Chapter 15: Performance Tuning and Optimization for KVM VMs

When we're thinking about virtualization, there are always questions that keep coming up. Some of them might be simple enough, such as what are we going to get out of virtualization? Does it simplify things? Is it easier to back up? But there are also much more complex questions that start coming up once we've used virtualization for a while. How do we speed things up on a compute level? Is there a way to do more optimization? What can we tune additionally to get some more speed out of our storage or network? Can we introduce some configuration changes that will enable us to get more out of the existing infrastructure without investing a serious amount of money in it?

That's why performance tuning and optimization is so important to our virtualized environments. As we will find out in this chapter, there are loads of different parameters to consider – especially if we didn't design things properly from the very start, which is usually the case. So, we're going to cover the subject of design first, explain why it shouldn't be just a pure trial-and-error process, and then move on to disassembling that thought process through different devices and subsystems.

In this chapter, we will cover the following topics:

- Tuning VM CPU and memory performance – NUMA
- Kernel same-page merging
- Virtio device tuning
- Block I/O tuning
- Network I/O tuning

It's all about design

There are some fundamental patterns that we constantly repeat in many other aspects of our lives. We usually do so in IT, too. It's completely normal for us not to be good at something when we just start doing it. For example, when we start training in any kind of sport, we're usually not as good as we become after a couple of years of sticking with it. When we start musical training, we're usually much better at it after a couple of years of attending musical school. The same principle applies to IT – when we start doing IT, we're nowhere near as good at it as we become with time and – primarily – *experience*.

We as humans are really good at putting *intellectual defenses* in the way of our learning. We're really good at saying *I'm going to learn through my mistakes* – and we usually combine that with *leave me alone*.

The thing is – there's so much knowledge out there already, it would be silly not to use it. So many people already went through the same or similar process as we did; it would be a pointless exercise in futility *not* to use that experience to our advantage. Furthermore, why waste time on this whole *I'm going to learn through my mistakes* thing when we can learn much more from people with much more experience than us?

When we start using virtualization, we usually start small. For example, we start by installing a hosted-virtualization solution, such as VMware Player, Oracle VirtualBox, or something like that. Then, as time goes by, we move on to a hypervisor with a couple of **Virtual Machines** (**VMs**). As the infrastructure around us grows, we start following linear patterns in trying to make infrastructure work *as it used to, when it was smaller*, which is a mistake. Nothing in IT is linear – growth, cost, the time spent on administration… absolutely nothing. It's actually rather simple to deconstruct that – as environments grow, there are more co-dependencies, which means that one thing influences another, which influences another, and so on. This endless matrix of influences is something that people often forget, especially in the design phase.

Important note:

It's really simple: linear design will get you nowhere, and proper design is the basis of performance tuning, which leaves much less work to be done on performance tuning afterward.

Earlier on in this book (in ***Chapter 2***, *KVM as a Virtualization Solution*), we mentioned **Non-Uniform Memory Access** (**NUMA**). Specifically, we mentioned that the NUMA configuration options are a *very important part of VM configuration, especially if you're designing an environment that hosts loads of virtualized servers.* Let's use a couple of examples to elaborate on this point further. These examples will give us a good basis to take a *mile-high view* of the biggest problem in performance tuning and optimization and describe how to use good design principles to get us out of many different types of trouble. We're going to use Microsoft-based solutions as examples on purpose – not because we're religious about using them, but because of a simple fact. We have a lot of widely available documentation that we can use to our advantage – design documents, best practices, shorter articles, and so on. So, let's use them.

General hardware design

Let's say that you've just started to design your new virtualized environment. When you order servers today from your channel partners – whichever they are – you need to select a model from a big list. It doesn't really matter which brand – there are a lot of models on offer. You can go with **1U** (so-called *pizza box*) servers, which mostly have either one or two CPUs, depending on the model. Then, you can select a **2U** server, a **3U** server…the list gets exponentially bigger. Let's say that you selected a **2U** server with one CPU.

In the next step, you select the amount of memory – let's say 96 GB or 128 GB. You place your order, and a couple of days or weeks later, your server gets delivered. You open it up, and you realize something – all of the RAM is connected to **CPU1** memory channels. You put that in your memory bank, forget about it, and move on to the next phase.

Then, the question becomes about the micro-management of some very pedestrian settings. The BIOS version of the server, the drivers on the hypervisor level, and the BIOS settings (power management, C-states, Turbo Boost, hyper-threading, various memory-related settings, not allowing cores to turn themselves off, and so on) can have a vast influence on the performance of our VMs running on a hypervisor. Therefore, it's definitely best practice to first check whether there are any newer BIOS/firmware versions for our hardware, and check the manufacturer and other relevant documentation to make sure that the BIOS settings are as optimized as possible. Then, and only then, we can start *checkboxing* some physical and deployment procedures – deploying our server in a rack, installing an OS and everything that we need, and start using it.

Let's say that after a while, you realize that you need to do some upgrades and order some PCI Express cards – two single-port Fibre Channel 8 Gbit/s host-based adapters, two single-port 10 Gbit/s Ethernet cards, and two PCI Express NVMe SSDs. For example, by ordering these cards, you want to add some capabilities – to access Fibre Channel storage and to speed up your backup process and VM migrations by switching both of these functionalities from 1 Gbit/s to 10 Gbit/s networking. You place your order, and a couple of days or weeks later, your new PCI Express cards are delivered. You open them up, shut down your server, take it out of the rack, and install these cards. **2U** servers usually have space for two or even three PCI Express riser cards, which are effectively used for connecting additional PCI Express devices. Let's say that you use the first PCI Express riser to deploy the first two cards – the Fibre Channel controllers and 10 Gbit/s Ethernet cards. Then, noticing that you don't have enough PCI Express connectors to connect everything to the first PCI Express riser, you use the second PCI Express riser to install your two PCI Express NVMe SSDs. You screw everything down, close the server cover, put the server back in your rack, and power it back on. Then, you go back to your laptop and connect to your server in a vain attempt to format your PCI Express NVMe SSDs and use them for new VM storage. You realize that your server doesn't recognize these SSDs. You ask yourself – what's going on here? Do I have a bad server?

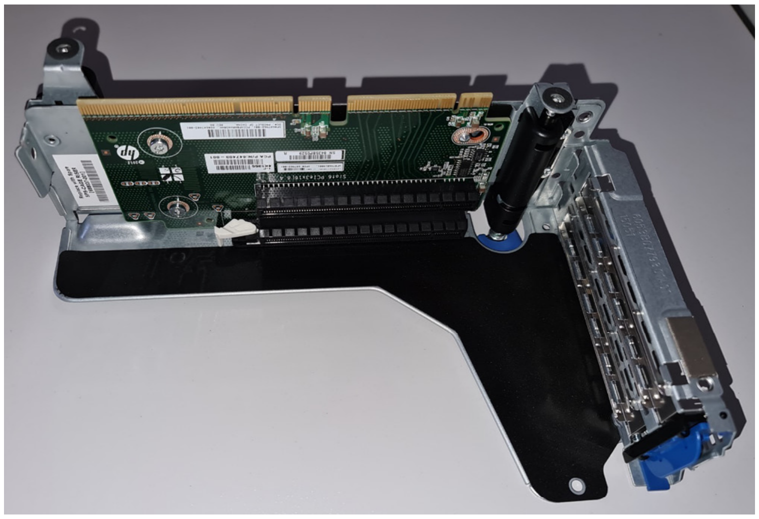


Figure 15.1 – A PCI Express riser for DL380p G8 – you have to insert your PCI Express cards into its slots

You call up your sales rep, and tell them that you think the server is malfunctioning as it can't recognize these new SSDs. Your sales rep connects you to the pre-sales tech; you hear a small chuckle from the other side and the following information: "Well, you see, you can't do it that way. If you want to use the second PCI Express riser on your server, you have to have a CPU kit (CPU plus heatsink) in your second CPU socket, and memory for that second CPU, as well. Order these two things, put them in your server, and your PCI Express NVMe SSDs will work without any problems."

You end your phone conversation and are left with a question mark over your head – *what is going on here? Why do I need to have a second CPU and memory connected to its memory controllers to use some PCI Express cards?*

This is actually related to two things:

- You can't use the memory slots of an uninstalled CPU, as that memory needs a memory controller, which is inside the CPU.
- You can't use PCI Express on an uninstalled CPU, as the PCI Express lanes that connect PCI Express risers' cards to the CPU aren't necessarily provided by the chipset – the CPU can also be used for PCI Express lanes, and it often is, especially for the fastest connections, as you'll learn in a minute.

We know this is confusing; we can feel your pain as we've been there. Sadly, you'll have to stay with us for a little bit longer, as it gets even more confusing.

In ***Chapter 4***, *Libvirt Networking*, we learned how to configure SR-IOV by using an Intel X540-AT2 network controller. We mentioned that we were using the HP ProLiant DL380p G8 server when configuring SR-IOV, so let's use that server for our example here, as well. If you take a look at specifications for that server, you'll notice that it uses an *Intel C600* chipset. If you then go to Intel's ARK website (**https://ark.intel.com**) and search for information about C600, you'll notice that it has five different versions (C602, C602J, C604, C606, and C608), but the most curious part of it is the fact that all of them only support eight PCI Express 2.0 lanes. Keeping in mind that the server specifications clearly state that this server supports PCI Express 3.0, it gets really confusing. How can that be and what kind of trickery is being used here? Yes, PCI Express 3.0 cards can almost always work at PCI Express 2.0 speeds, but it would be misguiding at best to flat-out say that *this server supports PCI Express 3.0*, and then discover that it supports it by delivering PCI Express 2.0 levels of performance (twice as slow per PCI Express lane).

It's only when you go to the HP ProLiant DL380p G8 QuickSpecs document and find the specific part of that document (the *Expansions Slots* part, with descriptions of three different types of PCI Express risers that you can use) where all the information that we need is actually spelled out for us. Let's use all of the PCI Express riser details for reference and explanation. Basically, the primary riser has two PCI Express v3.0 slots that are provided by processor 1 (x16 plus x8), and the third slot (PCI Express 2.0 x8) is provided by the chipset. For the optional riser, it says that all of the slots are provided by the CPU (x16 plus x8 times two). There are actually some models that can have three PCI Express risers, and for that third riser, all of the PCI Express lanes (x16 times two) are also provided by processor 2.

This is all *very important*. It's a huge factor in performance bottlenecks for many scenarios, which is why we centered our example around the idea of two PCI Express NVMe SSDs. We wanted to go through the whole journey with you.

So, at this point, we can have an educated discussion about what should be the de facto standard hardware design of our example server. If our intention is to use these PCI Express NVMe SSDs for local storage for our VMs, then most of us would treat that as a priority. That would mean that we'd absolutely want to connect these devices to the PCI Express 3.0 slot so that they aren't bottlenecked by PCI Express 2.0 speeds. If we have two CPUs, we're probably better off using the *first PCI Express slot* in both of our PCI Express risers for that specific purpose. The reasoning is simple – they're *PCI Express 3.0 compatible* and they're *provided by the CPU*. Again, that's *very important* – it means that they're *directly connected* to the CPU, without the *added latency* of going through the chipset. Because, at the end of the day, the CPU is the central hub for everything, and data going from VMs to SSDs and back will go through the CPU. From a design standpoint, we should absolutely use the fact that we know this to our advantage and connect our PCI Express NVMe SSDs *locally* to our CPUs.

The next step is related to Fibre Channel controllers and 10 Gbit/s Ethernet controllers. The vast load of 8 Gbit/s Fibre Channel controllers are PCI Express 2.0 compatible. The same thing applies to 10 Gbit/s Ethernet adapters. So, it's again a matter of priority. If you're using Fibre Channel storage a lot from our example server, logic dictates that you'd want to put your new and shiny Fibre Channel controllers in the fastest possible place. That would be the second PCI Express slot in both of our PCI Express risers. Again, second PCI Express slots are both provided by CPUs – processor 1 and processor 2. So now, we're just left with 10 Gbit/s Ethernet adapters. We said in our example scenario that we're going to be using these adapters for backup and VM migration. The backup won't suffer all that much if it's done via a network adapter that's on the chipset. VM migration might be a tad sensitive to that. So, you connect your first 10 Gbit/s Ethernet adapter to the third PCI Express slot on the primary riser (for backup, provided by the chipset). Then, you also connect your second 10 Gbit/s Ethernet adapter to the third PCI Express slot on the secondary riser (PCI Express lanes provided by processor 2).

We've barely started on the subject of design with the hardware aspect of it, and already we have such a wealth of information to process. Let's now move on to the second phase of our design – which relates to VM design. Specifically, we're going to discuss how to create new VMs that are designed properly from scratch. However, if we're going to do that, we need to know which application this VM is going to be created for. For that matter, we're going to create a scenario. We're going to use a VM that we're creating to host a node in a Microsoft SQL database cluster on top of a VM running Windows Server 2019. The VM will be installed on a KVM host, of course. This is a task given to us by a client. As we already did the general hardware design, we're going to focus on VM design now.

VM design

Creating a VM is easy – we can just go to **`virt-manager`**, click a couple of times, and we're done. The same applies to oVirt, RedHat Enterprise Virtualization Manager, OpenStack, VMware, and Microsoft virtualization solutions… it's more or less the same everywhere. The problem is designing VMs properly. Specifically, the problem is creating a VM that's going to be pre-tuned to run an application on a very high level, which then only leaves a small number of configuration steps that we can take on the server or VM side to improve performance – the premise being that most of the optimization process later will be done on the OS or application level.

