Leduccc Follow
Jul 10, 2020 · 9 min read · Listen

Improving the performance of a Windows Guest on KVM/QEMU

This guide is intended for existing KVM/QEMU Libvirt setups and will help you reach near native performance on your Windows 10 or 11 Guest. This guide is a collection of the all interventions I could find on wikis, forums and blogs that had a measurable impact on guest performance benchmarks.

Table of content

Last Update: 2021-12-05

1 —Preamble & Prerequisites

Without tuning your VM, you may experience stuttering, high CPU usage and slow interrupts and I/O but it may still be usable. That said, if you want to get closer to bare

1.1 — Prerequisites

Mandatory
- Working Windows 10 or Windows 11 Guest
- CPU supporting VT-D or AMD-Vi
- Kernel: 5.8 +
- QEMU 6+
- Libvirt 7+
- KVM

Optional
- Working GPU Passthrough
- Kernel 5.15+ (Best performances)
- Kernel 5+ Compiled with voluntary Preemption

1.2 — Reference Setup

The reference setup is describe in details with instruction on how to reproduce it in this article.

Hardware
CPU: i9-9880H
GPU: Quadro T1000 Mobile
IGPU: Intel UHD 630

Software
Distribution: Manjaro XFCE
Libvirt: 7.9.0
QEMU: 6.1.0
Kernel: 5.15.2
Kernel Parameters:
- preempt=voluntary
- intel_iommu=on
- iommu.passthrough=1
- vfio-pci.ids=10de:1b80

Virtualization Setup
Disk: RAW & NVME PCI Passthrough
Hypervisor: KVM
Chipset:Q35
Firmware: UEFI x86_64 | OVMF
CPU: Host passthrough with manual, no emulation
IGPU: GVT-G Passthrough
GPU: PCIe Passthrough
Network: virtio NAT, Linked

1.3 — Kernel Versions

As of December 2021, I found that the kernel offering the best performance and easiest setup is version **5.15.2**. Note that kernels version 5.11+ have issues with GVT-G/GVT-D and will tend to freeze up.

2 — Guest CPU Optimization

2.1— CPU mode, topology

The CPU mode you pick will be one of the biggest factor in CPU related performance. If you disable all CPU emulation and pass the CPU as-is to the VM using the "host-passthrough" mode then your performance will be as close to bare metal as can be for CPU bound tasks.

Now, for the guest topology config, the *sockets*, *dies* and *threads* attributes of the *topology* tag must match your physical CPU. The *cores* attribute doesn't but we will talk about it later.

Typically, a consumer motherboard will only have one physical cpu die and socket. As such the *sockets* and *dies* attribute will be '1' in that case. The *threads* attributes will depend on the CPU model and specifies how many virtual cores per physical core your CPU has.

And finally the amount of cores that I want to pass to the VM is configured using the *cores* attribute. The total amount of cores you want to pass to your VM is equal to the number of physical core times (*cores*) the number of virtual cores(*threads*) per core *(cores X threads*).

Int the following example, I only want to pass seven physical cores to the VM and reserve one for the host (See 4.1 — CPU Isolation). Thus, I'm passing a total of 14 cores and will have to set my *vcpu* tag to 14.

```
<domain type='kvm' ...>
  ...
  <cpu mode='host-passthrough' check='none'>
    <topology sockets='1' dies='1' cores='7' threads='2'/>
```

2.2— CPU Pinning

The easiest and most significant way to improve performance so far is to use CPU pinning in conjunction with a host-passthrough CPU mode. The process involves setting some cores to be dedicated to virtualization processes such as I/O, some for your host and others to be dedicated to your virtual machine. It is also possible to isolate the cores so that your host does not use them but I didn't find that it was necessary so long as I didn't run anything intensive on the host while my VMs are running.

First you have to specify the amount of I/O threads that you will be using in the *iothreads* tag. In my case a single I/O thread was sufficient. After defining how many threads are dedicated to IO you will need to assign it a matching amount of cores but we will go over how to do this later. The same goes for the threads assigned to your VM but you will be using the *vcpu* tag to defined how many threads the VM will have.

In my case I have a total of 16 threads on 8 physical cores (2 threads per physical cores) and I decided to reserve my first physical core for the host and other emulation tasks. As such the vcpu 0 and 8 are not assigned to the VM.

