *soft limits* has been repeatedly emphasised. For this reason, the nova extension does not offer the possibility to add more VCPUS, but rather to change the libvirt <shares> parameter (see Section 3.2.3). The libvirt CPU *quota* was also not exposed through the nova API, because this is a *hard limit*. Because VCPU pinning restricts the VM from using all available CPUs, it is rarely used [3]. Therefore these settings will remain configurable only through virsh and are not contained in the API extension.

4.4 Set Disk Priorities (I/O <weight>)

As mentioned in Section 4.3, the importance of *soft limits* also led to the decision to only make the libvirt disk IO <weight> parameter available through the nova API. Also, the results of setting a disk IO limit in IOPS (hard limit) were not very precise, as explained in the end of Section 3.3.1. The extension only enables to set the disk IO weight of the whole guest. Weights on a per disk level have to be set with virsh.

4.5 Set Network Bandwidth Priorities

The network throughput of VMs launched with OpenStack can be controlled using the tc utility. For the default network setup in a KVM only environment I refer to Section 3.4. An OpenStack network configured in the *Flat DHCP Network Manager* mode [25] creates an additional interface and a new bridge labeled *br100* by default. The KVM bridge *virbr0* is still around but no interfaces are attached to it. For matter of simplicity, the new interface will be labeled *eth1* in this section. In a real environment, the label depends on how much physical ethernet interfaces the host machine has. If the host has no physical ethernet interface, the interface created by OpenStack may also be labeled *eth0*, because *eth0* did not exist yet (see Listing 4.1).

```
user@host0:~$ brctl show
bridge name ... interfaces
br100       ... eth0
            ... vnet0
            ... vnet1
virbr0
```

Listing 4.1: Illustrative OpenStack bridge setup on a host with no physical ethernet interface.

```
user@host1:~$ brctl show
bridge name ... interfaces
br100       ... eth1
            ... vnet0
            ... vnet1
virbr0
```

Listing 4.2: Illustrative OpenStack bridge setup on a host with a physical ethernet interface labeled *eth0*. The interface created by OpenStack is therefore labeled *eth1*.

Tc queuing disciplines (*qdiscs*) can be used to control the egress and ingress traffic. A classfull egress qdisc can contain many other qdiscs with more classes (see Figure 4.3). Whereas outgoing traffic can be filtered by its source and shaped accordingly, an ingress policy only allows to drop incoming packets. The tc *ingress* qdisc is classless. It is therefore not possible to use a priority qdisc for incoming VM traffic on *eth1*.

4.5.1 Priorize with the `tc` priority (prio) qdisc

> Note: the most recent version of the nova API extension features the htb qdisc approach described in the next section.

To prioritize the egress traffic of several VMs, the *prio* priority qdisc can be used. Even though the current version of the API extension uses a htb qdisc (see next section), this chapter describes how network can be prioritized with a prio qdisc. Such an approach has been used in early iterations of the API extension.

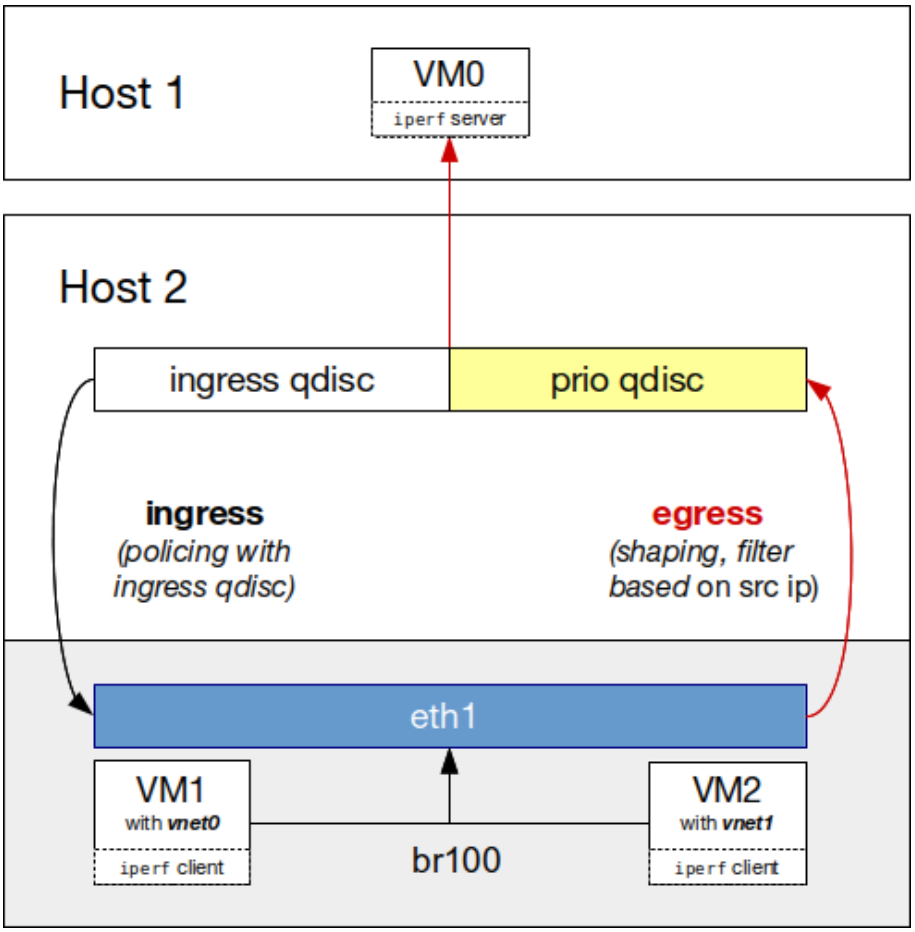


Figure 4.2: The setup used to experiment with the `tc` qdiscs and filters on a default OpenStack network in "Flat DHCP Manger" mode. *eth1, vnet0* and *vnet1* are attached to the bridge *br100*. The red arrows show the direction, in which the nova extension is able to prioritize network traffic. Figure 4.3 describes the priority qdisc (yellow) with its classes and filters in further detail.

In my tests, *VM0* on *host 1* served as a reference machine representing the outside world. With the `iperf` utility running in server mode on *VM0* and in client mode on *VM1* and *VM2*, the egress qdiscs and filters could be confirmed as working correctly. Showing packet statistics with the `tc -s qdisc ls dev eth1` command and pinging the reference VM0 from the corresponding VMs on *host 2* confirmed that packets are classified correctly in classes *1:1* and *1:2*. The qdiscs and filters on *host 2* were set up as in Figure 4.3.

The *pfifo* qdisc can be used to limit the queue size of an interface. The length of qdisc *10:* was set to 5 and the length of qdisc *20:* to 1 packet. If the queue is full, no more packets

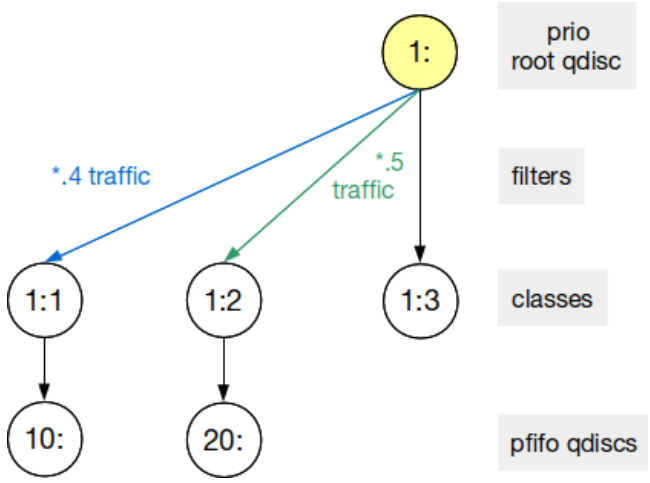


Figure 4.3: The qdiscs and filters used to shape VM outbound traffic; **.4* and **.5* are the IPs of the VMs *VM1* and *VM2*. Traffic ist filtered and classified by source IP adress.

can be enqueued. Because the queue on qdisc *10:* is longer, the VM with IP **.4* is able to send more packets at a time.

Tests without the *10:* and *20:* qdiscs made it impossible to send packets from the two VMs *VM1* and *VM2* at the same time, because packets enqueued in class *1:1* always had priority. With no pfifo qdisc in place, which limits the amount of packets allowed to send, the packets in class *1:2* with lower priority had to wait until no more packets were enqueued in class *1:1*. With the *pfifo* qdiscs installed, VM **.5* was always able to send a packet for a short time, after the qdisc *1:1* with packets from VM **.4* stopped sending packets because the queue limit of 5 has been reached. This approach allowed for more fairness, but still leaves the priorities intact.

An early version of the python `nova` client extension did only allow to bulk set network priorities for a set of instances. In comparison to the reallocation of the other resources (as CPU shares or disk weight), the network priorities could only be set for a list of instances, not for individual instances. This was so, because the priority mechanism was implemented with the `tc` traffic control utility and network priority is not controlled with libvirt. As described in this section, the setup of a working qdisc structure is slightly more complicated than accessing a single method of a library. Once the *prio* qdisc with a fixed number of bands is created, it is for example not possible to add another band, i.e to add a priority for another machine. Of course, one could just create a *prio* qdisc with a lot of bands, but how does one find a reasonable maximum? Therefore this ancient client version did only allow to set network priorities for a list of instances. This simplified the setup process. Every time new network priorities where set, the qdisc structure has been destroyed and recreated from scratch.