So, people usually start creating a VM in one of two ways – either by creating a VM from scratch with *XYZ* amount of resources added to the VM, or by using a template, which – as we explained in ***Chapter 8***, *Creating and Modifying VM Disks, Templates, and Snapshots* – will save a lot of time. Whichever way we use, there's a certain amount of resources that will be configured for our VM. We then remember what we're going to use this VM for (SQL), so we increase the amount of CPUs to, for example, four, and the amount of memory to 16 GB. We put that VM in the local storage of our server, spool it up, and start deploying updates, configuring the network, and rebooting and generally preparing the VM for the final installation step, which is actually installing our application (SQL Server 2016) and some updates to go along with it. After we're done with that, we start creating our databases and move on to the next set of tasks that need to be done.

Let's take a look at this process from a design and tuning perspective next.

Tuning the VM CPU and memory performance

There are some pretty straightforward issues with the aforementioned process. Some are just engineering issues, while some are more procedural issues. Let's discuss them for a second:

- There is no *one-size-fits-all* solution to almost anything in IT. Every VM of every single client has a different set of circumstances and is in a

different environment that consists of different devices, servers, and so on. Don't try to speed up the process to *impress* someone, as it will most definitely become a problem later.
- When you're done with deployment, stop. Learn the practice of breathe in, breathe out, and stop for a second and think – or wait for an hour or even a day. Remember what you're designing a VM for.
- Before allowing a VM to be used in production, check its configuration. The number of virtual CPUs, the memory, the storage placement, the network options, the drivers, software updates – everything.
- A lot of pre-configuration can be done before the installation phase or during the template phase, before you clone the VM. If it's an existing environment that you're migrating to a new one, *collect information about the old environment*. Find out what the database sizes are, what storage is being used, and how happy people are with the performance of their database server and the applications using them.

At the end of the whole process, learn to take a *mile-high perspective* on the IT-related work that you do. From a quality assurance standpoint, IT should be a highly structured, procedural type of work. If you've done something before, learn to document the things that you did while installing things and the changes that you made. Documentation – as it stands now – is one of the biggest Achilles' heels of IT. Writing documentation will make it easier for you to repeat the process in the future when faced with the same (less often) or a similar (much more often) scenario. Learn from the greats – just as an example, we would know much less about Beethoven, for example, if he didn't keep detailed notes of the things he did day in, day out. Yes, he was born in 1770 and this year will mark 250 years since he was born, and that was a long time ago, but that doesn't mean that 250-year-old routines are bad.

So now, your VM is configured and in production, and a couple of days or weeks later, you get a call from the company and they ask why the performance is *not all that great*. Why isn't it working just like on a physical server?

As a rule of thumb, when you're looking for performance issues on Microsoft SQL, they can be roughly divided into four categories:

- Your SQL database is memory-limited.
- Your SQL database is storage-limited.
- Your SQL database is just misconfigured.
- Your SQL database is CPU-limited.

In our experience, the first and second category can easily account for 80–85% of SQL performance issues. The third would probably account for 10%, while the last one is rather rare, but it still happens. Keeping that in mind, from an infrastructure standpoint, when you're designing a database VM, you should always look into VM memory and storage configuration first, as they are by far the most common reasons. The problems just kind of accumulate and snowball from there. Specifically, some of the most common key reasons for sub-par SQL VM performance is the memory location, looking at it from a CPU perspective, and storage issues – latencies/IOPS and bandwidth being the problem. So, let's describe these one by one.

The first issue that we need to tackle is related to – funnily enough – *geography*. It's very important for a database to have its memory content as close as possible to the CPU cores assigned to its VMs. This is what NUMA is all about. We can easily overcome this specific issue on KVM with a bit of configuration. Let's say that we chose that our VM uses four virtual CPUs. Our test server has Intel Xeon E5-2660v2 processors, which have 10 physical cores each. Keeping in mind that our server has two of these Xeon processors, we have 20 cores at our disposal overall.

We have two basic questions to answer:

- How do these four cores for our VM correlate to 20 physical cores below?
- How does that relate to the VM's memory and how can we optimize that?

The answer to both of these questions is that it depends on *our* configuration. By default, our VM might use two cores from two physical processors each and spread itself in terms of memory across both of them or 3+1. None of these configuration examples are good. What you want is to have all the virtual CPU cores on *one* physical processor, and you want those virtual CPU cores to use memory that's local to those four physical cores – directly connected to the underlying physical processor's memory controller. What we just described is the basic idea behind NUMA – to have nodes (consisting of CPU cores) that act as building compute blocks for your VMs with local memory.

If at all possible, you want to reserve all the memory for that VM so that it doesn't swap somewhere outside of the VM. In KVM, that *outside of the VM* would be in the KVM host swap space. Having access to real RAM memory all of the time is a performance and SLA-related configuration option. If the VM uses a bit of underlying swap partition that acts as its memory, it will not have the same performance. Remember, swapping is usually done on some sort of local RAID array, an SD card, or a similar medium, which are many orders of magnitude slower in terms of bandwidth and latency compared to real RAM memory. If you want a high-level statement about this – avoid memory overcommitment on KVM hosts at all costs. The same goes for the CPU, and this is a commonly used best practice on any other kind of virtualization solution, not just on KVM.

Furthermore, for critical resources, such as a database VM, it definitely makes sense to *pin* vCPUs to specific physical cores. That means that we can use specific physical cores to run a VM, and we should configure other VMs running on the same host *not* to use those cores. That way, we're *reserving* these CPU cores specifically for a single VM, thus configuring everything for maximum performance not to be influenced by other VMs running on the physical server.

Yes, sometimes managers and company owners won't like you because of this best practice (as if you're to blame), as it requires proper planning and enough resources. But that's something that they have to live with – or not, whichever they prefer. Our job is to make the IT system run as best as it possibly can.

VM design has its basic principles, such as the CPU and memory design, NUMA configuration, configuring devices, storage and network configuration, and so on. Let's go through all of these topics step by step, starting with an advanced CPU-based feature that can really help make our systems run as best as possible if used properly – CPU pinning.

CPU pinning

CPU pinning is nothing but the process of setting the *affinity* between the vCPU and the physical CPU core of the host so that the vCPU will be executing on that physical CPU core only. We can use the **`virsh vcpupin`** command to bind a vCPU to a physical CPU core or to a subset of physical CPU cores.

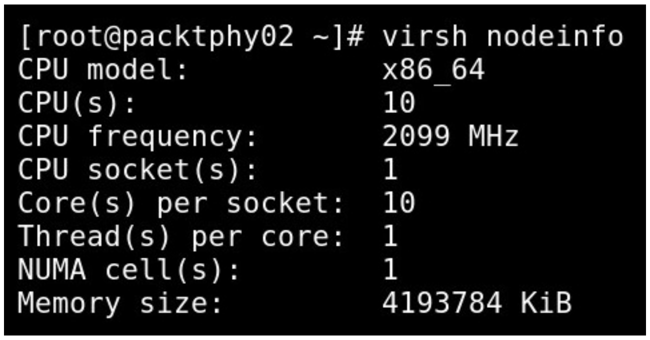
There are a couple of best practices when doing vCPU pinning:

- If the number of guest vCPUs is more than the single NUMA node CPUs, don't go for the default pinning option.

- If the physical CPUs are spread across different NUMA nodes, it is always better to create multiple guests and pin the vCPUs of each guest to physical CPUs in the same NUMA node. This is because accessing different NUMA nodes, or running across multiple NUMA nodes, has a negative impact on performance, especially for memory-intensive applications.

Let's look at the steps of vCPU pinning:

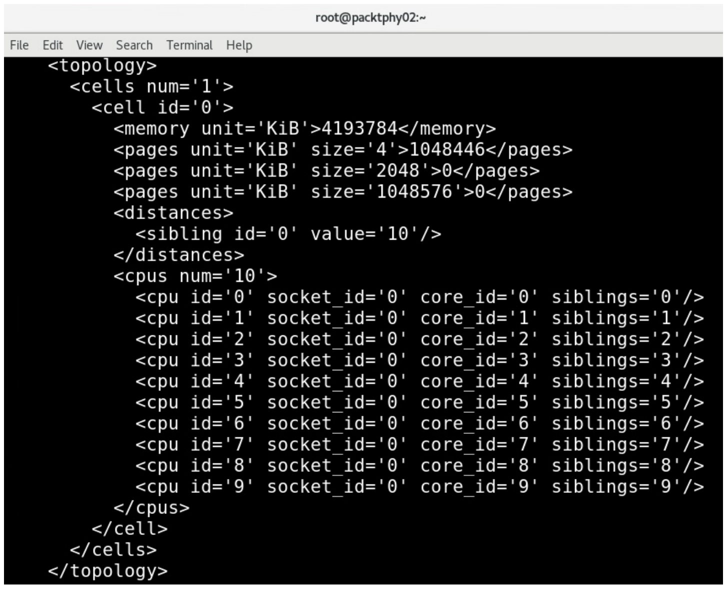
1. Execute **`virsh nodeinfo`** to gather details about the host CPU configuration:



```
[root@packtphy02 ~]# virsh nodeinfo
CPU model:           x86_64
CPU(s):              10
CPU frequency:       2099 MHz
CPU socket(s):       1
Core(s) per socket:  10
Thread(s) per core:  1
NUMA cell(s):        1
Memory size:         4193784 KiB
```

Figure 15.2 – Information about our KVM node

2. The next step is to get the CPU topology by executing the **`virsh capabilities`** command and check the section tagged **`<topology>`**:

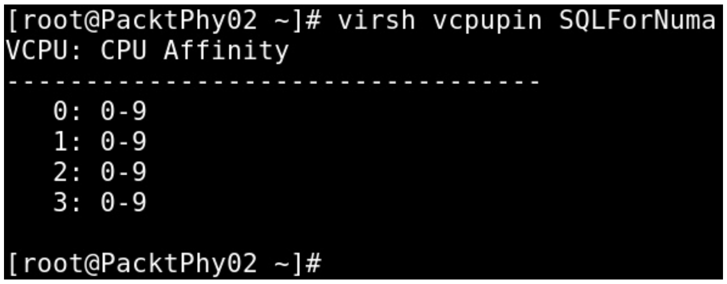


```
    <topology>
      <cells num='1'>
        <cell id='0'>
          <memory unit='KiB'>4193784</memory>
          <pages unit='KiB' size='4'>1048446</pages>
          <pages unit='KiB' size='2048'>0</pages>
          <pages unit='KiB' size='1048576'>0</pages>
          <distances>
            <sibling id='0' value='10'/>
          </distances>
          <cpus num='10'>
            <cpu id='0' socket_id='0' core_id='0' siblings='0'/>
            <cpu id='1' socket_id='0' core_id='1' siblings='1'/>
            <cpu id='2' socket_id='0' core_id='2' siblings='2'/>
            <cpu id='3' socket_id='0' core_id='3' siblings='3'/>
            <cpu id='4' socket_id='0' core_id='4' siblings='4'/>
            <cpu id='5' socket_id='0' core_id='5' siblings='5'/>
            <cpu id='6' socket_id='0' core_id='6' siblings='6'/>
            <cpu id='7' socket_id='0' core_id='7' siblings='7'/>
            <cpu id='8' socket_id='0' core_id='8' siblings='8'/>
            <cpu id='9' socket_id='0' core_id='9' siblings='9'/>
          </cpus>
        </cell>
      </cells>
    </topology>
```

Figure 15.3 – The virsh capabilities output with all the visible physical CPU cores

Once we have identified the topology of our host, the next step is to start pinning the vCPUs.

3. Let's first check the current affinity or pinning configuration with the guest named **`SQLForNuma`**, which has four vCPUs:

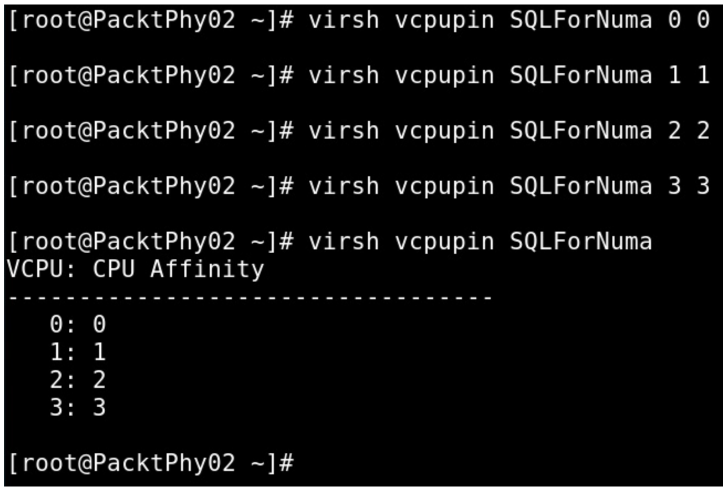


```
[root@PacktPhy02 ~]# virsh vcpupin SQLForNuma
VCPU: CPU Affinity
----------------------------------
   0: 0-9
   1: 0-9
   2: 0-9
   3: 0-9

[root@PacktPhy02 ~]#
```

Figure 15.4 – Checking the default vcpupin settings

Let's change that by using CPU pinning.

4. Let's pin **`vCPU0`** to physical core 0, **`vCPU1`** to physical core 1, **`vCPU2`** to physical core 2, and **`vCPU3`** to physical core 3:

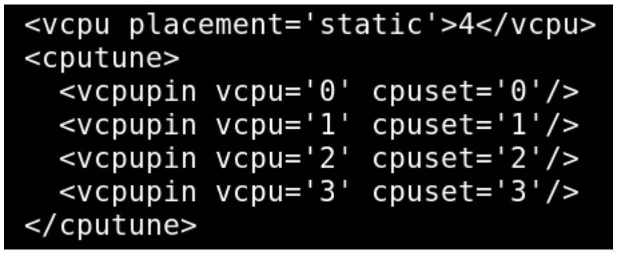


```
[root@PacktPhy02 ~]# virsh vcpupin SQLForNuma 0 0

[root@PacktPhy02 ~]# virsh vcpupin SQLForNuma 1 1

[root@PacktPhy02 ~]# virsh vcpupin SQLForNuma 2 2

[root@PacktPhy02 ~]# virsh vcpupin SQLForNuma 3 3

[root@PacktPhy02 ~]# virsh vcpupin SQLForNuma
VCPU: CPU Affinity
----------------------------------
   0: 0
   1: 1
   2: 2
   3: 3

[root@PacktPhy02 ~]#
```

Figure 15.5 – Configuring CPU pinning

By using **`virsh vcpupin`**, we changed a fixed virtual CPU allocation for this VM.

