1.6

REDHAWK *KVM-RT ™ User's Guide*

concurrent REAL-TIME

0898604-1.6
September 2024

Copyright 2024 by Concurrent Real-Time, Inc. All rights reserved. This publication or any part thereof is intended for use with Concurrent Real-Time products by Concurrent Real-Time personnel, customers, and end–users. It may not be reproduced in any form without the written permission of the publisher.

The information contained in this document is believed to be correct at the time of publication. It is subject to change without notice. Concurrent Real-Time makes no warranties, expressed or implied, concerning the information contained in this document.

To report an error or comment on a specific portion of the manual, photocopy the page in question and mark the correction or comment on the copy. Mail the copy (and any additional comments) to Concurrent Real-Time, 800 NW 33 Street, Pompano Beach, FL 33064. Mark the envelope **"Attention: Publications Department**." This publication may not be reproduced for any other reason in any form without written permission of the publisher.

Concurrent Real-Time and its logo are registered trademarks of Concurrent Real-Time, Inc. All other Concurrent Real-Time product names are trademarks of Concurrent Real-Time while all other product names are trademarks or registered trademarks of their respective owners. Linux® is used pursuant to a sublicense from the Linux Mark Institute.

Printed in U. S. A.

Revision History:	Level:	Effective With:
July 2019	1.0	RedHawk Linux 7.5
January 2020	1.1	RedHawk Linux 8.0
February 2021	1.2	RedHawk Linux 8.2
October 2021	1.3	RedHawk Linux 8.4
March 2023	1.4	RedHawk Linux 8.4
December 2023	1.5	RedHawk Linux 9.2
September 2024	1.6	RedHawk Linux 9.2.3

Preface

Scope of Manual

This manual provides information and instructions for using Concurrent Real-Time's RedHawk KVM-RT™.

Structure of Manual

This manual consists of:

- Chapter 1 introduces you to KVM-RT.
- Chapter 2 explains the steps in setting up and booting virtual machines in KVM-RT.
- Chapter 3 covers how to configure KVM-RT.
- Chapter 4 summarizes all the KVM-RT tools.
- Chapter 5 discusses time synchronization.
- Chapter 6 discusses I/O utilization in Virtual Environment.
- Chapter 7 discusses ways to analyze and debug guest VMs in KVM-RT.
- Appendix A covers NUMA node mappings of the Supermicro M12SWA-TF platform.
- Appendix B gives instructions in setting up single root I/O virtualization (SR-IOV).
- Appendix C lists the boot parameters that may be required for PCI Passthrough.

Syntax Notation

The following notation is used throughout this manual:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms may also appear in *italic*.
`list bold`	User input appears in **`list bold`** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **`list bold`** type.
`list`	Operating system and program output such as prompts, messages and listings of files and programs appears in `list` type.
`[]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify these options or arguments.

hypertext links — When viewing this document online, clicking on chapter, section, figure, table and page number references will display the corresponding text. Clicking on Internet URLs provided in blue type will launch your web browser and display the web site. Clicking on publication names and numbers in red type will display the corresponding manual PDF, if accessible.

Related Publications

The following table lists Concurrent Real-Time documentation. Depending upon the document, they are available online on RedHawk Linux systems or from Concurrent Real-Time's documentation web site at **http://redhawk.concurrent-rt.com/docs**.

RedHawk KVM-RT	**Pub. Number**
RedHawk KVM-RT Release Notes	0898603
RedHawk KVM-RT User's Guide	0898604
RedHawk Architect	
RedHawk Architect Release Notes	0898600
RedHawk Architect User's Guide	0898601
RedHawk Linux	
RedHawk Linux Release Notes	0898003
RedHawk Linux User's Guide	0898004
RedHawk Linux Cluster Manager User's Guide	0898016
RedHawk Linux FAQ	N/A
NightStar RT Development Tools	
NightView User's Guide	0898395
NightTrace User's Guide	0898398
NightProbe User's Guide	0898465
NightTune User's Guide	0898515

Contents

1
Introduction to KVM-RT

This chapter provides a general overview and requirements for using RedHawk KVM-RT.