In the old prio implementation of the API extension, network priorities could be set in range [1:9], with 1 being the highest priority. This way, the classes can be labeled from 1:1 up to 1:9. In a very early version of the extension, the length of the pfifo queue varied depending on the class priority. The size of the pfifo qdiscs has been made static in later versions, because different pfifo qdisc length meant that some classes may drop more packets than others (tail-drop).

To reallocate network priorities, the new configuration variable `pfifo_limit` has been introduced. It could be defined in the same place as the `ballooning_ram_ratio` (see Section 4.2). This value determined the length (in packets) of the *pfifo* qdiscs inside the classes/bands. The length may depend on the amount of network traffic and should be changed to suit the needs of the network. With a low `pfifo_limit` (i.e. with shorter *pfifo* qdiscs), the VMs will alternate more frequently while sending packets. If the pfifo qdiscs are too short, too many of the arriving packages will be dropped. If the qdisc size is big, the VMs with smaller network priority have to wait longer until they are able to send a packet.

4.5.2 Priorize with the `tc` htb qdisc

The hierarchy token bucket qdisc (htb) is another qdisc which allows to prioritize network traffic. In comparison to the prio qdisc, a minimum bandwidth can be garuanteed. This avoids the starvation of traffic flows, as can be seen by example in the evaluation (see Section 5.4.1). Therefore, the most recent API extension code makes use of htb qdiscs.

Classes under a root class in an htb qdisc can borrow excess from each other. This concept of *borrowing* the excess bandwidth to other machines in combination with the guaranteed minimum *rate* makes the htb qdisc the perfect tool for priority based traffic control. Figure 4.4 illustrates a `tc` htb setup with classes which are allowed to borrow excess bandwidth from one another.

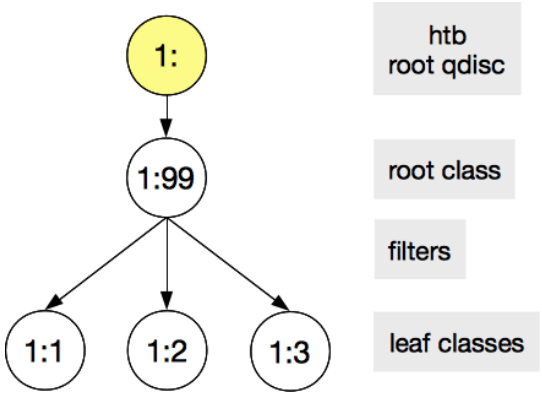


Figure 4.4: htb qdisc setup to prioritize network traffic. Without the root class, excess bandwidth would not be borrowed.

The most recent version of the client extension does allow to change the priority of a single instance. This is an enhancement to the earlier implementation with the prio qdisc in terms of usability and consistency, because the libvirt parameters can also only be changed for single VMs.

The nova configuration has been extended by a new constant in the `[tc]` group of the configuration file at */etc/nova/nova.conf*. `htb_rate` defaults to 80 and represents the total amount of allowed bandwidth of all the VMs on the host in Mbit. It needs to be changed according to the networks specific needs.

Chapter 5

Evaluation of the nova API extension

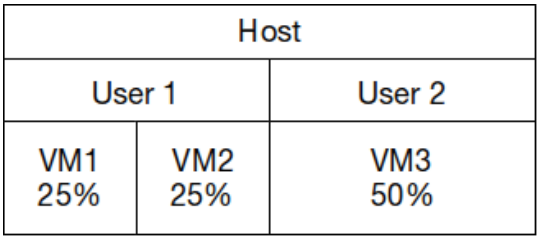


Figure 5.1: The simple multi user scenario to evaluate various aspects of the API extension. In an ideal world, both users may be allowed to use half of the resources available.

The scenario to evaluate the nova API extension and the `nova` CLI extension is illustrated in Figure 5.1. One user runs two instances which in sum should not use more than fifty percent of the hosts resources. The other user runs only one machine, which is allowed to use the other fifty percent of the resources available. In a more general scenario, resource partitioning decisions should be made by the user. But to keep the scenario simple, user 1 decides to distribute the resource load evenly on the two VMs *VM1* and *VM2*. The sections in this chapter show the results when the extension is used to give *user 1* more resources and taking it from *user 2* (giving *user 1* more weight or a higher priority).

To test the correct behavior of the extension—especially the methods to set new resource parameters—the `virt-top` utility has been used for the CPU and disk I/O experiments. Memory has been measured with the `smem` utility using the proportional set size (PSS) (see Section 3.1.2). The network bandwidth has been assessed with `iperf`. All results have been processed as CSV and illustrated graphically. All self-written scripts used in this evaluation are on the CD. The getter methods to display the current resource limits of the VMs were tested by verifying the provided output. All tests have been performed on a machine running devstack on Ubuntu 14.04.

5.1 Memory Reallocation

5.1.1 Hard Limit, `nova lv-set-mem`

All three VMs have been booted with 2048 MiB of maximum memory (`<memory>`). Afterwards, the current memory (`<currentMemory>`) of VM1 and VM2 has been restricted to 512 MiB and the current memory of VM3 to 1024 MiB, such that RAM is distributed according to the chosen evaluation scenario from Figure 5.1.

After around 30 to 40 seconds, the current memory of VM3 has been decreased by 512 MiB and the current memory of VM2 has been increased by the same amount (see Figure 5.2). User 1 does not use as much RAM on VM1 as he could, which is perfectly fine. The new constraints are applied within 5 seconds which is fast in comparison to the memory soft limit (see Figure 5.3).

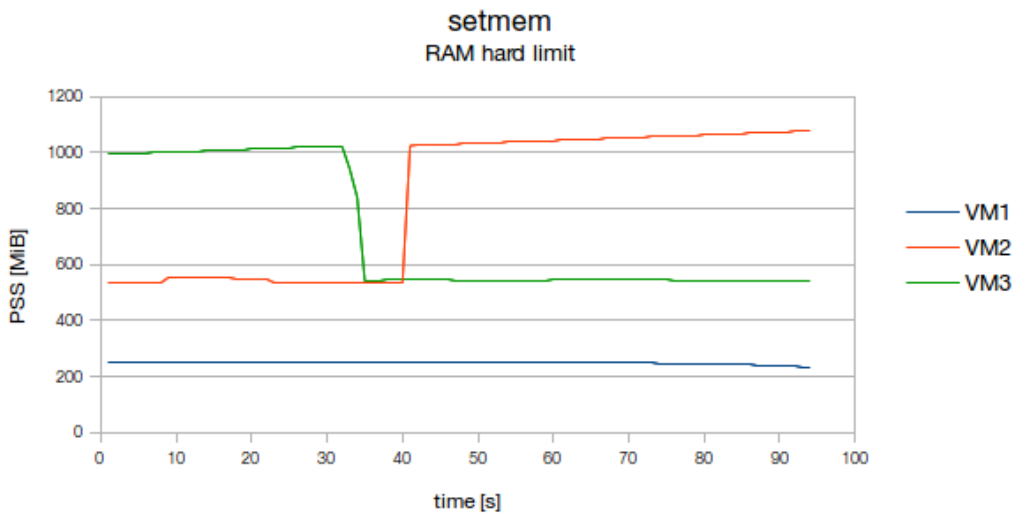


Figure 5.2: Using `nova lv-set-mem` to distribute RAM: `nova lv-set-mem VM3 512 && nova lv-set-mem VM2 1024`.

5.1.2 Soft Limit, `nova lv-set-mem-soft-limit`

VM1 and VM2 had a memory soft limit of 512 MiB and VM3 a soft limit of 1024 MiB, according to the evaluation scenario in Figure 5.1. After some time (around 100 to 150 seconds), the soft limits have been changed. The memory hard limit was 2048 MiB for all machines. Because both users make extensive use of their VMs there is memory congestion on the host and it falls back to the memory soft limits. The new RAM soft limit is 1024 MiB for VM2 and 512 MiB for VM3. This change in RAM priority (a reallocation of RAM from user 2 to user 1) is visible in Figure 5.3 around second 250 to 350. It can be noticed, that the soft limit reacts slower to a change than the memory hard limit from Figure 5.2. This might be due memory soft limits being a cgroup *best-effort* feature [24],

which does not come with any sort of garuantee. When using the soft limit, RAM cannot be controlled as precisely as when using the hard limit.

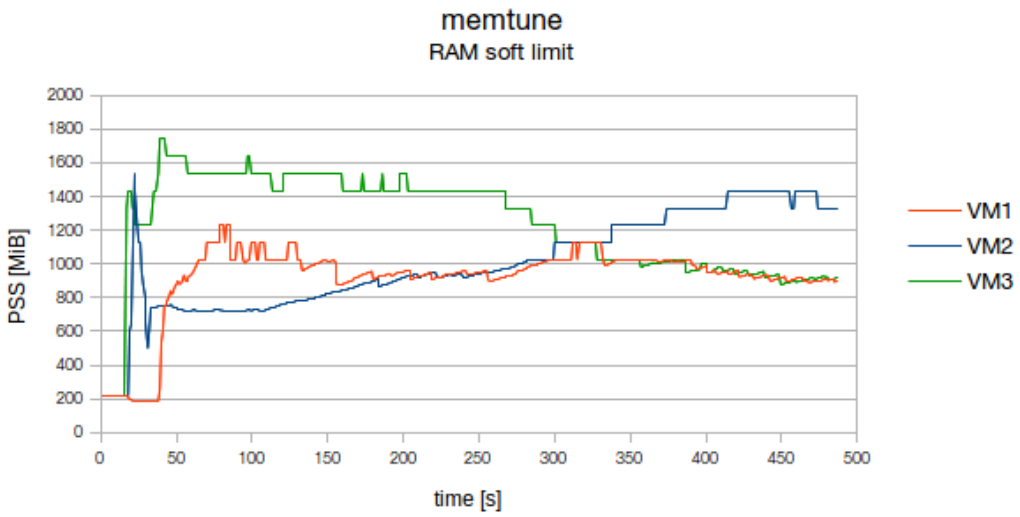


Figure 5.3: Effect of `nova lv-set-mem-soft-limit` to distribute RAM (`nova lv-set-mem-soft-limit VM3 512 && nova lv-set-mem-soft-limit VM2 1024`). The soft limit on VM1 remains unchanged.