5. Let's use **`virsh dumpxml`** on this VM to check the configuration change:



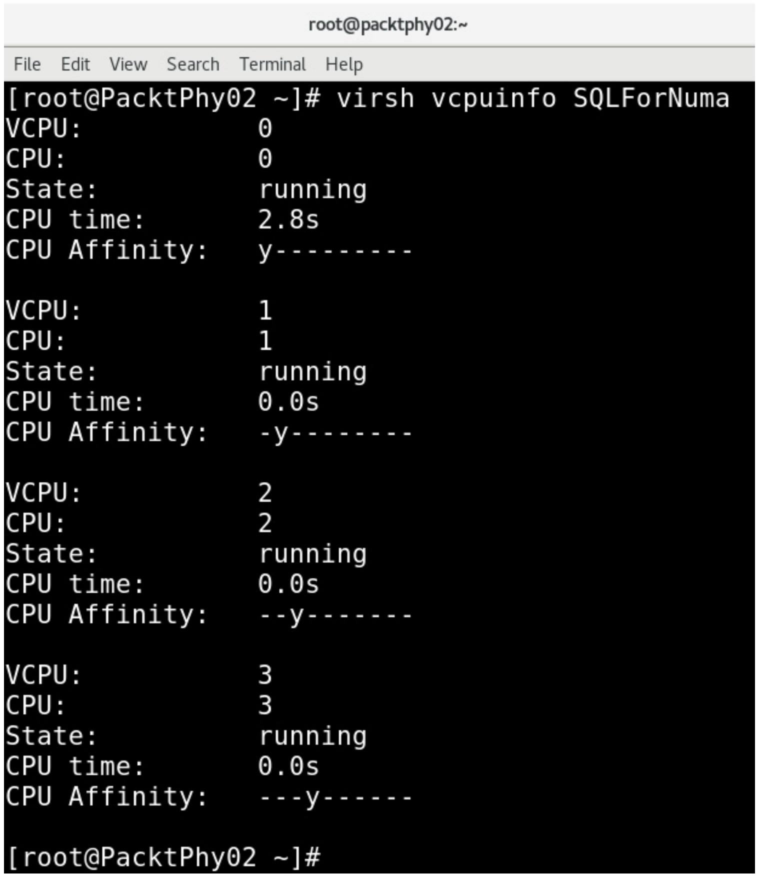
```
<vcpu placement='static'>4</vcpu>
<cputune>
  <vcpupin vcpu='0' cpuset='0'/>
  <vcpupin vcpu='1' cpuset='1'/>
  <vcpupin vcpu='2' cpuset='2'/>
  <vcpupin vcpu='3' cpuset='3'/>
</cputune>
```

Figure 15.6 – CPU pinning VM configuration changes

Notice the CPU affinity listed in the **`virsh`** command and the **`<cputune>`** tag in the XML dump of the running guest. As the XML tag says, this comes under the CPU tuning section of the guest. It is also possible to configure a set of physical CPUs for a particular vCPU instead of a single physical CPU.

There are a couple of things to remember. vCPU pinning can improve performance; however, this depends on the host configuration and the other settings on the system. Make sure you do enough tests and validate the settings.

You can also make use of **`virsh vcpuinfo`** to verify the pinning. The output of the **`virsh vcpuinfo`** command is as follows:



```
[root@PacktPhy02 ~]# virsh vcpuinfo SQLForNuma
VCPU:           0
CPU:            0
State:          running
CPU time:       2.8s
CPU Affinity:   y---------

VCPU:           1
CPU:            1
State:          running
CPU time:       0.0s
CPU Affinity:   -y--------

VCPU:           2
CPU:            2
State:          running
CPU time:       0.0s
CPU Affinity:   --y-------

VCPU:           3
CPU:            3
State:          running
CPU time:       0.0s
CPU Affinity:   ---y------

[root@PacktPhy02 ~]#
```

Figure 15.7 – virsh vcpuinfo for our VM

If we're doing this on a busy host, it will have consequences. Sometimes, we literally won't be able to start our SQL machine because of these settings. So, for the greater good (the SQL VM working instead of not wanting to start), we can change the memory mode configuration from **`strict`** to **`interleave`** or **`preferred`**, which will relax the insistence on using strictly local memory for this VM.

Let's now explore the memory tuning options as they are the next logical thing to discuss.

Working with memory

Memory is a precious resource for most environments, isn't it? Thus, the efficient use of memory should be achieved by tuning it. The first rule in optimizing KVM memory performance is not to allocate more resources to a guest during setup than it will use.

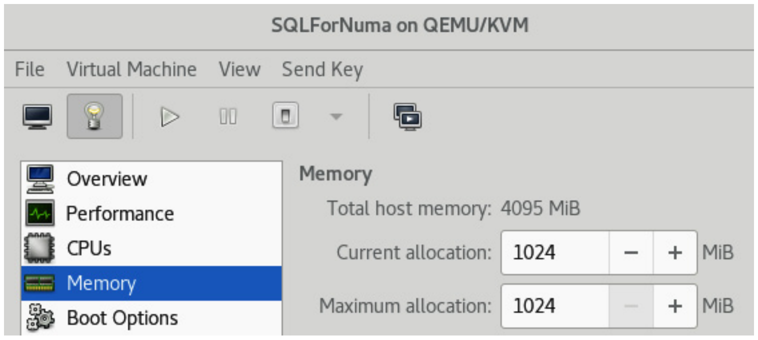
We will discuss the following in greater detail:

- Memory allocation
- Memory tuning
- Memory backing

Let's start by explaining how to configure memory allocation for a virtual system or guest.

Memory allocation

To make the allocation process simple, we will consider the **`virt-manager`** libvirt client again. Memory allocation can be done from the window shown in the following screenshot:



Total host memory: 4095 MiB
Current allocation: 1024 MiB
Maximum allocation: 1024 MiB

Figure 15.8 – VM memory options

As you can see in the preceding screenshot, there are two main options: **Current allocation** and **Maximum allocation**:

- **Maximum allocation**: The runtime maximum memory allocation of the guest. This is the maximum memory that can be allocated to the guest when it's running.
- **Current allocation**: How much memory a guest always uses. For memory ballooning rea-

sons, we can have this value lower than the maximum.

The **virsh** command can be used to tune these parameters. The relevant **virsh** command options are **setmem** and **setmaxmem**.

Memory tuning

The memory tuning options are added under **<memtune>** of the guest configuration file.

Additional memory tuning options can be found at http://libvirt.org/formatdomain.html#elementsMemoryTuning.

The admin can configure the memory settings of a guest manually. If the **<memtune>** configuration is omitted, the default memory settings apply for a guest. The **virsh** command at play here is as follows:

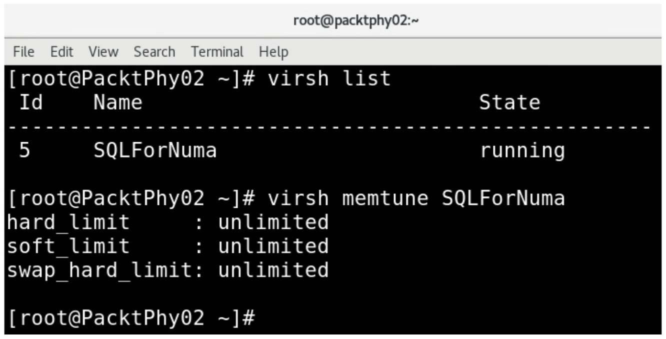
```
# virsh memtune <virtual_machine> --
parameter size parameter
```

It can have any of the following values; this best practice is well documented in the man page:

```
--hard-limit       The maximum memory
the guest can use.
--soft-limit       The memory limit
to enforce during memory contention.
--swap-hard-limit  The maximum memory
plus swap the guest can use.  This
has to be more than hard-limit value
provided.
--min-guarantee    The guaranteed
minimum memory allocation for the
```

```
guest.
```

The default/current values that are set for the **`memtune`** parameter can be fetched as shown:

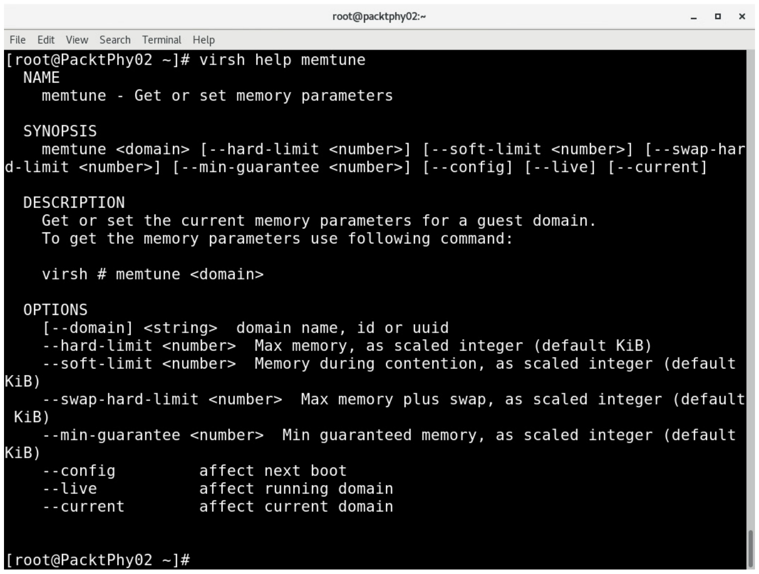


```
[root@PacktPhy02 ~]# virsh list
 Id    Name                           State
----------------------------------------------------
 5     SQLForNuma                     running

[root@PacktPhy02 ~]# virsh memtune SQLForNuma
hard_limit     : unlimited
soft_limit     : unlimited
swap_hard_limit: unlimited

[root@PacktPhy02 ~]#
```

Figure 15.9 – Checking the memtune settings for the VM

When setting **`hard_limit`**, you should not set this value too low. This might lead to a situation in which a VM is terminated by the kernel. That's why determining the correct amount of resources for a VM (or any other process) is such a design problem. Sometimes, designing things properly seems like dark arts.

To learn more about how to set these parameters, please see the help output for the **`memtune`** command in the following screenshot:



```
[root@PacktPhy02 ~]# virsh help memtune
  NAME
    memtune - Get or set memory parameters

  SYNOPSIS
    memtune <domain> [--hard-limit <number>] [--soft-limit <number>] [--swap-har
d-limit <number>] [--min-guarantee <number>] [--config] [--live] [--current]

  DESCRIPTION
    Get or set the current memory parameters for a guest domain.
    To get the memory parameters use following command:

    virsh # memtune <domain>

  OPTIONS
    [--domain] <string>  domain name, id or uuid
    --hard-limit <number>  Max memory, as scaled integer (default KiB)
    --soft-limit <number>  Memory during contention, as scaled integer (default
KiB)
    --swap-hard-limit <number>  Max memory plus swap, as scaled integer (default
 KiB)
    --min-guarantee <number>  Min guaranteed memory, as scaled integer (default
KiB)
    --config         affect next boot
    --live           affect running domain
    --current        affect current domain


[root@PacktPhy02 ~]#
```

Figure 15.10 – Checking virsh help memtune

As we have covered memory allocation and tuning, the final option is memory backing.

Memory backing

The following is the guest XML representation of memory backing:

```
<domain>     ...
  <memoryBacking>
    <hugepages>
    <page size="1" unit="G"
nodeset="0-3,5"/>
    <page size="2" unit="M"
nodeset="4"/>
    </hugepages>
    <nosharepages/>
    <locked/>
</memoryBacking>     ...
  </domain>
```

You may have noticed that there are three main options for memory backing: **`locked`**, **`nosharepages`**, and **`hugepages`**. Let's go through them one by one, starting with **`locked`**.

locked

In KVM virtualization, guest memory lies in the process address space of the **`qemu-kvm`** process in the KVM host. These guest memory pages can be swapped out by the Linux kernel at any time, based on the requirement that the host has, and this is where **`locked`** can help. If you set the memory backing option of the guest to **`locked`**, the host will not swap out memory pages that belong to the virtual system or guest. The virtual memory pages in the host system memory are locked when this option is enabled:

```
<memoryBacking>
    <locked/>
</memoryBacking>
```

We need to use **`<memtune>`** to set **`hard_limit`**. The calculus is simple – whatever the amount of memory for the guest we need plus overhead.

nosharepages

The following is the XML representation of **`nosharepages`** from the guest configuration file:

```
<memoryBacking>
    <nosharepages/>
</memoryBacking>
```

There are different mechanisms that can enable the sharing of memory when the memory pages are identical. Techniques such as **Kernel Same-Page Merging** (**KSM**) share pages among guest systems. The **`nosharepages`** option instructs the hypervisor to disable shared pages for this guest – that is, setting this option will prevent the host from deduplicating memory between guests.

hugepages

The third and final option is **`hugepages`**, which can be represented in XML format, as follows:

```
<memoryBacking>
</hugepages>
</memoryBacking>
```

HugePages were introduced in the Linux kernel to improve the performance of memory management. Memory is managed in blocks known as pages. Different architectures (i386, ia64) support different page sizes. We don't necessarily have to use the default setting for x86 CPUs (4 KB memory pages), as we can use larger memory pages (2 MB to 1 GB), a feature that's called HugePages. A part of the CPU called the **Memory Management Unit** (**MMU**) manages these pages by using a list. The pages are referenced through page tables, and each page has a reference in the page table. When a system wants to handle a huge amount of memory, there are mainly two options. One of them involves increasing the number of page table entries in the hardware MMU. The second method increases the default page size. If we opt for the first method of increasing the page table entries, it is really expensive.