Intel CPUs have their CPUsets numerated in a peculiar way. The first virtual core of each cores are occupying the lower half of the cpuset range and the second virtual cores are occupying the upper half.

```
Physical core #1
- Virtual core 1 = 0
- Virutal core 2 = 8
Physical core #2
- Virtual core 1 = 1
- Virutal core 2 = 9
...
Physical core #8
- Virtual core 1 = 7
- Virtual core 2 = 15
```

In my case I decided to assign the physical cores 2 to 8 to the VM using the *vcpuin* tags. And then the emulator and io threads to the first physical core using the *emulatorpin* and *iothreadpin* tags.



```
  <vcpu placement='static'>14</vcpu>
  <iothreads>1</iothreads>
  <cputune>
     <vcpupin vcpu='0' cpuset='1'/>
     <vcpupin vcpu='1' cpuset='2'/>
     <vcpupin vcpu='2' cpuset='3'/>
     <vcpupin vcpu='3' cpuset='4'/>
     <vcpupin vcpu='4' cpuset='5'/>
     <vcpupin vcpu='5' cpuset='6'/>
     <vcpupin vcpu='6' cpuset='7'/>
     <vcpupin vcpu='7' cpuset='9'/>
     <vcpupin vcpu='8' cpuset='10'/>
     <vcpupin vcpu='9' cpuset='11'/>
     <vcpupin vcpu='10' cpuset='12'/>
     <vcpupin vcpu='11' cpuset='13'/>
     <vcpupin vcpu='12' cpuset='14'/>
     <vcpupin vcpu='13' cpuset='15'/>
     <emulatorpin cpuset='0,8'/>
     <iothreadpin iothread='1' cpuset='0,8'/>
  </cputune>
  ...
</domain>
```

2.3— Enlightenments

One way to dramatically improve your Windows guest performance is to enable enlightenments. Now that code 43 is no longer an issue, we can tell Windows that its running in a virtual machine and thus allow it to run code optimized for virtualized contexts.

Enlightenments will also define what features your guest will be able to support such as hibernation, sleep, hyper-v (for nested virtualization) but may also impact performance. I found that enabling some hyper-v features helped with my use case. Also I wanted to have hibernation enabled so I had to enable the right features in the *clock*, *pm* and *features* tags

```
<domain type='kvm' ...>
    ...
    <features>
        <acpi/>
        <apic/>
        <pae/>
        <hyperv>
```



```
            <stimer state='on'/>
            <reset state='on'/>
            ...
        </hyperv>
    </features>
    <clock ...>
        ...
        <timer name='hpet' present='no'/>
        <timer name='hypervclock' present='yes'/>
        ...
    </clock>
    ...
</domain>
```

2.4 — Clock Optimizations & Fixes

Disabling *hpet* helped me with reducing idle CPU usage. Without these settings your CPU could rest in the 15% usage when your VM is running without any workload.

```
<clock offset='localtime'>
   ...
  <timer name='rtc' tickpolicy='catchup'/>
  <timer name='pit' tickpolicy='delay'/>
  <timer name='hpet' present='no'/>
  ...
</clock>
```

3 — Host Optimizations

3.1— CPU Isolation

If you are experiencing micro stutters or missed frame on higher resolutions (4k or 4k ultrawide) it can be beneficial to isolate the cores responsible for the virtualization I/O and emulator processes from the host and guest processes.

What I like to do is set a core for the virtualization, one for the host and assign the rest to the VMs. That is only useful if you plan on doing any work on the host however.

If you do not plan on using your host at all, then isolating the virtualization processes core from the VM cores if the only thing you need to do.

```
# Create the hook scripts directory if it doesn't exist
sudo mkdir -p /etc/libvirt/hooks/qemu.d/

# Download the script file (Github link)
curl -L -o isolate-cpus.sh https://bit.ly/3Gai56Q

# Set the script permissions
chmod +rx isolate-cpus.sh && sudo chown root:root isolate-cpus.sh

# Move it to the hooks folder
sudo mv isolate-cpus.sh /etc/libvirt/hooks/qemu.d/

# Configure your reserved cores and physical cpu topology
sudo nano /etc/libvirt/hooks/qemu.d/isolate-cpus.sh

# See the comments in the script file for more instructions
------------------------------------------------------------------
reserved=YOUR_CORES_RESERVED_FOR_HOST
cores=YOUR_CORE_RANGE_FOR_PHYSICAL_CPU
------------------------------------------------------------------
```

4 — I/O Optimizations

4.1— Using Virtio drivers for disks & network devices

Using virtio drivers is a must to improve the performance of your machine. When used in combination with CPU pinning and IO threads, using the proper drivers may improve the performance of your machine overall. So far we are using an emulated network adapter and an emulated SATA disk which is not optimal for performance as emulating physical devices is much more demanding that running drivers that are optimized for virtualization.