5.2 CPU Reallocation

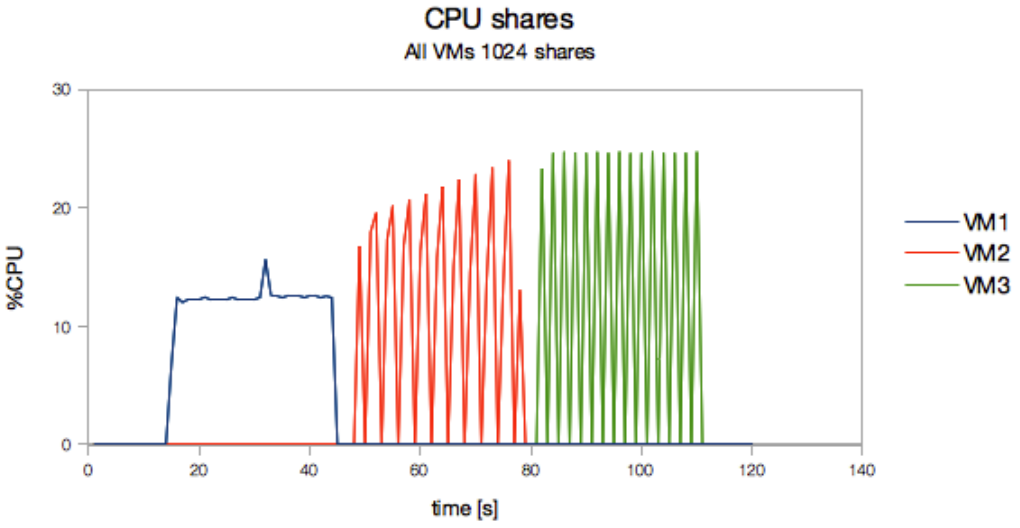


Figure 5.4: Three VMs with the same `<share>` value, which use CPU resources sequentially.

Figure 5.4 shows the default configuration. All new OpenStack instances are assigned the same `<share>` value. Because the VMs work sequentially, this soft limit has no effect.

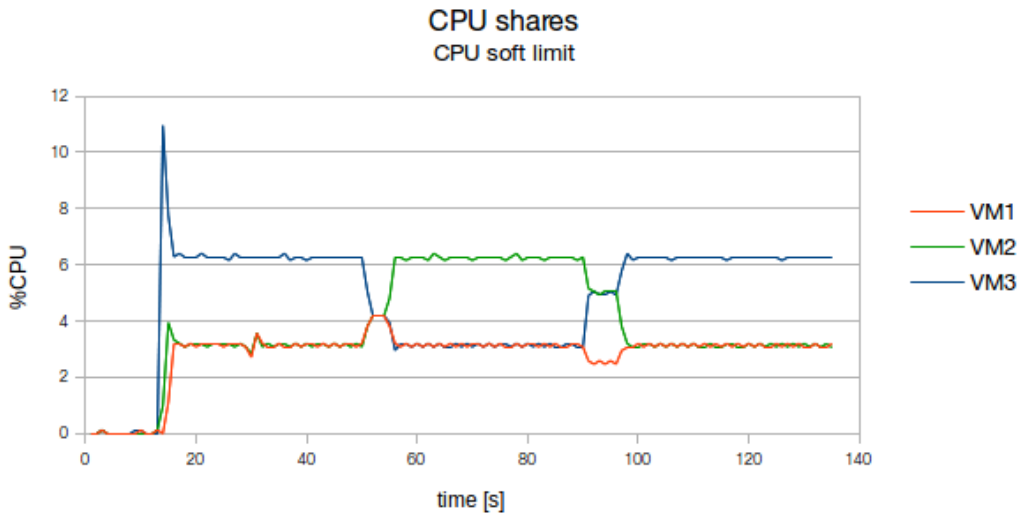


Figure 5.5: Effect of `nova lv-set-cpu-priority` to distribute computing time (`nova lv-set-cpu-priority VM3 1024 && nova lv-set-cpu-priority VM2 2048`). The CPU shares on VM1 remain unchanged (1024 shares).

CPU is not scarce in this example. In Figure 5.5 the VMs of user 1 both have 1024 CPU shares and VM3 of user 2 is configured with 2048 shares at the beginning. Because all the three VMs are pinned to the same physical host CPU, they compete against each other for CPU resources. After 50 to 60 seconds, 1024 shares are taken from VM3 and given to user 1. Again 30 to 40 seconds later, the original configuration where both users have 2048 CPU shares in total has been restored. This change in CPU priority is applied within 5 to 10 seconds, which is comparable with a change in the memory hard limit.

5.3 Disk I/O Reallocation

To evaluate the disk I/O weight capabilities of the nova extension, the CFQ scheduler has to be used (Section 3.3.2). VM1 and VM2 of user 1 both have a disk I/O weight of 500 for the start. VM3 of user 2 has an I/O weight of 1000. As observed in Figure 5.7a this starting position to evaluate disk I/O reallocation does not fully comply with the evaluation scenario in Figure 5.1. It shows, that VM3 completes a write job (writing 3 GiB) a little (around 100 seconds) faster than the machines of user 1. For a synchronous disk I/O job on the same host block device (which takes about 700 seconds to complete in this example) a VM with weight 1000 could be expected to be ten times faster than a VM with weight 100. So a VM with weight 1000 could also be expected to complete the task twice as fast as a VM with weight 500. Figure 5.7a shows that this is not quite the case. This might be due to too little disk activity to fully activate the soft limit during the whole duration of the disk job.

To test another combination of I/O weights, VM3 has been assigned a weight of 300, VM2 a weight of 900 and VM1 a weight of 500 afterwards. As seen in Figure 5.7b the priorities

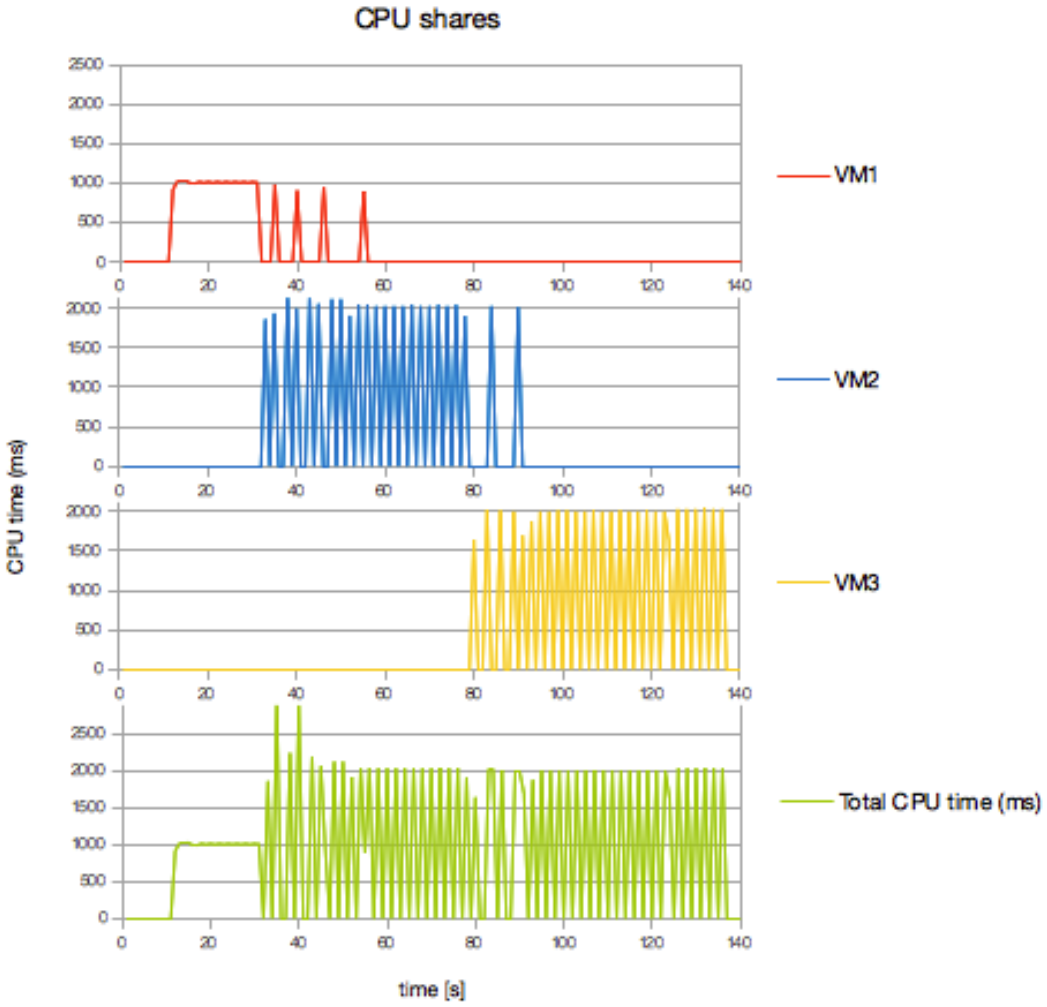
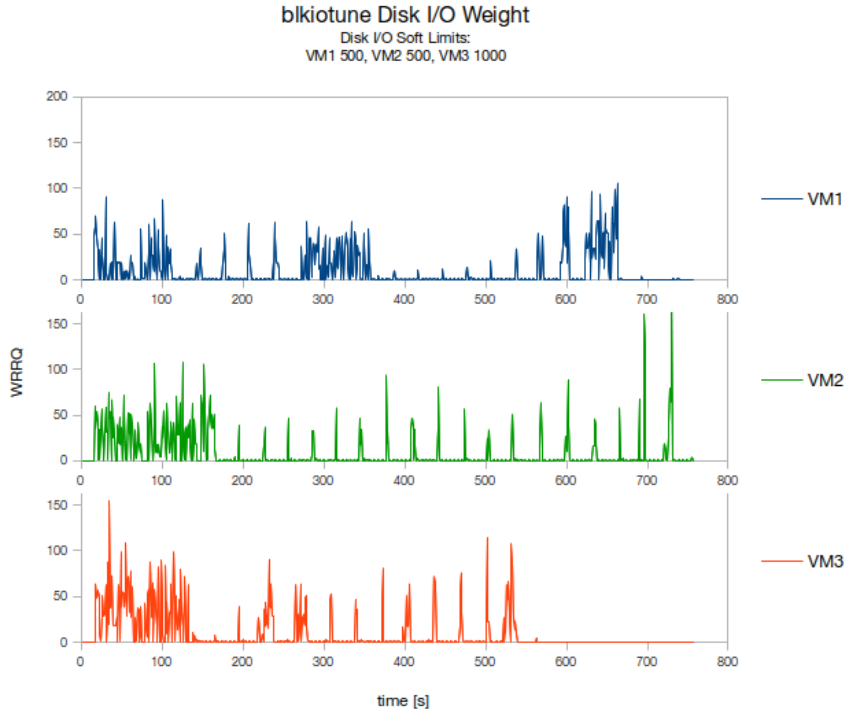