The second and more efficient method when dealing with large amounts of memory is using HugePages or increased page sizes by using HugePages. The different amounts of memory that each and every server has means that there is a need for different page sizes. The default values are okay for most situations, while huge memory pages (for example, 1 GB) are more efficient if we have large amounts of memory (hundreds of gigabytes or even terabytes). This means less *administrative* work in terms of referencing memory pages and more time spent actually getting the content of these memory pages, which can lead to a significant performance boost. Most of the known Linux distributions can use HugePages to manage large memory amounts. A process can use HugePages memory support to improve performance by increasing the CPU cache hits against the **Translation LookAside Buffer** (**TLB**), as explained in ***Chapter 2***, *KVM as a Virtualization Solution*. You already know that guest systems are simply processes in a Linux system, thus the KVM guests are eligible to do the same.

Before we move on, we should also mention **Transparent HugePages** (**THP**). THP is an abstraction layer that automates the HugePages size allocation based on the application request. THP support can be entirely disabled, can only be enabled inside **`MADV_HUGEPAGE`** regions (to avoid the risk of consuming more memory resources), or enabled system-wide. There are three main options for configuring THP in a system: **`always`**, **`madvise`**, and **`never`**:

```
#
cat/sys/kernel/mm/transparent_hugepag
e/enabled [always] madvise never
```

From the preceding output, we can see that the current THP setting in our server is **`madvise`**. Other options can be enabled by using one of the following commands:

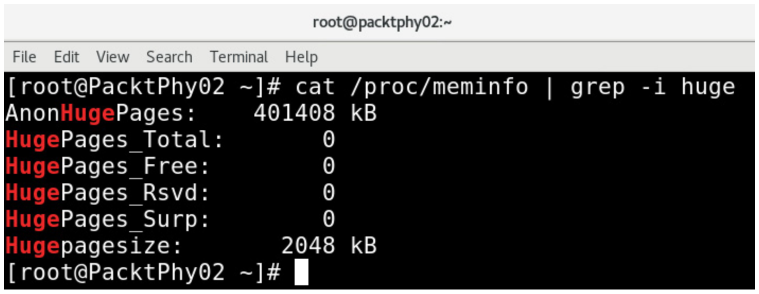
```
echo always
>/sys/kernel/mm/transparent_hugepage/
enabled
echo madvise
>/sys/kernel/mm/transparent_hugepage/
enabled
echo never
>/sys/kernel/mm/transparent_hugepage/
enabled
```

In short, what these values mean is the following:

- `always`: Always use THP.
- `madvise`: Use HugePages only in **Virtual Memory Areas (VMAs)** marked with `MADV_HUGEPAGE`.
- `never`: Disable the feature.

The system settings for performance are automatically optimized by THP. We can have performance benefits by using memory as cache. It is possible to use static HugePages when THP is in place or in other words THP won't prevent it from using a static method. If we don't configure our KVM hypervisor to use static HugePages, it will use 4 Kb transparent HugePages. The advantages we get from using HugePages for a KVM guest's memory are that less memory is used for page tables and TLB misses are reduced; obviously, this increases performance. But keep in mind that when using HugePages for guest memory, you can no longer swap or balloon guest memory.

Let's have a quick look at how to use static HugePages in your KVM setup. First, let's check the current system configuration – it's clear that the HugePages size in this system is currently set at 2 MB:



```
[root@PacktPhy02 ~]# cat /proc/meminfo | grep -i huge
AnonHugePages:    401408 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
[root@PacktPhy02 ~]#
```

Figure 15.11 – Checking the HugePages settings

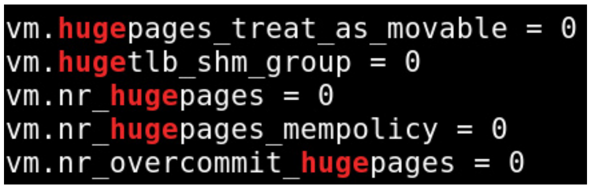
We're primarily talking about all the attributes starting with HugePages, but it's worth mentioning what the **AnonHugePages** attribute is. The **AnonHugePages** attribute tells us the current THP usage on the system level.

Now, let's configure KVM to use a custom HugePages size:

1. View the current explicit **hugepages** value by running the following command or fetch it from **sysfs**, as shown:

   ```
   #  cat /proc/sys/vm/nr_hugepages
   0
   ```

2. We can also use the **sysctl -a |grep huge** command:

   ```
   vm.hugepages_treat_as_movable = 0
   vm.hugetlb_shm_group = 0
   vm.nr_hugepages = 0
   vm.nr_hugepages_mempolicy = 0
   vm.nr_overcommit_hugepages = 0
   ```

   Figure 15.12 – The sysctl hugepages settings

3. As the HugePage size is 2 MB, we can set hugepages in increments of 2 MB. To set the number of hugepages to 2,000, use the following command:

   ```
   # echo 2000 >
   /proc/sys/vm/nr_hugepages
   ```

   The total memory assigned for hugepages cannot be used by applications that are not hugepage-aware – that is, if you over-allocate hugepages, normal operations of the host system can be affected. In our examples, 2048*2 MB would equal 4,096 MB of memory, which we should have available when we do this configuration.

4. We need to tell the system that this type of configuration is actually OK and configure **/etc/security/limits.conf** to reflect that. Otherwise, the system might refuse to give us access to 2,048 hugepages times 2 MB of memory. We need to add two lines to that file:

   ```
   soft memlock <value>
   hard memlock <value>
   ```

   The **<value>** parameter will depend on the configuration we want to do. If we want to configure everything according to our 2048*2 MB example, **<value>** would be 4,194,304 (or 4096*1024).

5. To make it persistent, you can use the following:

   ```
   # sysctl -w vm.nr_hugepages=<number
   of hugepages>
   ```

6. Then, mount the **fs** hugepages, reconfigure the VM, and restart the host:

   ```
   # mount -t hugetlbfs hugetlbfs
   /dev/hugepages
   ```

Reconfigure the HugePage-configured guest by adding the following settings in the VM configuration file:

```
<memoryBacking>
</hugepages>
</ memoryBacking>
```

It's time to shut down the VM and reboot the host. Inside the VM, do the following:

```
# systemctl poweroff
```

On the host, do the following:

```
# systemctl reboot
```

After the host reboot and the restart of the VM, it will now start using the hugepages.

The next topic is related to sharing memory content between multiple VMs, referred to as KSM. This technology is heavily used to *save* memory. At any given moment, when multiple VMs are powered on the virtualization host, there's a big statistical chance that those VMs have blocks of memory contents that are the same (they have the same contents). Then, there's no reason to store the same contents multiple times. Usually, we refer to KSM as a deduplication process being applied to memory. Let's learn how to use and configure KSM.

Getting acquainted with KSM

KSM is a feature that allows the sharing of identical pages between the different processes running on a system. We might presume that the identical pages exist due to certain reasons—for example, if there are multiple processes spawned from the same binary or something similar. There is no rule such as this though. KSM scans these identical memory pages and consolidates a **Copy-on-Write** (**COW**) shared page. COW is nothing but a mechanism where when there is an attempt to change a memory region that is shared and common to more than one process, the process that requests the change gets a new copy and the changes are saved in it.

Even though the consolidated COW shared page is accessible by all the processes, whenever a process tries to change the content (write to that page), the process gets a new copy with all of the changes. By now, you will have understood that, by using KSM, we can reduce physical memory consumption. In the KVM context, this can really add value, because guest systems are **`qemu-kvm`** processes in the system, and there is a huge possibility that all the VM processes will have a good amount of similar memory.

For KSM to work, the process/application has to register its memory pages with KSM. In KVM-land, KSM allows guests to share identical memory pages, thus achieving an improvement in memory consumption. That might be some kind of application data, a library, or anything else that's used frequently. This shared page or memory is marked as **`copy on write`**. In short, KSM avoids memory duplication and it's really useful when similar guest OSes are present in a KVM environment.

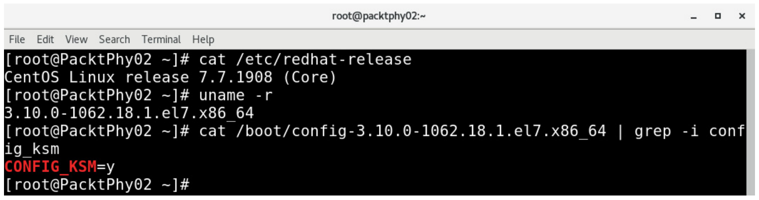
By using the theory of prediction, KSM can provide enhanced memory speed and utilization. Mostly, this common shared data is stored in cache or main memory, which causes fewer cache misses for the KVM guests. Also, KSM can reduce the overall guest memory footprint so that, in a way, it allows the user to do memory overcommitting in a KVM setup, thus supplying the greater utilization of available resources. However, we have to keep in mind that KSM requires more CPU resources to identify the duplicate pages and to perform tasks such as sharing/merging.

Previously, we mentioned that the processes have to mark the *pages* to show that they are eligible candidates for KSM to operate. The marking can be done by a process based on the **`MADV_MERGEABLE`** flag, which we will discuss in the next section. You can explore the use of this flag in the **`madvise`** man page:

```
# man 2 madvise
MADV_MERGEABLE (since Linux 2.6.32)
Enable Kernel Samepage Merging (KSM)
for the pages in the range specified
by addr and length. The kernel
regularly scans those areas of user
memory that have been marked as
mergeable, looking for pages with
identical content.  These are
replaced by a single write-protected
page (that is automatically copied if
a process later wants to update the
content of the page).  KSM merges
only private anonymous pages (see
mmap(2)).
The KSM feature is intended for
applications that generate many
instances of the same data (e.g.,
virtualization systems such as
KVM).  It can consume a lot of
processing   power; use with
care.  See the Linux kernel source
file Documentation/ vm/ksm.txt for
more details.
```

```
The MADV_MERGEABLE and
MADV_UNMERGEABLE operations are
available only if the kernel was
configured with CONFIG_KSM.
```

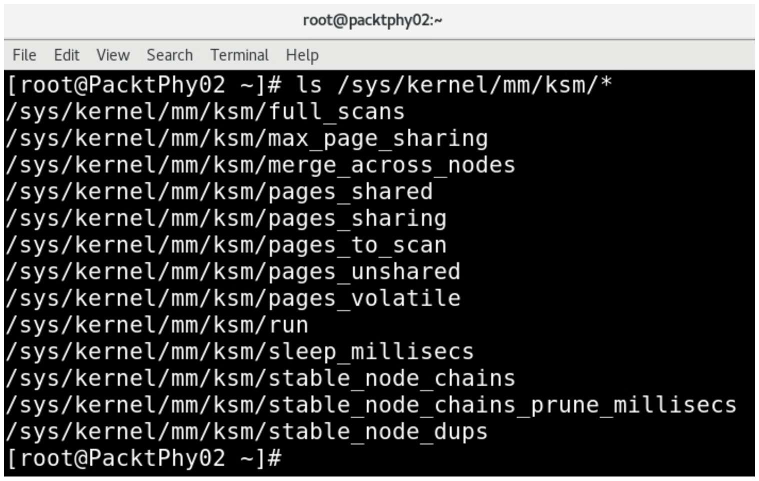
So, the kernel has to be configured with KSM, as follows:



```
[root@PacktPhy02 ~]# cat /etc/redhat-release
CentOS Linux release 7.7.1908 (Core)
[root@PacktPhy02 ~]# uname -r
3.10.0-1062.18.1.el7.x86_64
[root@PacktPhy02 ~]# cat /boot/config-3.10.0-1062.18.1.el7.x86_64 | grep -i conf
ig_ksm
CONFIG_KSM=y
[root@PacktPhy02 ~]#
```

Figure 15.13 – Checking the KSM settings

KSM gets deployed as a part of the **qemu-kvm** package. Information about the KSM service can be fetched from the **sysfs** filesystem, in the **/sys** directory. There are different files available in this location, reflecting the current KSM status. These are updated dynamically by the kernel, and it has a precise record of the KSM usage and statistics:



```
[root@PacktPhy02 ~]# ls /sys/kernel/mm/ksm/*
/sys/kernel/mm/ksm/full_scans
/sys/kernel/mm/ksm/max_page_sharing
/sys/kernel/mm/ksm/merge_across_nodes
/sys/kernel/mm/ksm/pages_shared
/sys/kernel/mm/ksm/pages_sharing
/sys/kernel/mm/ksm/pages_to_scan
/sys/kernel/mm/ksm/pages_unshared
/sys/kernel/mm/ksm/pages_volatile
/sys/kernel/mm/ksm/run
/sys/kernel/mm/ksm/sleep_millisecs
/sys/kernel/mm/ksm/stable_node_chains
/sys/kernel/mm/ksm/stable_node_chains_prune_millisecs
/sys/kernel/mm/ksm/stable_node_dups
[root@PacktPhy02 ~]#
```

Figure 15.14 – The KSM settings in sysfs

In an upcoming section, we will discuss the **ksm-tuned** service and its configuration variables. As **ksmtuned** is a service to control KSM, its configuration variables are analogous to the files we see in the **sysfs** filesystem. For more details, you can check out https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html.