To gain from this change you must have configured your emulator and io threads because when you aren't using dedicated cores, the I/O tasks may be distributed across many cores, some of which may already be busy processing tasks for your game or productivity software. We want to avoid that if possible.

To install the virtio drivers on your guest, we will first need to configure Windows to boot in safe mode, turn off the guest, switch the disk and NIC to virtio in libvirt-manager, boot in Windows and install the drivers using the virtio driver iso. This ISO

```
bcdedit /set {default} safeboot network
```

Once you are booted in safe mode you can install the drivers by opening the Computer Management and then selecting the Device Manager. There should be several unrecognized devices. Right click, select install driver and select them from the ISO.

Once you are done you can return to normal boot by using:

```
bcdedit /deletevalue {default} safeboot
```

Using virtio drivers when possible in conjunction with CPU pinning greatly improved my overall performance and solved issues such as games writing to disk or downloading large files causing stutter.

4.2 — Disable caching for RAW disks

By going through the libvirt interface and disabling caching for RAW disk you can gain some performance.

4.3— Enabling TRIM support for you Virtlo RAW disks

To enable TRIM support select your VirtIO disk in libvirt-manager and open the Advanced Options drawer and then the Performance Options and set “Discard Mode” to “unmap”.

This will improve performance of some tasks such as deleting lots of files. Also it will prolong the life of your SSD.

Another way to greatly improve performance is to use a physical device such as an NVME drive which can then be passed with PCI Passthrough. In which case you do not need to enable trim support. This will greatly improve your I/O performance at the cost of not being able to clone your disk as easily, you will also lose the possibility of hibernating and snapshotting your VM.



to boot your machine the more RAM you add. With 24 Gb passed to one of my guest I experienced wait times of 2 to 3 minutes which are unacceptable. This is due to most linux distribution having preemption enabled for all processes and in the case of qemu, it will make ram allocation extremely slow. This can be fixed by setting preemption to optional, qemu will disable then preemption when appropriate thus speeding your VMs boot time. With 24Gb I achieve cold boot times of 10 to 15 seconds.

Linux kernel version ≤ 5.11

It is necessary to recompile your kernel with Preemption set to optional.

```
CONFIG_PREEMPT_VOLUNTARY=y
# CONFIG_PREEMPT=y
```

The instructions on how to achieve this are outlined in this article.

Linux kernel version ≥ 5.12

As of version 5.12 preemption configuration is now configurable at boot time. Therefore it is now possible to override your distribution's kernel default with a boot parameter.

See the following guide for systems using grub as a bootloader:

```
# Edit your grub file
$ sudo nano /etc/default/grub

# Locate the line containingGRUB_CMDLINE_LINUX_DEFAULT="..."

# Add the following value like such
GRUB_CMDLINE_LINUX_DEFAULT="... preempt=voluntary ..."

# Save the file, quit nano and update grub
$ sudo update-grub

# Reboot
$ sudo reboot now
```



6.1 — Conclusion

As you have seen there are many things that can be done to improve your guest performance. I personally stopped there because I was satisfied with my performance but there are still more steps that could be taken. Notably the use of hugepages has helped many users but its usefulness in the context of GPU pass-through is questionable and the use of core isolation which prevents the hosts from running tasks on core passed to your VM.

If you have any other suggestion do not hesitate to comment and I will your performance tuning interventions to this guide.

6.2 — Sources

https://wiki.archlinux.org/title/PCI_passthrough_via_OVMF

Domain XML format

The root element required for all virtual machines is named domain. It has two attributes, the type specifies the...

libvirt.org

Hyper-V Enlightenments with Libvirt | Jochen Delabie

With Windows 10, it's helpful to enable Hyper-V Enlightenments, to save CPU and increase VM responsiveness.

jochendelabie.com

QEMU

According to the QEMU about page, "QEMU is a generic and open source machine emulator and virtualizer." When used as a...

wiki.archlinux.org

Running Windows 10 on Linux using KVM with VGA Passthrough - Heiko's Blog

Latest update: September 4, 2021 You want to use Linux as your main operating system, but still need Windows for...

Qemu/kvm provides you with a plethora of ways to configure your storage devices. Yet no other type of device shows such...

www.heiko-sieger.info