Figure 5.6: Effect of `nova lv-set-cpu-priority` to distribute computing time (`nova lv-set-cpu-priority VM3 100 && nova lv-set-cpu-priority VM2 500 && nova lv-set-cpu-priority VM3 1000`). It is clearly visible, that every time a VM with more shares starts a CPU intensive task, the CPU time of the other machine with less CPU shares gets scarce. The VM1 with weight 100 (which is the minimum weight) also shows throttled CPU time if no other VM competes for CPU.

are respected when performing a synchronous write task of size 1 GiB. Even though VM1 and VM3 complete the job at the same time, VM1 starts the write job before VM3 due to heigher I/O weight. VM2 has been clearly prioritized and completes the job first and also starts writing early in comparison to the other two machines.



(a) The starting point to evaluate disk I/O weights with `nova lv-set-io-priority`. The VMs write 3 GiB of data.

(b) Using `nova lv-set-io-priority` to change host disk I/O priorities: `nova lv-set-io-priority VM3 300 && nova lv-set-io-priority VM2 900`. The disk weight of VM1 remains unchanged (weight 500). The VMs write 1 GiB of data.

Figure 5.7: Evaluation of `nova lv-set-io-priority`

5.4 Network Bandwidth Reallocation

5.4.1 Priority qdisc (prio) based Reallocation

> Note: the most recent version of the nova API extension features the htb qdisc approach (see next section). Method names such as `tc-nova-set-net-priorities` do not exist anymore in the most recent version!

Figure 5.8a shows the starting point to evaluate the deprecated `nova tc-set-net-priorites` method which allowed to change network priorities of several machines using the prio qdisc. It shows that the three VMs have no network priority and they all use the same amount of network bandwidth.

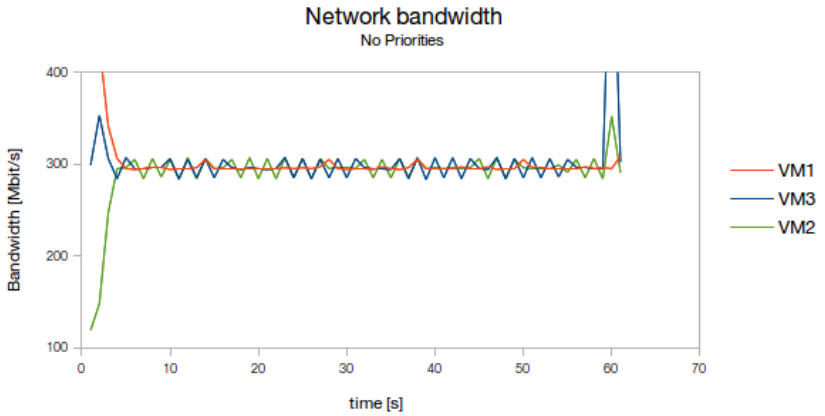
Figure 5.8b shows the network bandwidth of the three VMs, when a very large qdisc size is chosen (600 packets). VM1 and VM2 traffic should have higher priority than VM3 traffic. Figure 5.8b demonstrates that this is not the case. VM1 is starved whereas VM3 is able to send packets constantly.

Figure 5.8c shows a very similar picture. In this example, the priorities are also not respected. It is not ideal that VM1 uses all the bandwidth for itself, because VM1 and VM2 both have priority 1.

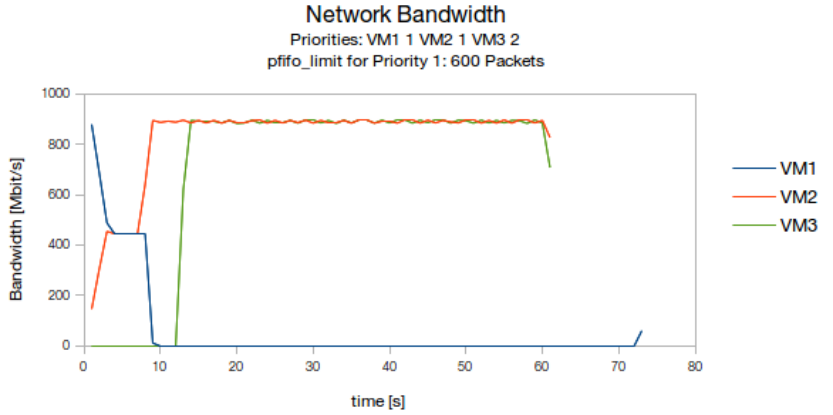
Because the tests with a large qdisc size were inconclusive, it is advised not to set a qdisc size that large. Figure 5.9a shows a test with `pfifo_limit` 30. A qdisc length around 20 to 30 packets has been found as ideal. VM1 and VM2 are able to send packets, because they have both priority 1. VM3 is starved because VM1 and VM2 are constantly sending new packets which will always have priority before VM3 packets.

The traffic of the two prioritized VMs in Figure 5.9b with `pfifo_limit=20` is less bursty than in Figure 5.9a where `pfifo_limit=30`. Figure 5.9c shows that a very short qdisc is good, if the VMs with lower priority should not be starved. But unfortunately the priorities are not respected anymore. Especially after second 35, where VM3 uses most of the bandwidth and VM1 and VM2 only very little. This example shows, that a very short qdisc is also not ideal.

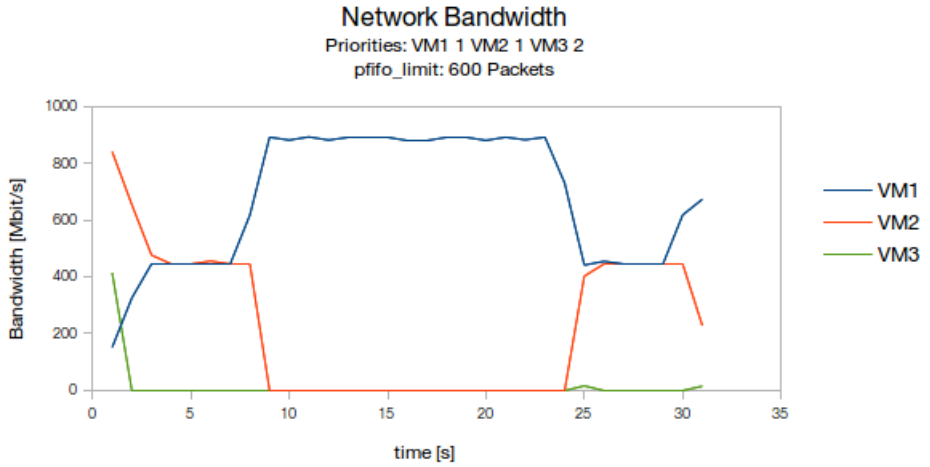
Figure 5.8: Network bandwidth priority test using `nova tc-set-net-priorities`



(a) The starting point to evaluate network bandwidth priorities with `nova tc-set-net-priorities`.