It is also possible to tune these parameters with the **virsh** command. The **virsh node-memory-tune** command does this job for us. For example, the following command specifies the number of pages to scan before the shared memory service goes to sleep:

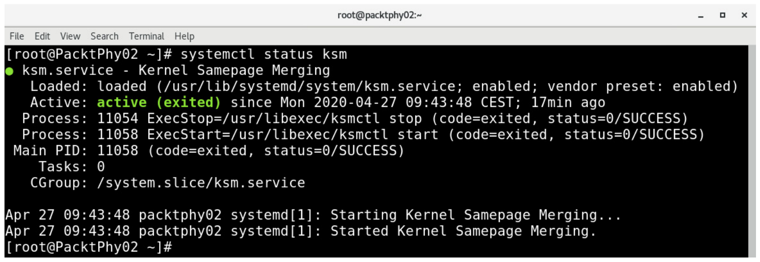
```
# virsh node-memory-tune --shm-pages-to-scan number
```

As with any other service, the **ksmtuned** service also has logs stored in a log file, **/var/log/ksmtuned**. If we add **DEBUG=1** to **/etc/ksmtuned.conf**, we will have logging from any kind of KSM tuning actions. Refer to https://www.kernel.org/doc/Documentation/vm/ksm.txt for more details.

Once we start the KSM service, as shown next, you can watch the values change depending on the KSM service in action:

```
# systemctl start ksm
```

We can then check the status of the **ksm** service like this:



```
[root@PacktPhy02 ~]# systemctl status ksm
● ksm.service - Kernel Samepage Merging
   Loaded: loaded (/usr/lib/systemd/system/ksm.service; enabled; vendor preset: enabled)
   Active: active (exited) since Mon 2020-04-27 09:43:48 CEST; 17min ago
  Process: 11054 ExecStop=/usr/libexec/ksmctl stop (code=exited, status=0/SUCCESS)
  Process: 11058 ExecStart=/usr/libexec/ksmctl start (code=exited, status=0/SUCCESS)
 Main PID: 11058 (code=exited, status=0/SUCCESS)
    Tasks: 0
   CGroup: /system.slice/ksm.service

Apr 27 09:43:48 packtphy02 systemd[1]: Starting Kernel Samepage Merging...
Apr 27 09:43:48 packtphy02 systemd[1]: Started Kernel Samepage Merging.
[root@PacktPhy02 ~]#
```

Figure 15.15 – The ksm service command and the ps command output

Once the KSM service is started and we have multiple VMs running on our host, we can check the changes by querying **sysfs** by using the following command multiple times:

```
cat /sys/kernel/mm/ksm/*
```

Let's explore the **ksmtuned** service in more detail. The **ksmtuned** service is designed so that it goes through a cycle of actions and adjusts KSM. This cycle of actions continues its work in a loop. Whenever a guest system is created or destroyed, libvirt will notify the **ksmtuned** service.

The **/etc/ksmtuned.conf** file is the configuration file for the **ksmtuned** service. Here is a brief explanation of the configuration parameters available. You can see these configuration parameters match with the KSM files in **sysfs**:

```
# Configuration file for ksmtuned.
# How long ksmtuned should sleep between tuning adjustments
# KSM_MONITOR_INTERVAL=60
# Millisecond sleep between ksm scans for 16Gb server.
# Smaller servers sleep more, bigger sleep less.
# KSM_SLEEP_MSEC=10
# KSM_NPAGES_BOOST - is added to the `npages` value, when `free memory` is less than `thres`.
# KSM_NPAGES_BOOST=300
# KSM_NPAGES_DECAY - is the value given is subtracted to the `npages` value, when `free memory` is greater than `thres`.
# KSM_NPAGES_DECAY=-50
# KSM_NPAGES_MIN - is the lower limit
for the `npages` value.
# KSM_NPAGES_MIN=64
# KSM_NPAGES_MAX - is the upper limit
for the `npages` value.
# KSM_NPAGES_MAX=1250
# KSM_THRES_COEF - is the RAM
percentage to be calculated in
parameter `thres`.
# KSM_THRES_COEF=20
# KSM_THRES_CONST - If this is a low
memory system, and the `thres` value
is less than `KSM_THRES_CONST`, then
reset `thres` value to
`KSM_THRES_CONST` value.
# KSM_THRES_CONST=2048
```

KSM is designed to improve performance and allow memory overcommits. It serves this purpose in most environments; however, KSM may introduce a performance overhead in some setups or environments – for example, if you have a few VMs that have similar memory content when you start them and loads of memory-intensive operations afterward. This will create issues as KSM will first work very hard to reduce the memory footprint, and then lose time to cover for all of the memory content differences between multiple VMs. Also, there is a concern that KSM may open a channel that could potentially be used to leak information across guests, as has been well documented in the past couple of years. If you have these concerns or if you see/experience KSM not helping to improve the performance of your workload, it can be disabled.

To disable KSM, stop the **`ksmtuned`** and **`ksm`** services in your system by executing the following:

```
# systemctl stop ksm
# systemctl stop ksmtuned
```

We have gone through the different tuning options for CPU and memory. The next big subject that we need to cover is NUMA configuration, where both CPU and memory configuration become a part of a larger story or context.

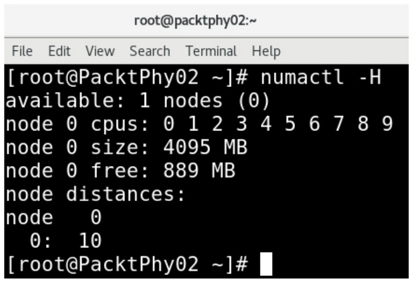
Tuning the CPU and memory with NUMA

Before we start tuning the CPU and memory for NUMA-capable systems, let's see what NUMA is and how it works.

Think of NUMA as a system where you have more than one system bus, each serving a small set of processors and associated memory. Each group of processors has its own memory and possibly its own I/O channels. It may not be possible to stop or prevent running VM access across these groups. Each of these groups is known as a **NUMA node**.

In this concept, if a process/thread is running on a NUMA node, the memory on the same node is called local memory and memory residing on a different node is known as foreign/remote memory. This implementation is different from the **Symmetric Multiprocessor System** (**SMP**), where the access time for all of the memory is the same for all the CPUs, as memory access happens through a centralized bus.

An important subject in discussing NUMA is the NUMA ratio. The NUMA ratio is a measure of how quickly a CPU can access local memory compared to how quickly it can access remote/foreign memory. For example, if the NUMA ratio is 2.0, then it takes twice as long for the CPU to access remote memory. If the NUMA ratio is 1, that means that we're using SMP. The bigger the ratio, the bigger the latency price (overhead) that a VM memory operation will have to pay before getting the necessary data (or saving it). Before we explore tuning in more depth, let's discuss exploring the NUMA topology of a system. One of the easiest ways to show the current NUMA topology is via the **`numactl`** command:



```
[root@PacktPhy02 ~]# numactl -H
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9
node 0 size: 4095 MB
node 0 free: 889 MB
node distances:
node   0
  0:  10
[root@PacktPhy02 ~]#
```

Figure 15.16 – The numactl -H output

The preceding **`numactl`** output conveys that there are 10 CPUs in the system and they belong to a single NUMA node. It also lists the memory associated with each NUMA node and the node distance. When we discussed CPU pinning, we displayed the topology of the system using the **`virsh`** capabilities. To get a graphical view of the NUMA topology, you can make use of a command called **`lstopo`**, which is available with the **`hwloc`** package in CentOS-/Red Hat-based systems:

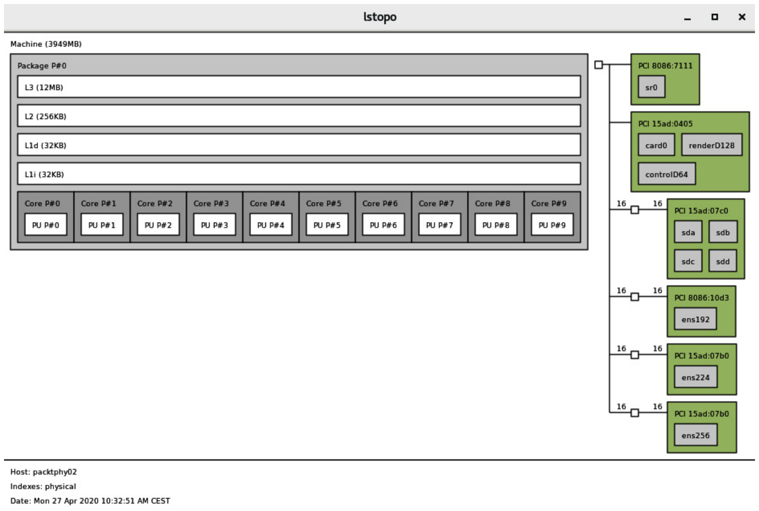


Figure 15.17 – The lstopo command to visualize the NUMA topology

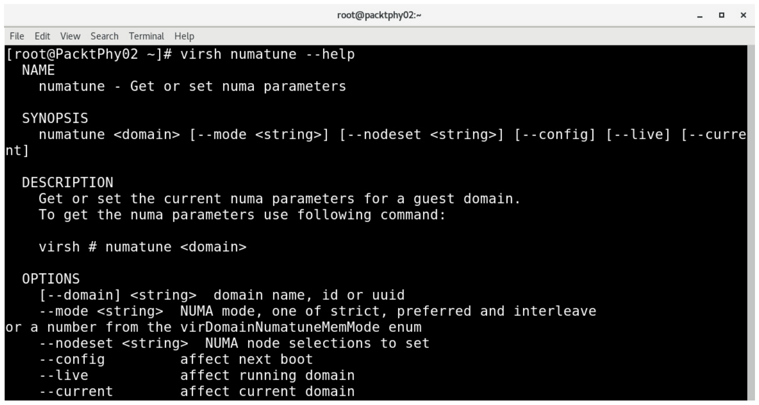
This screenshot also shows the PCI devices associated with the NUMA nodes. For example, **`ens*`** (network interface) devices are attached to NUMA node 0. Once we have the NUMA topology of the system and understand it, we can start tuning it, specially for the KVM virtualized setup.

NUMA memory allocation policies

By modifying the VM XML configuration file, we can do NUMA tuning. Tuning NUMA introduces a new element tag called **`numatune`**:

```
<domain>   ...
    <numatune>
      <memory mode="strict"
nodeset="1-4,^3"/>
    </numatune>   ...
</domain>
```

This is also configurable via the **`virsh`** command, as shown:



```
[root@PacktPhy02 ~]# virsh numatune --help
  NAME
    numatune - Get or set numa parameters

  SYNOPSIS
    numatune <domain> [--mode <string>] [--nodeset <string>] [--config] [--live] [--curre
nt]

  DESCRIPTION
    Get or set the current numa parameters for a guest domain.
    To get the numa parameters use following command:

    virsh # numatune <domain>

  OPTIONS
    [--domain] <string>  domain name, id or uuid
    --mode <string>  NUMA mode, one of strict, preferred and interleave
or a number from the virDomainNumatuneMemMode enum
    --nodeset <string>  NUMA node selections to set
    --config         affect next boot
    --live           affect running domain
    --current        affect current domain
```

Figure 15.18 – Using virsh numatune to configure the NUMA settings

The XML representation of this tag is as follows:

```
<domain>
…
  <numatune>
    <memory mode="strict" nodeset="1-
4,^3"/>
    <memnode cellid="0" mode="strict"
nodeset="1"/>
    <memnode cellid="2"
mode="preferred" nodeset="2"/>
  </numatune>   ...
</domain>
```

Even though the element called **`numatune`** is optional, it is provided to tune the performance of the NUMA host by controlling the NUMA policy for the domain process. The main sub-tags of this optional element are **`memory`** and **`nodeset`**. Some notes on these sub-tags are as follows:

- **`memory`**: This element describes the memory allocation process on the NUMA node. There are three policies that govern memory allocation for NUMA nodes:
  a) **`Strict`**: When a VM tries to allocate memory and that memory isn't available, allocation will

fail.
b) **`Interleave`**: Nodeset-defined round-robin allocation across NUMA nodes.
c) **`Preferred`**: The VM tries to allocate memory from a preferred node. If that node doesn't have enough memory, it can allocate memory from the remaining NUMA nodes.

- **`nodeset`**: Specifies a NUMA node list available on the server.

One of the important attributes here is *placement*, as explained at the following URL – https://libvirt.org/formatdomain.html:

"Attribute placement can be used to indicate the memory placement mode for domain process, its value can be either "static" or "auto", defaults to placement of vCPU, or "static" if nodeset is specified. "auto" indicates the domain process will only allocate memory from the advisory nodeset returned from querying numad, and the value of attribute nodeset will be ignored if it's specified. If placement of vCPU is 'auto', and numatune is not specified, a default numatune with placement 'auto' and mode 'strict' will be added implicitly."

We need to be careful with these declarations, as there are inheritance rules that apply. For example, the **`<numatune>`** and **`<vcpu>`** elements default to the same value if we specify the **`<nodeset>`** element. So, we can absolutely configure different CPU and memory tuning options, but also be aware of the fact that these options can be inherited.

There are some more things to consider when thinking about CPU pinning in the NUMA context. We discussed the basis of CPU pinning earlier in this chapter, as it gives us better, predictable performance for our VMs and can increase cache efficiency. Just as an example, let's say that we want to run a VM as fast as possible. It would be prudent to run it on the fastest storage available, which would be on a PCI Express bus on the CPU socket where we pinned the CPU cores. If we're not using an NVMe SSD local to that VM, we can use a storage controller to achieve the same thing. However, if the storage controller that we're using to access VM storage is physically connected to another CPU socket, that will lead to latency. For latency-sensitive applications, that will mean a big performance hit.

However, we also need to be aware of the other extreme – if we do too much pinning, it can create other problems in the future. For example, if our servers are not architecturally the same (having the same amount of cores and memory), migrating VMs might become problematic. We can create a scenario where we're migrating a VM with CPU cores pinned to cores that don't exist on the target server of our migration process. So, we always need to be careful about what we do with the configuration of our environments so that we don't take it too far.

The next subject on our list is **`emulatorpin`**, which can be used to pin our **`qemu-kvm`** emulator to a specific CPU core so that it doesn't influence the performance of our VM cores. Let's learn how to configure that.