Introduction

RedHawk KVM-RT is a Real-Time Hypervisor solution that utilizes Quick Emulator (QEMU) and Kernel-Based Virtual Machine (KVM) and RedHawk real-time features to extend RedHawk's real-time determinism to guest RedHawk virtual machines. It supports multiple guests, both real-time and non-real-time, running in virtual machines on a single host system.

RedHawk KVM-RT leverages the capabilites of RedHawk Linux together with QEMU and KVM to give users the ability to:

- virtualize legacy operating systems
- reduce physical system hardware footprint
- run real-time and non-real-time virtual machines in a secure, isolated environment
- provide I/O implementation solutions that allow VMs to perform close to their bare-metal configuration

Host System Requirements and Installation

Refer to the *RedHawk KVM-RT Release Notes* for hardware host system requirements and software installation instructions.

Though not a requirement, it is highly recommended that the entire host system be dedicated to running the Real-Time Hypervisor. Administrators of the KVM-RT host system must be careful not to disturb CPU shielding or CPU affinities on the system, or else real-time performance of virtual machines may be compromised.

KVM-RT requires that a RedHawk kernel is booted on the host system while KVM-RT is being used. Additional system configuration may be required. For example, PCI Passthrough may require the addition of boot time parameters. See Appendix C, PCI Passthrough Boot Parameters, for more information.

Once KVM-RT is installed, the following command can be run to test the suitability of the host system.

```
$ sudo kvmrt-validate-host
```

2
Getting Started

This chapter explains the steps in setting up and booting virtual machines in KVM-RT. Also discussed are the various QEMU/KVM threads that run on the host for each virtual machine.

Building Virtual Machines

KVM-RT works with virtual machines that have been created and configured within the libvirt framework. A virtual machine may be created and configured within libvirt in several ways, including:

- with Virtual Machine Manager
- with RedHawk Architect
- by cloning another virtual machine

Detailed instructions on how to build virtual machines are beyond the scope of this book but are well documented. General instructions and references to documentation are given in the following sections.

Real-time virtual machines must contain a guest OS of RedHawk Linux 7.0 or later. The guest CPU architecture must match that of the host.

Using Virtual Machine Manager to Create a Virtual Machine

The Virtual Machine Manager is a GUI tool that can be used to create, configure, and manage virtual machines within the libvirt framework.

Start Virtual Machine Manager by running:

```
$ sudo run virt-manager
```

See the **virt-manager(1)** man page for more information.

Using RedHawk Architect to Create a Virtual Machine

RedHawk Architect is an optional product offered by Concurrent Real-Time that specializes in creating, customizing and deploying RedHawk Linux disk images.

Architect can be used to create a RedHawk virtual machine and to export it to the Virtual Machine Manager. Detailed instructions can be found in the documentation that comes with RedHawk Architect. Below are the general steps required:

- run Architect
- create a new session and configure the image as desired
- build the image
- deploy the image to a virtual machine
- export the virtual machine to Virtual Machine Manager

Cloning a Virtual Machine Image

Any existing virtual machine within the libvirt framework can be cloned by using the virt-clone command. For example:

```
$ sudo virt-clone -o old_vm -n new_vm
```

See the **virt-clone(1)** man page for more information.

Importing Virtual Machines into KVM-RT

Once virtual machines have been created within the libvirt framework, they can be imported into KVM-RT.

All libvirt virtual machines can be imported into KVM-RT with the following command:

```
$ sudo kvmrt-import
```

This command may be run at any time new VMs are created. Run **kvmrt-import --help** for more information and options.

When a VM is imported into KVM-RT it inherits the VM configuration settings from libvirt. Once this is done a VM may be further configured with KVM-RT as needed. See "Configuring Virtual Machines" in Chapter 3 for more information.