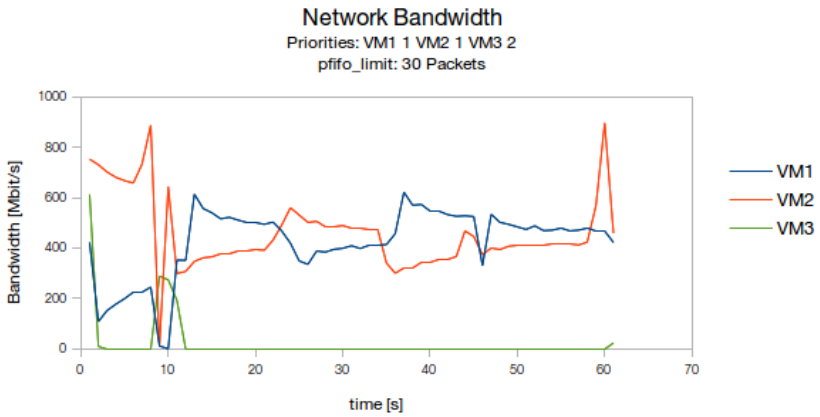


(b) The network priorities have been set with `nova tc-set-net-priorities 'VM1,VM2,VM3' '1,1,2'`. `pfifo_limit` is 600 packets.

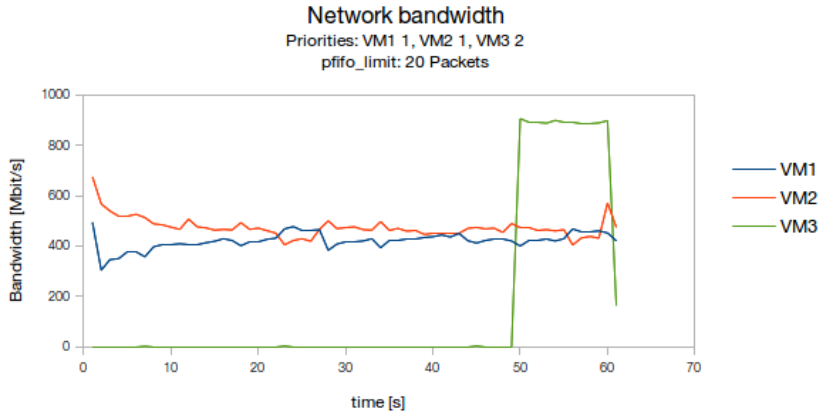


(c) The network priorities have been set with `nova tc-set-net-priorities 'VM1,VM2,VM3' '1,1,2'`. `pfifo_limit` is 600 packets.

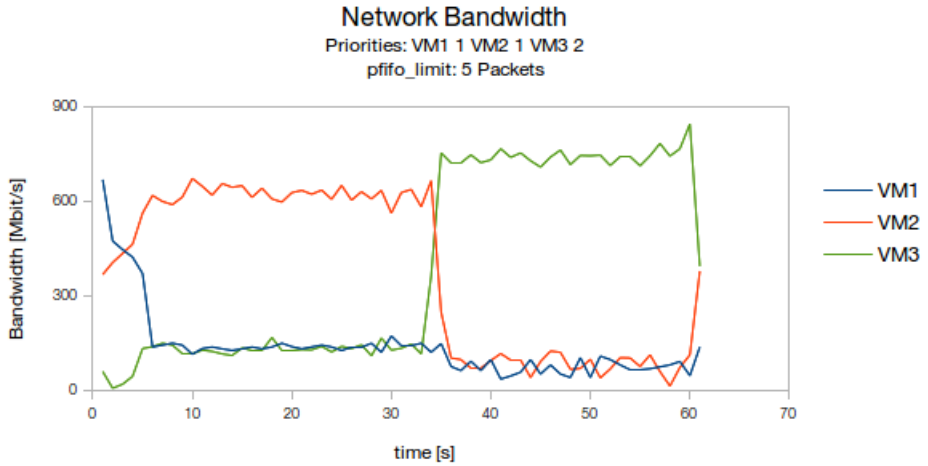
Figure 5.9: Network bandwidth priority test using `nova tc-set-net-priorities`



(a) The network priorities have been set with `nova tc-set-net-priorities 'VM1,VM2,VM3' '1,1,2'`. `pfifo_limit` is 30 packets.



(b) The network priorities have been set with `nova tc-set-net-priorities 'VM1,VM2,VM3' '1,1,2'`. `pfifo_limit` is 20 packets.



(c) The network priorities have been set with `nova tc-set-net-priorities 'VM1,VM2,VM3' '1,1,2'`. `pfifo_limit` is very small (qdisc is only 5 packets long).

5.4.2 Hierarchy Token Bucket qdisc (htb) based Reallocation

Figure 5.10 demonstrates the effect of a missing *ceil* paramter on the htb qdisc. In this case, *ceil* is set to *rate* and it is impossible to borrow excess bandwidth. Figure 5.11 shows the result of a measurement when network traffic is prioritized with the htb qdisc as it is done in the most current version of the API extension.

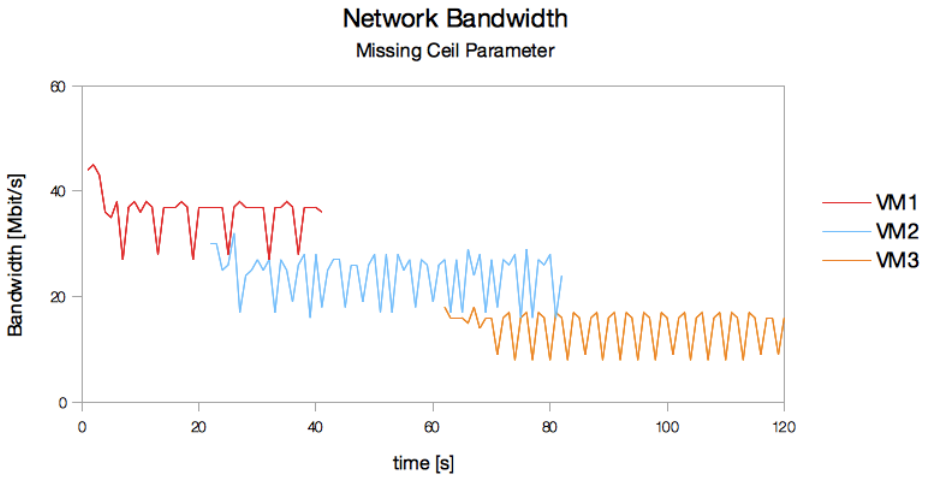


Figure 5.10: htb provides the *ceil* parameter to allow borrowing of excess bandwidth. Without explicitely setting the parameter to a higher value than the minimuman guaranteed *rate*, borrowing will not work properly. The rate priorities for the three VMs where: VM1 39mbit, VM2 26mbit, VM3 15mbit.

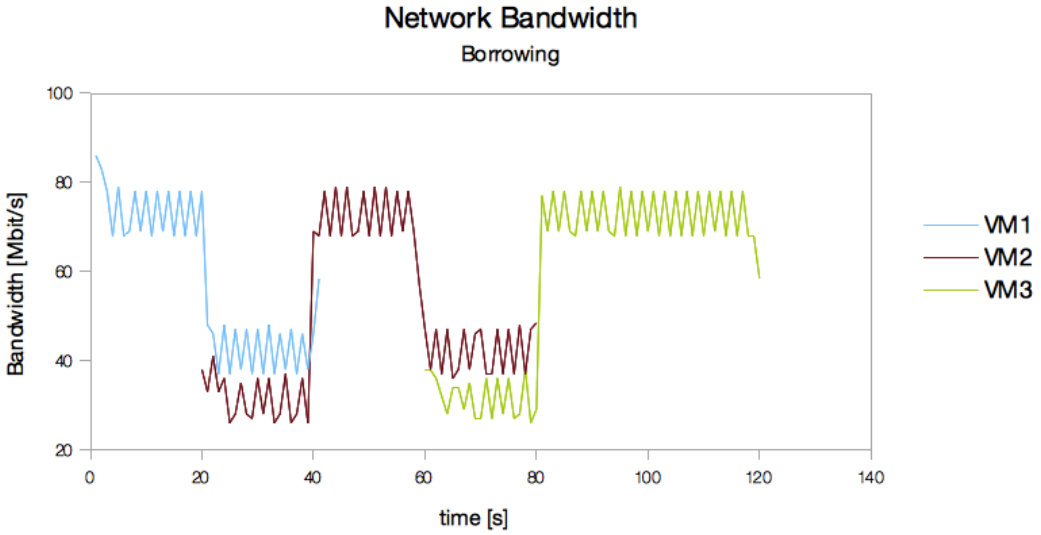


Figure 5.11: This illustration shows the effect of using the *ceil* parameter and setting it to 80mbit. The rate priorities for the VMs are: VM1 39mbit, VM2 26mbit, VM3 15mbit. The excess bandwidth is always used when only one VM is active on the network or in the case, where VM2 und VM3 are able to use more than their minimum guaranteed bandwidth (second 60 to 80).

Chapter 6

Conclusions

6.1 Future Work

This section lists some future improvements and mentions missing functionality and possible weaknesses of the extension. The extension is built to control the PR allocation to VMs of individual PMs. It serves this conceptual formulation of the thesis task very well as shown in the evaluation. Even though the OpenStack extension exposes some reallocation functionality of the hypervisors on the orchestration layer, the extension written during this thesis cannot be considered a complete tool set. Especially not, if it is compared with the current set of reallocation tools available on the hypervisor layer.