Understanding emulatorpin

The **`emulatorpin`** option also falls into the CPU tuning category. The XML representation of this would be as follows:

```
<domain>   ...
    <cputune>     …..       <emulator
pin cpuset="1-3"/>      …..
    </cputune>   ...
</domain>
```

The **`emulatorpin`** element is optional and is used to pin the emulator (**`qemu-kvm`**) to a host physical CPU. This does not include the vCPU or IO threads from the VM. If this is omitted, the emulator is pinned to all the physical CPUs of the host system by default.

Important note:

Please note that **`<vcpupin>`**, **`<numatune>`**, *and* **`<emulatorpin>`** *should be configured together to achieve optimal, deterministic performance when you tune a NUMA-capable system.*

Before we leave this section, there are a couple more things to cover: the guest system NUMA topology and hugepage memory backing with NUMA.

Guest NUMA topology can be specified using the **`<numa>`** element in the guest XML configuration; some call this virtual NUMA:

```
    <cpu>      ...
    <numa>
      <cell id='0' cpus='0-3'
memory='512000' unit='KiB'/>
      <cell id='1' cpus='4-7'
memory='512000' unit='KiB'
/>     </numa>         ...
</cpu>
```

The **`cell id`** element tells the VM which NUMA node to use, while the **`cpus`** element configures a specific core (or cores). The **`memory`** element assigns the amount of memory per node. Each NUMA node is indexed by number, starting from **`0`**.

Previously, we discussed the **`memorybacking`** element, which can be specified to use hugepages in guest configurations. When NUMA is present in a setup, the **`nodeset`** attribute can be used to configure the specific hugepage size per NUMA node, which may come in handy as it ties a given guest's NUMA nodes to certain hugepage sizes:

```
<memoryBacking>
    <hugepages>
      <page size="1" unit="G"
nodeset="0-2,4"/>
      <page size="4" unit="M"
nodeset="3"/>
    </hugepages>
</memoryBacking>
```

This type of configuration can optimize the memory performance, as guest NUMA nodes can be moved to host NUMA nodes as required, while the guest can continue to use the hugepages allocated by the host.

NUMA tuning also has to consider the NUMA node locality for PCI devices, especially when a PCI device is being passed through to the guest from the host. If the relevant PCI device is affiliated to a remote NUMA node, this can affect data transfer and thus hurt the performance.

The easiest way to display the NUMA topology and PCI device affiliation is by using the **lstopo** command that we discussed earlier. The non-graphic form of the same command can also be used to discover this configuration. Please refer to the earlier sections.

KSM and NUMA

We discussed KSM in enough detail in previous sections. KSM is NUMA-aware, and it can manage KSM processes happening on multiple NUMA nodes. If you remember, we encountered a **sysfs** entry called **merge_across_node** when we fetched KSM entries from **sysfs**. That's the parameter that we can use to manage this process:

```
#  cat
/sys/kernel/mm/ksm/merge_across_nodes
1
```

If this parameter is set to **0**, KSM only merges memory pages from the same NUMA node. If it's set to **1** (as is the case here), it will merge *across* the NUMA nodes. That means that the VM CPUs that are running on the remote NUMA node will experience latency when accessing a KSM-merged page.

Obviously, you know the guest XML entry (the **`memorybacking`** element) for asking the hypervisor to disable shared pages for the guest. If you don't remember, please refer back to the memory tuning section for details of this element. Even though we can configure NUMA manually, there is something called automatic NUMA balancing. We did mention it earlier, but let's see what this concept involves.

Automatic NUMA balancing

The main aim of automatic NUMA balancing is to improve the performance of different applications running in a NUMA-aware system. The strategy behind its design is simple: if an application is using local memory to the NUMA node where vCPUs are running, it will have better performance. By using automatic NUMA balancing, KVM tries to shift vCPUs around so that they are local (as much as possible) to the memory addresses that the vCPUs are using. This is all done automatically by the kernel when automatic NUMA balancing is active. Automatic NUMA balancing will be enabled when booted on the hardware with NUMA properties. The main conditions or criteria are as follows:

- **`numactl --hardware`**: Shows multiple nodes
- **`cat /sys/kernel/debug/sched_features`**: Shows NUMA in the flags

To illustrate the second point, see the following code block:

```
#  cat
/sys/kernel/debug/sched_features
GENTLE_FAIR_SLEEPERS START_DEBIT
NO_NEXT_BUDDY LAST_BUDDY
CACHE_HOT_BUDDY
WAKEUP_PREEMPTION ARCH_POWER
NO_HRTICK NO_DOUBLE_TICK LB_BIAS
NONTASK_
POWER TTWU_QUEUE NO_FORCE_SD_OVERLAP
RT_RUNTIME_SHARE NO_LB_MIN NUMA
NUMA_FAVOUR_HIGHER
NO_NUMA_RESIST_LOWER
```

We can check whether it is enabled in the system via the following method:

```
#  cat
/proc/sys/kernel/numa_balancing
1
```

Obviously, we can disable automatic NUMA balancing via the following:

```
# echo 0 >
/proc/sys/kernel/numa_balancing
```

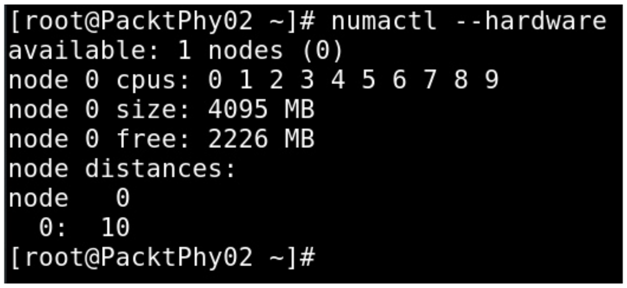
The automatic NUMA balancing mechanism works based on the number of algorithms and data structures. The internals of this method are based on the following:

- NUMA hinting page faults
- NUMA page migration
- Pseudo-interleaving
- Fault statistics
- Task placement
- Task grouping

One of the best practices or recommendations for a KVM guest is to limit its resource to the amount of resources on a single NUMA node. Put simply, this avoids the unnecessary splitting of VMs across NUMA nodes, which can degrade the performance. Let's start by checking the current NUMA configuration. There are multiple available options to do this. Let's start with the **numactl** command, NUMA daemon, and **numastat**, and then go back to using a well-known command, **virsh**.

The numactl command

The first option to confirm NUMA availability uses the **numactl** command, as shown:



```
[root@PacktPhy02 ~]# numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9
node 0 size: 4095 MB
node 0 free: 2226 MB
node distances:
node   0
  0:  10
[root@PacktPhy02 ~]#
```

Figure 15.19 – The numactl hardware output

This lists only one node. Even though this conveys the unavailability of NUMA, further clarification can be done by running the following command:

```
# cat
/sys/kernel/debug/sched_features
```

This will *not* list NUMA flags if the system is not NUMA-aware.

Generally, don't make VMs *wider* than what a single NUMA node can provide. Even if the NUMA is available, the vCPUs are bound to the NUMA node and not to a particular physical CPU.

Understanding numad and numastat

The **numad** man page states the following:

numad is a daemon to control efficient use of CPU and memory on systems with NUMA topology.

numad is also known as the automatic **NUMA Affinity Management Daemon**. It constantly monitors NUMA resources on a system in order to dynamically improve NUMA performance. Again, the **numad** man page states the following:

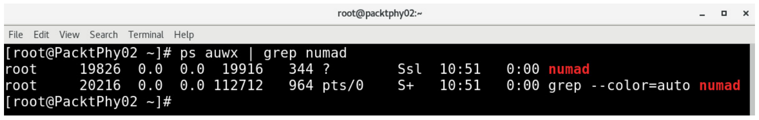
"numad is a user-level daemon that provides placement advice and process management for efficient use of CPUs and memory on systems with NUMA topology."

numad is a system daemon that monitors the NUMA topology and resource usage. It will attempt to locate processes for efficient NUMA locality and affinity, dynamically adjusting to changing system conditions. **numad** also provides guidance to assist management applications with the initial manual binding of CPU and memory resources for their processes. Note that **numad** is primarily intended for server consolidation environments, where there might be multiple applications or multiple virtual guests running on the same server system. **numad** is most likely to have a positive effect when processes can be localized in a subset of the system's NUMA nodes. If the entire system is dedicated to a large in-memory database application, for example, especially if memory accesses will likely remain unpredictable, **numad** will probably not improve performance.

To adjust and align the CPUs and memory resources automatically according to the NUMA topology, we need to run **numad**. To use **numad** as an executable, just run the following:

```
# numad
```

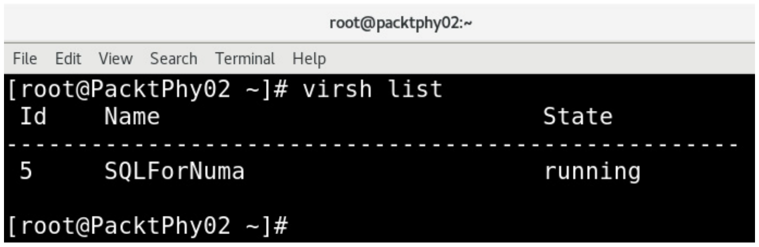
You can check whether this is started as shown:



```
[root@PacktPhy02 ~]# ps auwx | grep numad
root     19826  0.0  0.0  19916   344 ?        Ssl  10:51   0:00 numad
root     20216  0.0  0.0 112712   964 pts/0    S+   10:51   0:00 grep --color=auto numad
[root@PacktPhy02 ~]#
```

Figure 15.20 – Checking whether numad is active

Once the **numad** binary is executed, it will start the alignment, as shown in the following screenshot. In our system, we have the following VM running:



```
[root@PacktPhy02 ~]# virsh list
 Id    Name                           State
----------------------------------------------------
 5     SQLForNuma                     running

[root@PacktPhy02 ~]#
```

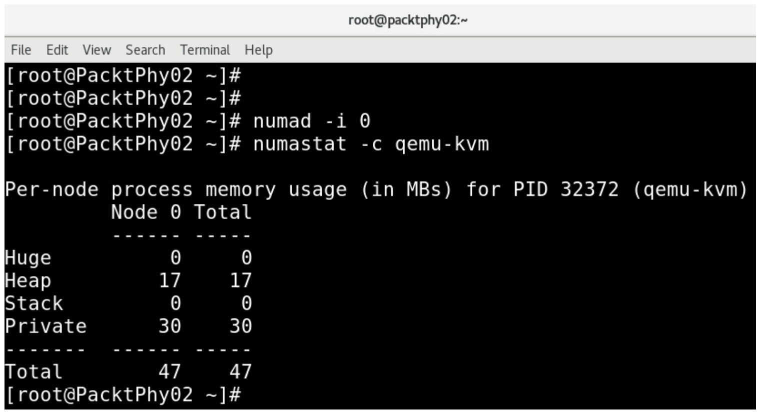
Figure 15.21 – Listing running VMs

You can use the **numastat** command, covered in an upcoming section, to monitor the difference before and after running the **numad** service. It will run continuously by using the following command:

```
# numad -i 0
```

We can always stop it, but that will not change the NUMA affinity state that was configured by **numad**. Now let's move on to **numastat**.

The **`numactl`** package provides the **`numactl`** binary/command and the **`numad`** package provides the **`numad`** binary/command:



```
[root@PacktPhy02 ~]#
[root@PacktPhy02 ~]#
[root@PacktPhy02 ~]# numad -i 0
[root@PacktPhy02 ~]# numastat -c qemu-kvm

Per-node process memory usage (in MBs) for PID 32372 (qemu-kvm)
         Node 0 Total
         ------ -----
Huge          0     0
Heap         17    17
Stack         0     0
Private      30    30
-------  ------ -----
Total        47    47
[root@PacktPhy02 ~]#
```

Figure 15.22 – The numastat command output for the qemu-kvm process

Important note:

The numerous memory tuning options that we have used have to be thoroughly tested using different workloads before moving the VM to production.

Before we jump on to the next topic, we'd just like to remind you of a point we made earlier in this chapter. Live-migrating a VM with pinned resources might be complicated, as you have to have some form of compatible resources (and their amount) on the target host. For example, the target host's NUMA topology doesn't have to be aligned with the source host's NUMA topology. You should consider this fact when you tune a KVM environment. Automatic NUMA balancing may help, to a certain extent, the need for manually pinning guest resources, though.

Virtio device tuning

In the virtualization world, a comparison is always made with bare-metal systems. Paravirtualized drivers enhance the performance of guests and try to retain near-bare-metal performance. It is recommended to use paravirtualized drivers for fully virtualized guests, especially when the guest is running with I/O-heavy tasks and applications. **Virtio** is an API for virtual IO and was developed by Rusty Russell in support of his own virtualization solution, called `lguest`. Virtio was introduced to achieve a common framework for hypervisors for IO virtualization.

In short, when we use paravirtualized drivers, the VM OS knows that there's a hypervisor beneath it, and therefore uses frontend drivers to access it. The frontend drivers are part of the guest system. When there are emulated devices and someone wants to implement backend drivers for these devices, hypervisors do this job. The frontend and backend drivers communicate through a virtio-based path. Virtio drivers are what KVM uses as paravirtualized device drivers. The basic architecture looks like this:

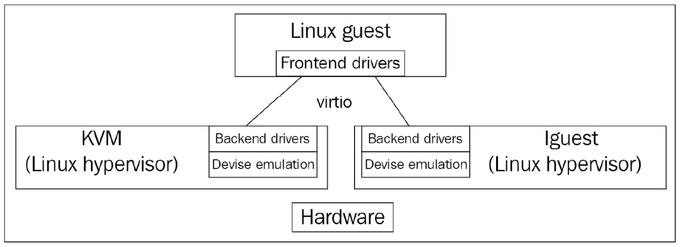


Figure 15.23 – The Virtio architecture

There are mainly two layers (virt queue and virtual ring) to support communication between the guest and the hypervisor.