OpenStack may present VMs of different hosts in the same project dashboard. The extension is not built to reallocate resources of VMs running on different hosts, because the libvirt settings and the resource configurations are bound to one host. This could lead to confusion. To use the extension effectively, it always has to be remembered which VM resides on which host. Changes in resource allocation may otherwise not have any effect.

6.1.1 nova API Extension

Some libvirt parameters were not considered in the implementation because the extension is focused on soft limits. The *current memory*—the memory available for the VM—is the only implemented hard limit. Because the extension has been developed and written in a *single node devstack* environment, the behavior in an environment with multiple hypervisors remains unclear. Devstack does provide the ability to create *multi node* OpenStack testing environments.

The extension does not yet have a method to reset the network priorities. They have to be reset manually with `tc` (`tc qdisc del dev <IFACE>`, where `<IFACE>` is the host NIC name which connects the VMs with the host). A future iteration of the extension may feature a seperate API call to do this.

6.1.2 Python `nova` client Extension

The usability of the `nova` client extension may not be perfect for daily use. A future user may label or arrange the functions and parameters in another way. Also, the more recent *V3 microversion framework* (see Section 4.1) needs to be adopted for the extension to be compliant with future versions of OpenStack. The client extension does not support the use of different units. To set the current memory for example with `nova lv-set-mem`, a value in MiB is expected. A future implementation may allow the user to change the resource parameters using different units as needed. An enhanced version of the extension would also enable the user to change the resource priority of several VMs at once, much like it has been implemented in a earlier version for the network priorities (see Section 4.5.1) or in the recent version for the `nova tc-get-net-priorities` method.

The syntax for setting the resource weights or priorities is not consistent across all four resources. Whereas a high memory soft limit, a high I/O weight or a high CPU share allows to use more of the mentioned resource, a hight bandwidth priority limits the guest to use less network bandwidth. A weight based scale for network bandwidth would be desirable in future versions of the extension, such that the syntax complies with the other three resources.

6.2 Summary and Conclusion

This thesis reviews the different resource reallocation capabilites on linux systems in general and especially in virtualized environments. Over the course of this thesis it became clear, that current theoretical methods to reallocate resources are focused on reallocating RAM. Memory ballooning and page sharing are two methods which are very prominent in literature. The assumption has been made, that the importance of this resource comes from its persistence character (see Section 2.3). The tools to reallocate RAM, CPU time, disk I/O and network bandwidth have been assessed. The libvirt API has been found to be the method of choice to reallocate resources in a KVM/OpenStack environment.

An extension for OpenStack has been built during this work, which enables to set resource priorities for VMs running on KVM. Resources can then be reallocated by changing the resource weights of the machines. The libvirt API did already provide useful soft limits for disk I/O, RAM, and CPU time. The methods and parameters offered by the libvirt API have been thoroughly tested and documented in Chapter 3. Because it is not possible to set network bandwidth priorities with the libvirt API, an alternative approach with the `tc` traffic control utility has been proposed and implemented (Section 4.5). During the development of this feature, different `tc` qdiscs have been tested and evaluated. The htb qidsc has been found to be the most reliable qdisc to prioritize network bandwidth. The evaluation of the nova API extension and the client showed, that a memory soft limit is not as timely and precisely as a memory hard limit or a CPU soft limit. Therefore it is less reliable than the hard limit. For using the I/O soft limit, the importance of having the right host I/O scheduler (CFQ) activated has been emphasized.

Bibliography

[1] *API Plugins –- nova 2015.2.0.dev243 documentation*. URL: http://docs.openstack.org/developer/nova/devref/api_plugins.html (visited on May 18, 2015).

[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. "Increasing memory density by using KSM". In: *2009 Linux Symposium*. Vol. 1. Montreal, Quebec, Canada, 2009, pp. 19–28. URL: http://landley.net/kdocs/ols/2009/ols2009-pages-19-28.pdf.

[3] *Beating a dead horse - using CPU affinity - frankdenneman.nl*. URL: http://frankdenneman.nl/2011/01/11/beating-a-dead-horse-using-cpu-affinity/ (visited on June 16, 2015).

[4] *CFQ (Complete Fairness Queueing)*. URL: https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt (visited on July 16, 2015).

[5] *Choose IO Schedulers*. URL: https://www.kernel.org/doc/Documentation/block/switching-sched.txt (visited on July 16, 2015).

[6] *ControlGroupInterface*. 2015. URL: http://www.freedesktop.org/wiki/Software/systemd/ControlGroupInterface/ (visited on Mar. 9, 2015).

[7] *Deadline IO scheduler tunables*. URL: https://www.kernel.org/doc/Documentation/block/deadline-iosched.txt (visited on July 16, 2015).

[8] *DevStack - an OpenStack Community Production –- DevStack 0.0.1.dev6071 documentation*. URL: http://docs.openstack.org/developer/devstack/ (visited on May 18, 2015).

[9] *Flavors - OpenStack Operations Guide*. URL: http://docs.openstack.org/openstack-ops/content/flavors.html (visited on May 18, 2015).

[10] Fei Guo. *Understanding Memory Resource Management in VMware vSphere 5.0*. Performance Study. Palo Alto, CA, USA, 2011, pp. 1–28. URL: http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf.

[11] IBM. *Best Practices for KVM*. Technical Report. Austin, TX, USA, 2010, pp. 1–28. URL: http://www.tdeig.ch/linux/pasche/12_BestPractices_IBM.pdf.

[12] *Index of /doc/Documentation/cgroups*. URL: https://www.kernel.org/doc/Documentation/cgroups/ (visited on Mar. 29, 2015).

[13] *iostat(1) - Linux man page*. URL: http://linux.die.net/man/1/iostat (visited on July 17, 2015).

[14] Beat Kuster. "Exhaustive Assembly of Framework Requirements Necessary to Practically Deploy the Greediness Alignment Algorithm (GAA)". PhD thesis.

[15] *Libcg - Library for Control Groups*. 2015. URL: http://libcg.sourceforge.net/ (visited on Mar. 9, 2015).
[16] *libvirt: Control Groups Resource Management*. 2015. URL: http://libvirt.org/cgroups.html (visited on Feb. 28, 2015).
[17] *libvirt: Domain XML format*. URL: https://libvirt.org/formatdomain.html (visited on Mar. 27, 2015).
[18] *libvirt: Network XML format*. 2015. URL: https://libvirt.org/formatnetwork.html (visited on Mar. 17, 2015).
[19] *libvirt: Wiki: Qemu guest agent*. URL: http://wiki.libvirt.org/page/Qemu_guest_agent (visited on Mar. 25, 2015).
[20] *libvirt: Wiki: VirtualNetworking*. 2015. URL: http://wiki.libvirt.org/page/VirtualNetworking (visited on Mar. 18, 2015).
[21] *Manage resources on overcommitted KVM hosts*. URL: http://www.ibm.com/developerworks/library/l-overcommit-kvm-resources/ (visited on Mar. 27, 2015).
[22] *MEMORY BANDWIDTH: STREAM BENCHMARK PERFORMANCE RESULTS*. URL: https://www.cs.virginia.edu/stream/ (visited on Apr. 20, 2015).
[23] *Memory Hotplug*. 2015. URL: https://www.kernel.org/doc/Documentation/memory-hotplug.txt (visited on Mar. 2, 2015).
[24] *Memory Resource Controller*. URL: https://www.kernel.org/doc/Documentation/cgroups/memory.txt (visited on Mar. 27, 2015).
[25] *Networking concepts - OpenStack Cloud Administrator Guide - current*. URL: http://docs.openstack.org/admin-guide-cloud/content/section_networking-options.html (visited on July 1, 2015).
[26] *orchestration*. URL: https://varchitectthoughts.files.wordpress.com/2013/06/screen-shot-2013-06-24-at-1-19-41-pm.png?w=640 (visited on July 25, 2015).
[27] *Re: [libvirt-users] virtual networking - virbr0-nic interface*. URL: https://www.redhat.com/archives/libvirt-users/2012-September/msg00038.html (visited on Mar. 22, 2015).
[28] Joel H. Schopp, Keir Fraser, and Martine J. Silbermann. "Resizing Memory With Balloons and Hotplug". In: vol. 2.
[29] *smem memory reporting tool*. URL: https://www.selenic.com/smem/ (visited on July 20, 2015).
[30] *STREAM Benchmark Reference Information*. URL: https://www.cs.virginia.edu/stream/ref.html (visited on Apr. 20, 2015).
[31] *systemd*. 2015. URL: http://freedesktop.org/wiki/Software/systemd/ (visited on Mar. 9, 2015).
[32] *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*. Tech. rep. 2007.
[33] *virsh(1): management user interface - Linux man page*. 2015. URL: http://linux.die.net/man/1/virsh (visited on Mar. 13, 2015).
[34] Wm. A. Wulf and Sally A. McKee. *Hitting the Memory Wall: Implications of the Obvious*. Tech. rep. 1994.