Virt queue and **virtual ring** (**vring**) are the transport mechanism implementations in virtio. Virt queue (virtio) is the queue interface that attaches the frontend and backend drivers. Each virtio device has its own virt queues and requests from guest systems are put into these virt queues. Each virt queue has its own ring, called a vring, which is where the memory is mapped between QEMU and the guest. There are different virtio drivers available for use in a KVM guest.

The devices are emulated in QEMU, and the drivers are part of the Linux kernel, or an extra package for Windows guests. Some examples of device/driver pairs are as follows:

- **`virtio-net`**: The virtio network device is a virtual Ethernet card. **`virtio-net`** provides the driver for this.
- **`virtio-blk`**: The virtio block device is a simple virtual block device (that is, a disk). **`virtio-blk`** provides the block device driver for the virtual block device.
- **`virtio-balloon`**: The virtio memory balloon device is a device for managing guest memory.
- **`virtio-scsi`**: The virtio SCSI host device groups together one or more disks and allows communicating to them using the SCSI protocol.
- **`virtio-console`**: The virtio console device is a simple device for data input and output between the guest and host userspace.
- **`virtio-rng`**: The virtio entropy device supplies high-quality randomness for guest use, and so on.

In general, you should make use of these virtio devices in your KVM setup for better performance.

Block I/O tuning

Going back to basics – a virtual disk of a VM can be either a block device or an image file. For better VM performance, a block device-based virtual disk is preferred over an image file that resides on a remote filesystem such as NFS, GlusterFS, and so on. However, we cannot ignore that the file backend helps the virt admin to better manage guest disks and it is immensely helpful in some scenarios. From our experience, we have noticed most users make use of disk image files, especially when performance is not much of a concern. Keep in mind that the total number of virtual disks that can be attached to a VM has a limit. At the same time, there is no restriction on mixing and using block devices and files and using them as storage disks for the same guest.

A guest treats the virtual disk as its storage. When an application inside a guest OS writes data to the local storage of the guest system, it has to pass through a couple of layers. That said, this I/O request has to traverse through the filesystem on the storage and the I/O subsystem of the guest OS. After that, the **`qemu-kvm`** process passes it to the hypervisor from the guest OS. Once the I/O is within the realm of the hypervisor, it starts processing the I/O like any other applications running in the host OS. Here, you can see the number of layers that the I/O has to pass through to complete an I/O operation. Hence, the block device backend performs better than the image file backend.

The following are our observations on disk backends and file- or image-based virtual disks:

- A file image is part of the host filesystem and it creates an additional resource demand for I/O operations compared to the block device backend.
- Using sparse image files helps to over allocate host storage but its usage will reduce the performance of the virtual disk.
- The improper partitioning of guest storage when using disk image files can cause unnecessary I/O operations. Here, we are mentioning the alignment of standard partition units.

At the start of this chapter, we discussed virtio drivers, which give better performance. So, it's recommended that you use the virtio disk bus when configuring the disk, rather than the IDE bus. The **`virtio_blk`** driver uses the virtio API to provide high performance for storage I/O device, thus increasing storage performance, especially in large enterprise storage systems. We discussed the different storage formats available in ***Chapter 5***, *Libvirt Storage*; however, the main ones are the **`raw`** and **`qcow`** formats. The best performance will be achieved when you are using the **`raw`** format. There is obviously a performance overhead delivered by the format layer when using **`qcow`**. Because the format layer has to perform some operations at times, for example, if you want to grow a **`qcow`** image, it has to allocate the new cluster and so on. However, **qcow** would be an option if you want to make use of features such as snapshots. These extra facilities are provided with the image format, **qcow**. Some performance comparisons can be found at http://www.Linux-kvm.org/page/Qcow2.

There are three options that can be considered for I/O tuning, which we discussed in *Chapter 7*, *Virtual Machine – Installation, Configuration, and Life Cycle Management*:

- Cache mode
- I/O mode
- I/O tuning

Let's briefly go through some XML settings so that we can implement them on our VMs.

The cache option settings can reflect in the guest XML, as follows:

```
<disk type='file' device='disk'>
<driver name='qemu' type='raw'
cache='writeback'/>
```

The XML representation of I/O mode configuration is similar to the following:

```
<disk type='file' device='disk'>
<driver name='qemu' type='raw'
io='threads'/>
```

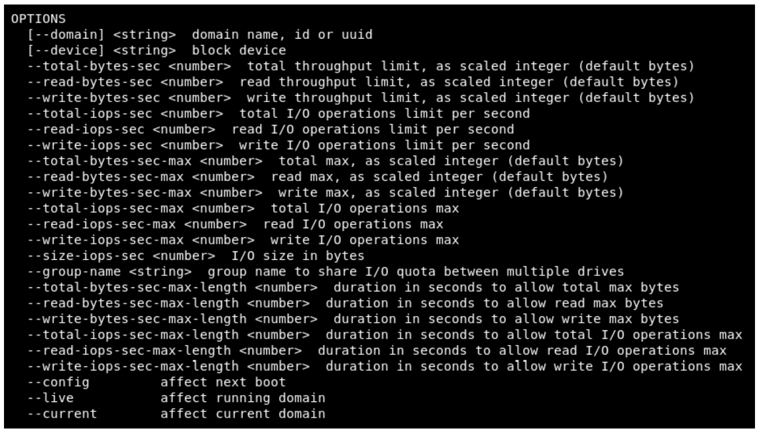
In terms of I/O tuning, a couple of additional remarks:

- Limiting the disk I/O of each guest may be required, especially when multiple guests exist in our setup.

- If one guest is keeping the host system busy with the number of disk I/Os generated from it (noisy neighbor problem), that's not fair to the other guests.

Generally speaking, it is the system/virt administrator's responsibility to ensure all the running guests get enough resources to work on —in other words, the **Quality of Service** (**QOS**).

Even though the disk I/O is not the only resource that has to be considered to guarantee QoS, this has some importance. Tuning I/O can prevent a guest system from monopolizing shared resources and lowering the performance of other guests running on the same host. This is really a requirement, especially when the host system is serving a **Virtual Private Server** (**VPS**) or a similar kind of service. KVM gives the flexibility to do I/O throttling on various levels – throughput and I/O amount, and we can do it per block device. This can be achieved via the **`virsh blkdeviotune`** command. The different options that can be set using this command are displayed as follows:



```
OPTIONS
  [--domain] <string>  domain name, id or uuid
  [--device] <string>  block device
  --total-bytes-sec <number>  total throughput limit, as scaled integer (default bytes)
  --read-bytes-sec <number>  read throughput limit, as scaled integer (default bytes)
  --write-bytes-sec <number>  write throughput limit, as scaled integer (default bytes)
  --total-iops-sec <number>  total I/O operations limit per second
  --read-iops-sec <number>  read I/O operations limit per second
  --write-iops-sec <number>  write I/O operations limit per second
  --total-bytes-sec-max <number>  total max, as scaled integer (default bytes)
  --read-bytes-sec-max <number>  read max, as scaled integer (default bytes)
  --write-bytes-sec-max <number>  write max, as scaled integer (default bytes)
  --total-iops-sec-max <number>  total I/O operations max
  --read-iops-sec-max <number>  read I/O operations max
  --write-iops-sec-max <number>  write I/O operations max
  --size-iops-sec <number>  I/O size in bytes
  --group-name <string>  group name to share I/O quota between multiple drives
  --total-bytes-sec-max-length <number>  duration in seconds to allow total max bytes
  --read-bytes-sec-max-length <number>  duration in seconds to allow read max bytes
  --write-bytes-sec-max-length <number>  duration in seconds to allow write max bytes
  --total-iops-sec-max-length <number>  duration in seconds to allow total I/O operations max
  --read-iops-sec-max-length <number>  duration in seconds to allow read I/O operations max
  --write-iops-sec-max-length <number>  duration in seconds to allow write I/O operations max
  --config         affect next boot
  --live           affect running domain
  --current        affect current domain
```

Figure 15.24 – Excerpt from the virsh blkdeviotune –help command

Details about parameters such as **total-bytes-sec**, **read-bytes-sec**, **writebytes-sec**, **total-iops-sec**, and so on are easy to understand from the preceding command output. They are also documented in the **virsh** command man page.

For example, to throttle the **vdb** disk on a VM called **SQLForNuma** to 200 I/O operations per second and 50 MB-per-second throughput, run this command:

```
# virsh blkdeviotune SQLForNuma vdb -
-total-iops-sec 200 --total-bytes-sec
52428800
```

Next, we are going to look at network I/O tuning.

Network I/O tuning

What we've seen in most KVM environments is that all the network traffic from a guest will take a single network path. There won't be any traffic segregation, which causes congestion in most KVM setups. As a first step for network tuning, we'd advise trying different networks or dedicated networks for management, backups, or live migration. But when you have more than one network interface for your traffic, please try to avoid multiple network interfaces for the same network or segment. If this is at all in play, apply some network tuning that is common for such setups; for example, use **arp_filter** to control ARP Flux. ARP Flux happens when a VM has more than one network interface and is using them actively to reply to ARP requests, so we should do the following:

```
echo 1 >
/proc/sys/net/ipv4/conf/all/arp_filte
r
```

After that, you need to edit **/etc/sysctl.conf** to make this setting persistent.

For more information on ARP Flux, please refer to http://linux-ip.net/html/ether-arp.html#ether-arp-flux.

Additional tuning can be done on the driver level; that said, by now we know that virtio drivers give better performance compared to emulated device APIs. So, obviously, using the **virtio_net** driver in guest systems should be taken into account. When we use the **virtio_net** driver, it has a backend driver in **qemu** that takes care of the communication initiated from the guest network. Even if this was performing better, some more enhancements in this area introduced a new driver called **vhost_net**, which provides in-kernel virtio devices for KVM. Even though vhost is a common framework that can be used by different drivers, the network driver, **vhost_net**, was one of the first drivers. The following diagram will make this clearer:

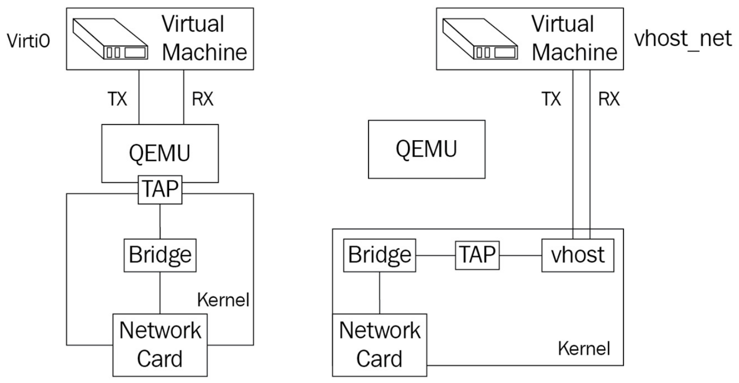


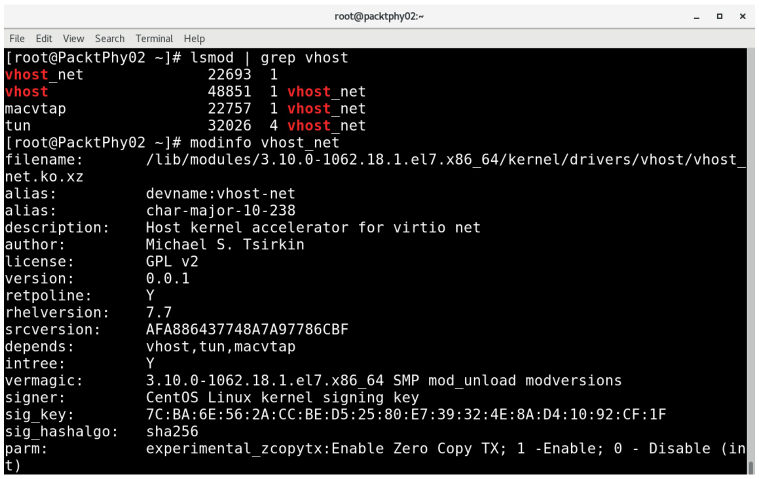
Figure 15.25 – The vhost_net architecture

As you may have noticed, the number of context switches is really reduced with the new path of communication. The good news is that there is no extra configuration required in guest systems to support vhost because there is no change to the frontend driver.

`vhost_net` reduces copy operations, lowers latency and CPU usage, and thus yields better performance. First of all, the kernel module called **`vhost_net`** (refer to the screenshot in the next section) has to be loaded in the system. As this is a character device inside the host system, it creates a device file called **`/dev/vhost-net`** on the host.

How to turn it on

When QEMU is launched with **`-netdev tap,vhost=on`**, it will instantiate the **`vhost-net`** interface by using **`ioctl()`** calls. This initialization process binds **`qemu`** with a **`vhost-net`** instance, along with other operations such as feature negotiations and so on:



```
[root@PacktPhy02 ~]# lsmod | grep vhost
vhost_net              22693  1
vhost                  48851  1 vhost_net
macvtap                22757  1 vhost_net
tun                    32026  4 vhost_net
[root@PacktPhy02 ~]# modinfo vhost_net
filename:       /lib/modules/3.10.0-1062.18.1.el7.x86_64/kernel/drivers/vhost/vhost_
net.ko.xz
alias:          devname:vhost-net
alias:          char-major-10-238
description:    Host kernel accelerator for virtio net
author:         Michael S. Tsirkin
license:        GPL v2
version:        0.0.1
retpoline:      Y
rhelversion:    7.7
srcversion:     AFA886437748A7A97786CBF
depends:        vhost,tun,macvtap
intree:         Y
vermagic:       3.10.0-1062.18.1.el7.x86_64 SMP mod_unload modversions
signer:         CentOS Linux kernel signing key
sig_key:        7C:BA:6E:56:2A:CC:BE:D5:25:80:E7:39:32:4E:8A:D4:10:92:CF:1F
sig_hashalgo:   sha256
parm:           experimental_zcopytx:Enable Zero Copy TX; 1 -Enable; 0 - Disable (in
t)
```

Figure 15.26 – Checking vhost kernel modules

One of the parameters available with the **`vhost_net`** module is **`experimental_ zcopytx`**. What does it do? This parameter controls something called bridge zero copy transmit. Let's see what this means (as stated on http://www.google.com/patents/US20110126195):

"A system for providing a zero copy transmission in virtualization environment includes a hypervisor that receives a guest operating system (OS) request pertaining to a data packet associated with a guest application, where the data packet resides in a buffer of the guest OS or a buffer of the guest application and has at least a partial header created during the networking stack processing. The hypervisor further sends, to a network device driver, a request to transfer the data packet over a network via a network device, where the request identifies the data packet residing in the buffer of the guest OS or the buffer of the guest application, and the hypervisor refrains from copying the data packet to a hypervisor buffer."

If your environment uses large packet sizes, configuring this parameter may have a noticeable effect. The host CPU overhead is reduced by configuring this parameter when the guest communicates to the external network. This does not affect the performance in the following scenarios:

- Guest-to-guest communication
- Guest-to-host communication
- Small packet workloads

Also, the performance improvement can be obtained by enabling multi queue **`virtio-net`**. For additional information, check out https://fedoraproject.org/wiki/Features/MQ_virtio_net.

One of the bottlenecks when using **virtio-net** was its single RX and TX queue. Even though there are more vCPUs, the networking throughput was affected by this limitation. **virtio-net** is a single-queue type of queue, so multi-queue **virtio-net** was developed. Before this option was introduced, virtual NICs could not utilize the multi-queue support that is available in the Linux kernel.

This bottleneck is lifted by introducing multi-queue support in both frontend and backend drivers. This also helps guests scale with more vCPUs. To start a guest with two queues, you could specify the **queues** parameters to both **tap** and **virtio-net**, as follows:

```
# qemu-kvm -netdev tap,queues=2,... -
device virtio-net-pci,queues=2,...
```

The equivalent guest XML is as follows:

```
<interface type='network'>
    <source network='default'/>
    <model type='virtio'/>
    <driver name='vhost' queues='M'/>
</interface>
```

Here, **M** can be **1** to **8**, as the kernel supports up to eight queues for a multi-queue tap device. Once it's configured for **qemu**, inside the guest, we need to enable multi-queue support with the **ethtool** command. Enable the multi-queue through **ethtool** (where the value of **K** is from **1** to **M**), as follows:

```
# ethtool -L eth0 combined 'K'
```

You can check the following link to see when multi-queue **virtio-net** provides the greatest performance benefit: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimiza virtualization_tuning_optimization_guide-net-working-techniques.

Don't use the options mentioned on the aforementioned URL blindly – please test the impact on your setup, because the CPU consumption will be greater in this scenario even though the network throughput is impressive.

KVM guest time-keeping best practices

There are different mechanisms for time-keeping. One of the best-known techniques is **Network Time Protocol** (**NTP**). By using NTP, we can synchronize clocks to great accuracy, even when using networks that have jitter (variable latency). One thing that needs to be considered in a virtualization environment is the maxim that the guest time should be in sync with the hypervisor/host, because it affects a lot of guest operations and can cause unpredictable results if they are not in sync.

There are different ways to achieve time sync, however; it depends on the setup you have. We have seen people using NTP, setting the system clock from the hardware clock using **hwclock -s**, and so on. The first thing that needs to be considered here is trying to make the KVM host time in sync and stable. You can use NTP-like protocols to achieve this. Once it's in place, the guest time has to be kept in sync. Even though there are different mechanisms for doing that, the best option would be using **`kvm-clock`**.

kvm-clock

`kvm-clock` is also known as a virtualization-aware (paravirtualized) clock device. When **`kvm-clock`** is in use, the guest asks the hypervisor about the current time, guaranteeing both stable and accurate timekeeping. The functionality is achieved by the guest registering a page and sharing the address with the hypervisor. This is a shared page between the guest and the hypervisor. The hypervisor keeps updating this page unless it is asked to stop. The guest can simply read this page whenever it wants time information. However, please note that the hypervisor should support **`kvm-clock`** for the guest to use it. For more details, you can check out https://lkml.org/lkml/2010/4/15/355.

By default, most of the newer Linux distributions use **Time Stamp Counter** (**TSC**), a CPU register, as a clock source. You can verify whether TSC or **`kvm_clock`** are configured inside the guest via the following method:

```
[root@kvmguest ]$  cat
/sys/devices/system/clocksource/clock
source0/current_clocksource
tsc
```

You can also use **`ntpd`** or **`chrony`** as your clock sources on Linux, which requires minimal configuration. In your Linux VM, edit **/etc/ntpd.conf** or **/etc/chronyd.conf** and modify the *server* configuration lines to point to your NTP servers by IP address. Then, just enable and start the service that you're using (we're using **chrony** as an example here):

```
systemctl enable chronyd
systemctl start chronyd
```

There's another, a bit newer, protocol that's being heavily pushed for time synchronization, which is called the **Precision Time Protocol (PTP)**. Nowadays, this is becoming the de facto standard service to be used on the host level. This protocol is directly supported in hardware (as in network interface cards) for many of the current network cards available on the market. As it's basically hardware-based, it should be even more accurate then **ntpd** or **chronyd**. It uses timestamping on the network interface, and external sources, and the computer's system clock for synchronization.

Installing all of the necessary pre-requisites is just a matter of one **yum** command to enable and start a service:

```
yum -y install linuxptp
systemctl enable ptp4l
systemctl start ptp4l
```

By default, the **ptp4l** service will use the **/etc/sysconfig/ptp4l** configuration file, which is usually bound to the first network interface. If you want to use some other network interface, the simplest thing to do would be to edit the configuration file, change the interface name, and restart the service via **systemctl**.

Now, from the perspective of VMs, we can help them time sync by doing a little bit of configuration. We can add the **ptp_kvm** module to the global KVM host configuration, which is going to make our PTP as a service available to **chronyd** as a clock source. This way, we don't have to do a lot of additional configuration. So, just add **ptp_kvm** as a string to the default KVM configuration, as follows:

```
echo ptp_kvm > /etc/modules-
load.d/kvm-chrony.conf
modprobe ptp_kvm
```

By doing this, a **ptp** device will be created in the **/dev** directory, which we can then use as a **chrony** time source. Add the following line to **/etc/chrony.conf** and restart **chronyd**:

```
refclock PHC /dev/ptp0 poll 3 dpoll
-2 offset 0
systemctl restart chronyd
```

By using an API call, all Linux VMs are capable of then getting their time from the physical host running them.

Now that we've covered a whole bunch of VM configuration options in terms of performance tuning and optimization, it's time to finally step away from all of these micro-steps and focus on the bigger picture. Everything that we've covered so far in terms of VM design (related to the CPU, memory, NUMA, virtio, block, network, and time configuration) is only as important as what we're using it for. Going back to our original scenario – a SQL VM – let's see how we're going to configure our VM properly in terms of the software that we're going to run on it.

Software-based design

Remember our initial scenario, involving a Windows Server 2019-based VM that should be a node in a Microsoft SQL Server cluster? We covered a lot of the settings in terms of tuning, but there's more to do – much more. We need to be asking some questions. The sooner we ask these questions, the better, as they're going to have a key influence on our design.

Some questions we may ask are as follows:

- Excuse me, dear customer, when you say *cluster*, what do you mean specifically, as there are different SQL Server clustering methodologies?
- Which SQL licenses do you have or are you planning to buy?
- Do you need active-active, active-passive, a backup solution, or something else?
- Is this a single-site or a multi-site cluster?
- Which SQL features do you need exactly?
- Which licenses do you have and how much are you willing to spend on them?
- Is your application capable of working with a SQL cluster (for example, in a multi-site scenario)?
- What kind of storage system do you have?
- What amount of IOPS can your storage system provide?

- How are latencies on your storage?
- Do you have a storage subsystem with different tiers?
- What are the service levels of these tiers in terms of IOPS and latency?
- If you have multiple storage tiers, can we create SQL VMs in accordance with the best practices – for example, place data files and log files on separate virtual disks?
- Do you have enough disk capacity to meet your requirements?

These are just licensing, clustering, and storage-related questions, and they are not going to go away. They need to be asked, without hesitation, and we need to get real answers before deploying things. We have just mentioned 14 questions, but there are actually many more.

Furthermore, we need to think about other aspects of VM design. It would be prudent to ask some questions such as the following:

- How much memory can you give for SQL VMs?
- Which servers do you have, which processors are they using, and how much memory do you have per socket?
- Are you using any latest-gen technologies, such as persistent memory?
- Do you have any information about the scale and/or amount of queries that you're designing this SQL infrastructure for?
- Is money a big deciding factor in this project (as it will influence a number of design decisions as SQL is licensed per core)? There's also

the question of Standard versus Enterprise pricing.

This stack of questions actually points to one very, very important part of VM design, which is related to memory, memory locality, the relationship between CPU and memory, and also one of the most fundamental questions of database design – latency. A big part of that is related to correct VM storage design – the correct storage controller, storage system, cache settings, and so on, and VM compute design – which is all about NUMA. We've explained all of those settings in this chapter. So, to configure our SQL VM properly, here's a list of the high-level steps that we should follow:

- Configure a VM with the correct NUMA settings and local memory. Start with four vCPUs for licensing reasons and then figure out whether you need more (such as if your VM becomes CPU-limited, which you will see from performance graphs and SQL-based performance monitoring tools).
- If you want to reserve CPU capacity, make use of CPU pinning so that specific CPU cores on the physical server's CPU is always used for the SQL VM, and only that. Isolate other VMs to the *remaining* cores.
- Reserve memory for the SQL VM so that it doesn't swap, as only using real RAM memory will guarantee smooth performance that's not influenced by noisy neighbors.
- Configure KSM per VM if necessary and avoid using it on SQL VMs as it might introduce la-

tency. In the design phase, make sure you buy as much RAM memory as possible so that memory doesn't become an issue as it will be a very costly issue in terms of performance if a server doesn't have enough of it. Don't *ever* overcommit memory.

- Configure the VM with multiple virtual hard disks and put those hard disks in storage that can provide levels of service needed in terms of latency, overhead, and caching. Remember, an OS disk doesn't necessarily need write caching, but database and log disks will benefit from it.
- Use separate physical connections from your hosts to your storage devices and tune storage to get as much performance out of it as possible. Don't oversubscribe – both on the links level (too many VMs going through the same infrastructure to the *same* storage device) and the datastore level (don't put one datastore on a storage device and store all VMs on it as it will negatively impact performance – isolate workloads, create multiple targets via multiple links, and use masking and zoning).
- Configure multipathing, load balancing, and failover – to get as much performance out of your storage, yes, but also to have redundancy.
- Install the correct virtio drivers, use vhost drivers or SR-IOV if necessary, and minimize the overhead on every level.
- Tune the VM guest OS – turn off unnecessary services, switch the power profile to `High Performance` (most Microsoft OSes have a default setting that puts the power profile into

`Balanced` mode for some reason). Tune the BIOS settings and check the firmware and OS updates – everything – from top to bottom. Take notes, measure, benchmark, and use previous benchmarks as baselines when updating and changing the configuration so that you know which way you're going.

- When using iSCSI, configure jumbo frames as in most use cases, this will have a positive influence on the storage performance, and make sure that you check the storage device vendor's documentation for any best practices in that regard.

The takeway of this chapter is the following – don't just blindly install an application just because a client asks you to install it. It will come to haunt you later on, and it will be much, much more difficult to resolve any kind of problems and complaints. Take your time and do it right. Prepare for the whole process by reading the documentation, as it's widely available.

Summary

In this chapter, we did some digging, going deep into the land of KVM performance tuning and optimization. We discussed many different techniques, varying from simple ones, such as CPU pinning, to much more complex ones, such as NUMA and proper NUMA configuration. Don't be put off by this, as learning design is a process, and designing things correctly is a craft that can always be improved with learning and experience. Think of it this way – when architects were designing the highest skyscrapers in the world, didn't they move the goalposts farther and farther with each new highest building?

In the next chapter – the final chapter of this book - we will discuss troubleshooting your environments. It's at least partially related to this chapter, as we will be troubleshooting some issues related to performance as well. Go through this chapter multiple times before switching to the troubleshooting chapter – it will be very, very beneficial for your overall learning process.

Questions

1. What is CPU pinning?
2. What does KSM do?
3. How do we enhance the performance of block devices?
4. How do we tune the performance of network devices?
5. How can we synchronize clocks in virtualized environments?
6. How do we configure NUMA?
7. How do we configure NUMA and KSM to work together?

Further reading

Please refer to the following links for more information:

- RedHat Enterprise Linux 7 – installing, configuring, and managing VMs on a RHEL physical machine:

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_a

- vCPU pinning: http://libvirt.org/formatdomain.html#elementsCPUTuning
- KSM kernel documentation: https://www.kernel.org/doc/Documentation/vm/ksm.txt
- Placement: http://libvirt.org/formatdomain.html#elementsNUMATuning
- Automatic NUMA balancing: https://www.redhat.com/files/summit/2014/summit2014_riel_chegu_w_
- Virtio 1.1 specification: http://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html
- ARP Flux: http://Linux-ip.net/html/ether-arp.html#ether-arp-flux
- MQ virtio: https://fedoraproject.org/wiki/Features/MQ_virtio_net
- libvirt NUMA tuning on RHEL 7: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimi virtualization_tuning_optimization_guide-numa-numa_and_libvirt