Abbreviations

API	Application Programming Interface
CFQ	Complete Fairness Queueing
CLI	Command line interface
CPU	Core processing unit
CSV	Comma-separated values
GB	Gigabyte, 1000^3 bytes
GiB	Gibibyte, 1024^3 bytes
HTB	Hierarchy Token Bucket
IaaS	Infastructure as a Service
IOPS	Input/Output operations per second
IOs	Input/Output operations
IP	Internet protocol
I/O	Input/Output
KB, KByte	Kilobyte, 1000 bytes
KiB	Kibibyte, 1024 bytes
KSM	Kernel Same-Page Merging
KVM	Kernel-based Virtual Machine
LAN	Local Area Network
MAC adress	Media access control address
MB	Megabyte, 1000^2 bytes
Mbit/s	Megabit/second
MiB	Mebibyte, 1024^2 bytes

NAT	Network Address Translation
NIC	Network interface card
PM	Physical machine
PR	Physical resource
PSS	Proportional set size
`tc`	Traffic control
TPS	Transfer per second
VM	Virtual Machine
QEMU	Quick Emulator

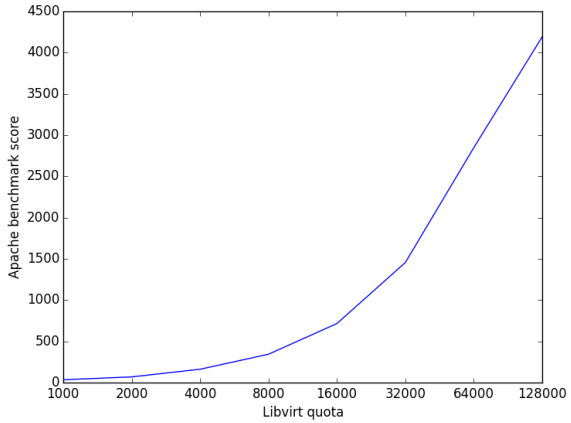
List of Figures

List of Tables

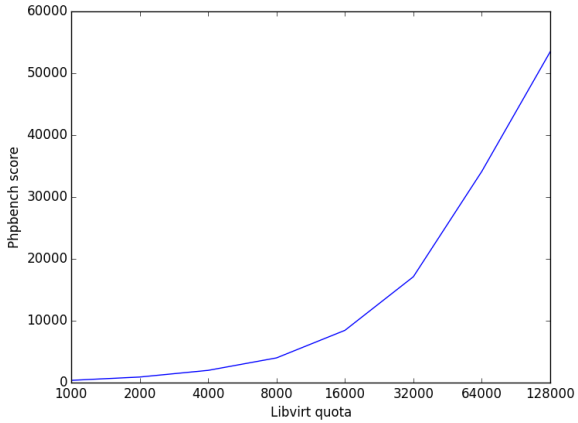
Appendix A

Quota Benchmarks

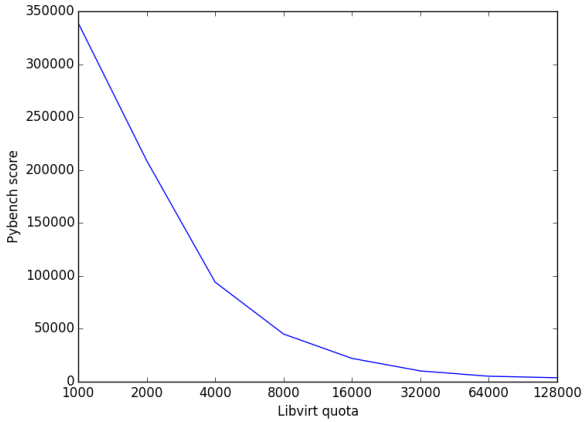
A.1 Memory Bandwidth Performance Analysis



(a) Apache benchmark; The benchmark score increases in memory bandwidth.



(b) PHP benchmark; The benchmark score increases in memory bandwidth.



(c) Python benchmark; The time required to run the benchmark decreases in memory bandwidth(quota).

Appendix B

Disk I/O Measurements with `dd`

B.1 Deadline I/O scheduler

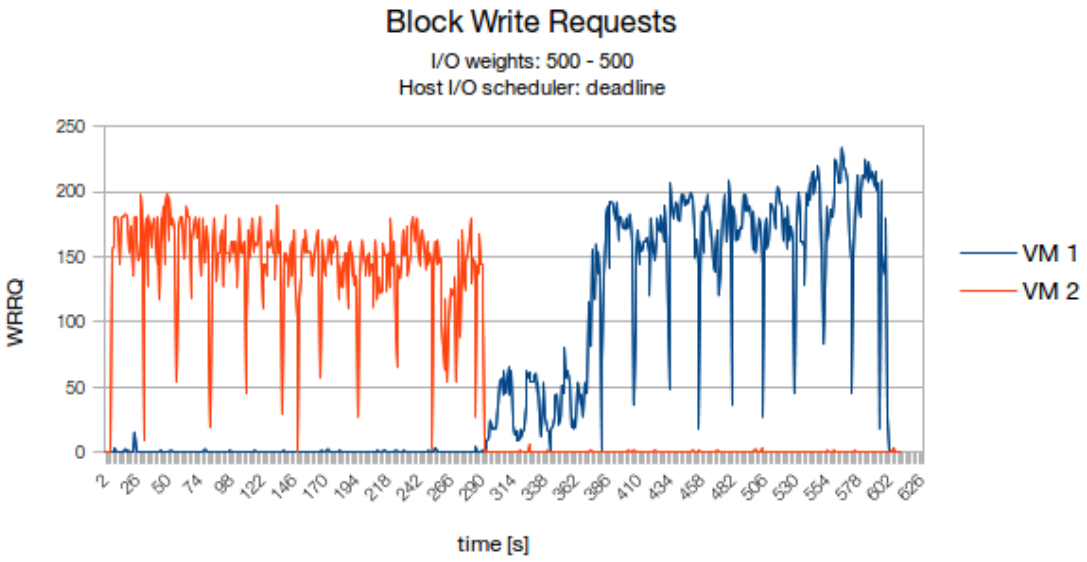


Figure B.1: Block write requests of two VMs running a `dd` test synchronously. The host performs the writes on disk in batches, because the *deadline* scheduler is active. The VMs both have the same libvirt I/O weights (500).

B.2 CFQ I/O scheduler

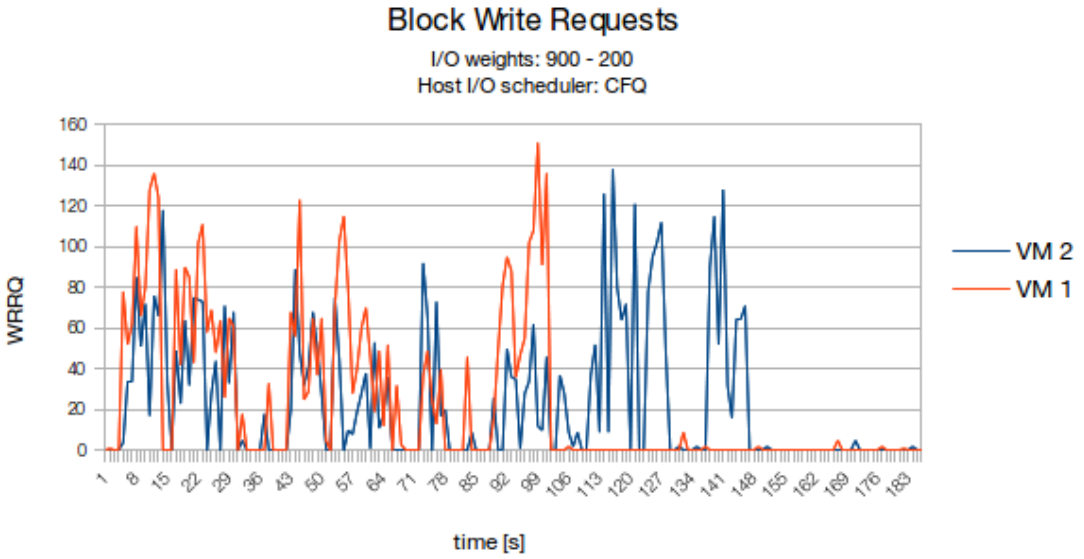


Figure B.2: Block write requests of two VMs running a `dd` test synchronously. The host prioritizes the writes on disk correctly, because the *CFQ* scheduler is active. The VMs have the libvirt I/O weights 900 (VM 1) and 200 (VM 2).

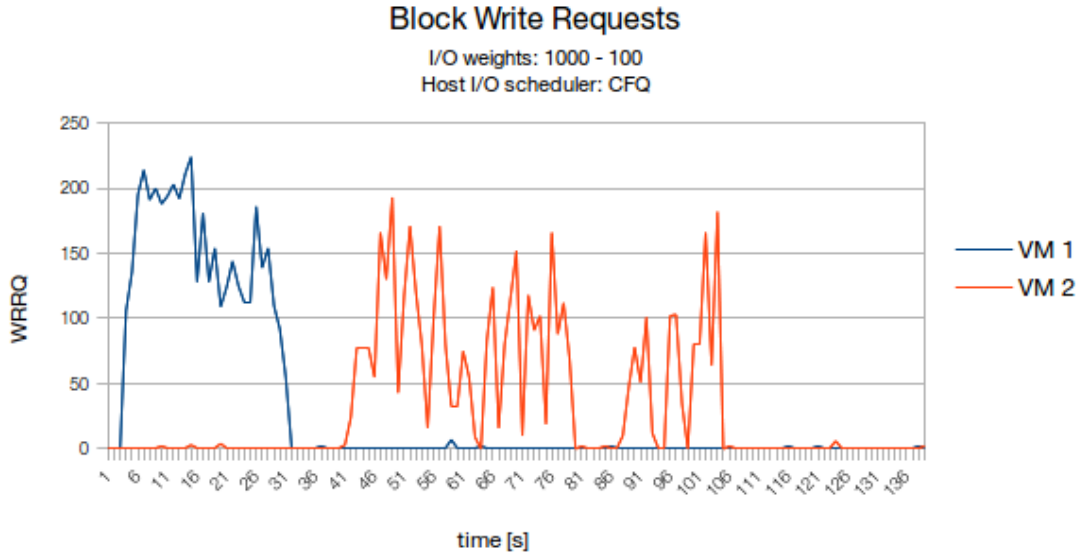


Figure B.3: Block write requests of two VMs running a `dd` test synchronously. The host prioritizes the writes on disk correctly, because the *CFQ* scheduler is active. The VMs have the libvirt I/O weights 1000 (VM 1) and 100 (VM 2).

Appendix C

Libvirt Extension Documentation

The extension to the OpenStack source code contains three parts:

- Section C.1: An extension for the nova API
- Section C.2: An extension for the python-novaclient CLI (`nova`)
- Section C.3: A hook to modify the default memory allocation after instance creation (see Section 4.2).

Those three parts have to be installed seperatly. All parts should be used with the stable OpenStack release *kilo*. The python-novaclient extension and the hook depend both on the nova API extension. The hook depends on the python-novaclient extension. The hook and the novaclient extension are not mandatory, the nova API extension can be used without installing the CLI extension or the hook.

C.1 Nova API Extension

The *servers* resource of the nova API was extended with actions for each resource request. The PR of the machine with `{server_id}` on project `{tenant_id}` can be controlled by sending a POST request to the URI `{tenant_id}/servers/{server_id}/action`. The action to be performed on the servers PR has to be specified in a JSON formatted request body.

Action	JSON Request	Response Body	Description
get-memory-soft-limit		`{ 'soft_limit': 500 }`	Returns the current memory soft limit of an instance in KiB.
set-memory-soft-limit	`{ 'set-memory-soft-limit': { 'memory_limit' : 128 } }`	`{ 'sucess': 1 }`	Allows to set the memory soft limit in MiB. Returns 1 on success, else 0.
get-current-memory		`{ 'actual': 65536 }`	Returns information about the current memory consumption in KiB. The *actual* memory represents the current memory allocated to the VM, which is returned by the `nova lv-get-mem command`.
set-current-memory	`{ 'set-current-memory': { 'memory' : 128 } }`	`{ 'sucess': 1 }`	Allows to set the current memory size in MiB. Returns 1 if ballooning was successful, else 0. *memory* has to be smaller than maximum memory (see Section 3.1).
get-cpu-priority		`{ 'share': 400 }`	Returns the CPU priority of the instance.
set-cpu-priority	`{ 'set-cpu-priority': { 'share' : 400 } }`	`{ 'share': 400 }`	Sets the CPU priority of the instance and returns the current CPU priority. The scale for the *share* value can be chosen by the user (see Section 3.2.3).
get-io-priority		`{ 'weight': 500 }`	Returns the current IO weight of an instance. *weight* is in range $[100, 1000]$.
set-io-priority	`{ 'set-io-priority': { 'weight' : 600 } }`	`{ 'weight': 600 }`	Set the IO weight of an instance. *weight* is in range $[100, 1000]$. Returns the current IO weight.

set-net-priority	`{ 'set-net-priority': { 'prio' : '3' }`	`{ 'success': True }`	Set network priority of an instance. The current qdisc structure on the host will be recreated. Existing priorities remain untouched. *prio* is the new network priority in range [0:98] (more details in Section 4.5.2).
get-net-priority		`{ 'prio': 1 }`	Get network priority of an instance. Returns -1 in the case where no priority is set.
balloon-driver-enabled		`{ 'ballooning_enabled': True }`	Checks if the balloon driver of the instance is ready for use. Returns True if this is the case, else False.

Table C.1: Nova API extension reference

C.2 Python Novaclient Extension

Command	Parameters	Description
`lv-get-mem-soft-limit`	`instance-name`: Name of the instance	Show current memory soft limit of the instance.
`lv-set-mem-soft-limit`	`instance-name`: Name of the instance `memory_limit`: New RAM soft limit in MiB.	Set the memory soft limit for an instance.
`lv-get-mem`	`instance-name`: Name of the instance	Show current memory configuration of instance.
`lv-set-mem`	`instance-name`: Name of the instance `memory`: New memory size in MiB. Value has to be smaller than maximum memory (see Section 3.1).	Set current memory configuration of instance.
`lv-get-cpu-priority`	`instance-name` Name of the instance	Get cpu priority of an instance.
`lv-set-cpu-priority`	`instance-name`: Name of the instance `shares`: New CPU share value	Set cpu priority for the instance.
`lv-get-io-priority`	`instance-name`: Name of the instance	Get I/O disk priority of an instance.
`lv-set-io-priority`	`instance-name`: Name of the instance `weight`: New IO weight value	Set I/O priority for the instance.
`tc-set-net-priority`	`instance-name`: Name of the instance `prio`: New network priority. Must be in range [0,98]. 0 is the highest priority!	Set network priority of an instance. Example: `tc-set-net-priority testvm 43`
`tc-get-net-priorities`	`instance-map`: List of instances, enclosed in quotation marks.	Get network priorities of instance list. Example: `tc-get-net-priorities "vm1,vm2"`

`tc-get-net-priority`	`instance-name`: Name of the instance	Get network priority of an instance. Example: `tc-get-net-priority vm1`
`ballooning-enabled`	`instance-name`: Name of the instance	Check the ballooning capabilities of an instance.

Table C.2: Python novaclient extension reference

C.3 Instance Creation Hook

The hook is only executed, if a `ballooning_ram_ratio` is set in the `[libvirt]` section of the nova configuration at */etc/nova/nova.conf*. After instance creation, the memory of the VM is set to:

$$\frac{flavor.memory\_mb}{libvirt.ballooning\_ram\_ratio}$$

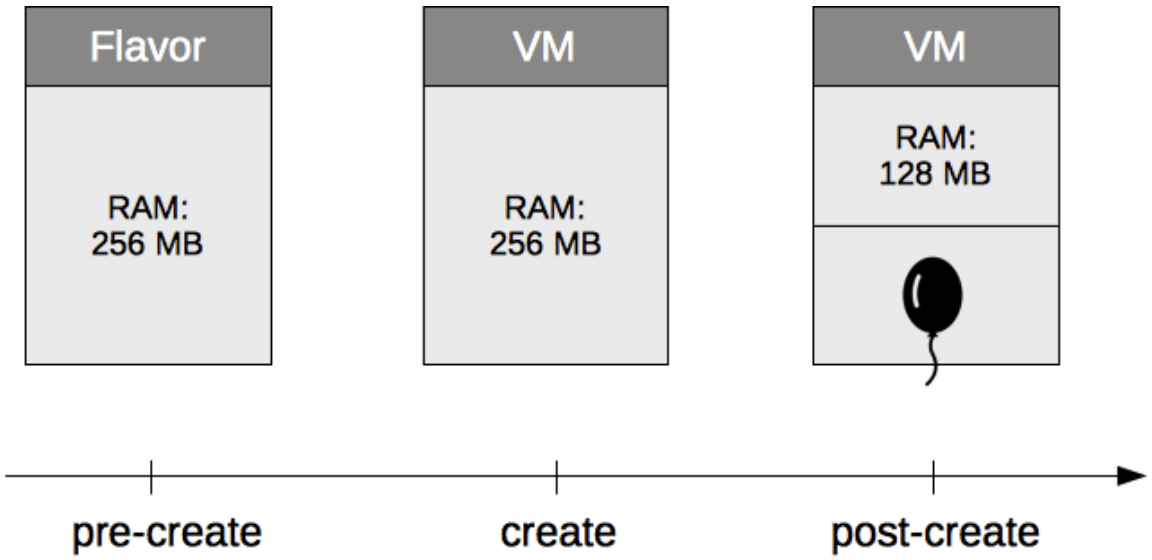


Figure C.1: Logic of the instance creation hook with `ballooning_ram_ratio=2`

There exists no instance start hook. On VM reboot, it will start with the memory size defined in the flavor.

Appendix D

Contents of the CD

The contents of the enclosed CD contains of the following parts:

- Experimental Cloud Setup in *openstack_install_logs*
- Benchmark scripts for disk I/O, CPU quota and memory in *benchmark_scripts*
- Nova API with extension in *ba_nova*
- Novaclient with extension in *ba_novaclient*
- Instance post creation hook for RAM ballooning in *ba_libvirt_resources_hooks*
- A *ba_shortcuts* folder with symbolic links for fast access to the newly written code

```
|-- ba_libvirt_resources_hooks
|-- ba_nova
|-- ba_novaclient
|-- ba_shortcuts
|   |-- libvirt_resources_client.py
|   |   -> ../ba_novaclient/novaclient/v2/contrib/libvirt_resources.py
|   |-- libvirt_resources_ram_hook.py
|   |   -> ../ba_libvirt_resources_hooks/lrhooks/hooks.py
|   |-- libvirt_resources_server.py
|   |   -> ../ba_nova/nova/api/openstack/compute/contrib/libvirt_resources.py
|-- benchmark_scripts
|   |-- blkdeviotune
|   |-- quota
|   |-- smem
|-- openstack_install_logs
```

Listing D.1: Overview of the contents of the enclosed CD