Booting and Shutting Down Virtual Machines

A **systemd** service, named *kvmrt*, exists for KVM-RT. It may be enabled so that the VMs configured with autostart set, will be automatically booted during system start-up and VMs running when the system is shutting down will be shut down. The service is not enabled by default. Once enabled, the service will automatically start on the next boot. If you want it to start it immediately, you must enable the service and start it as follows:

```
systemctl enable kvmrt
```

```
systemctl start kvmrt
```

Note that the service start will fail if there are any VMs running since it invokes **kmvrt-boot** with the **--clean** option.

The following KVM-RT tools can be used to boot, shutdown, and view the status of VMs.

To start up all configured VMs:

```
$ sudo kvmrt-boot
```

To shut down all the running VMs:

```
$ sudo kvmrt-shutdown
```

To query the state of all VMs:

```
$ sudo kvmrt-stat
```

Individual VMs can be specified to all these commands. For example:

```
$ sudo kvmrt-boot RedHawk-8.4VM Windows10VM

$ sudo kvmrt-shutdown RedHawk-8.4VM Windows10VM
```

Note that by default VMs are brought down in parallel. If the **-v** (verbose) option is used, the shutdown will be serialized so that the output from the different VMs is not garbled together.

Run any of the above commands with the **--help** option for more information and options.

Understanding QEMU/KVM Threads

QEMU/KVM runs multiple threads for each virtual machine. The names and purpose of these threads are as follows:

qemu-kvm

These are emulator threads. There may be two or more of these.

qemu-system-x86

This is an alternate name for *qemu-kvm* in some distributions.

worker

These are dynamically created threads for long I/O operations being performed by the emulator.

SPICE Worker

This is a thread for a virtual console.

IO mon_ioth

This is an optional thread used for some I/O.

CPU n/KVM

These are virtual CPU (vCPU) threads. There will be one per virtual CPU, where *n* is the vCPU ID.

Use the **kvmrt-stat -t** command to display information about all currently running VM threads.

3
Configuring Virtual Machines

Virtual machines that are configured within the libvirt framework have an XML configuration file that controls all attributes of the virtual machine.

This file usually exists as "**/etc/libvirt/qemu/{DOMAIN}.xml**" for the given VM domain name and is created when the VM is created or imported into the libvirt framework. This file gets updated when VM configuration changes are made in the Virtual Machine Manager.

KVM-RT uses a simplified configuration file, explained below, to manage multiple VMs. KVM-RT updates libvirt XML configuration files as needed to keep the two files in sync.

The KVM-RT Configuration File

The default location of the KVM-RT configuration file is **/etc/kvmrt.cfg,** but all **kvmrt-*** tools that use a configuration file accept a **-f** option that allows the user to specify an alternate configuration file.

The KVM-RT configuration file uses the INI file format, where each section describes a VM. The first line of each section is the UUID, a unique VM identification number generated by libvrt. An example configuration follows with two guest VMs, the second one not in use (disabled):

```
[fde74e84-0e1b-404e-90e7-72101e79c48a]
name = RedHawk-8.4-RT
title = Real-Time RedHawk 8.4
description = Configured for real-time
nr_vcpus = auto
cpu_topology = auto
cpuset = n1,n2
rt = True
rt_memory = auto
numatune = auto
hide_kvm = False
autostart = True
disabled = False
pcidevs = 0000:21:00.0 0000:22:00.0 0000:22:00.1
comments = remember to change autostart to true after
    testing

[aeec46cc-0638-4949-ac04-146b233194a9]
name = RedHawk-8.4
title = RedHawk8.4
description = RedHawk8.4VM
nr_vcpus = 2
```

```
cpu_topology = auto
cpuset =
rt = False
rt_memory = auto
numatune = auto
hide_kvm = False
autostart = True
disabled = True
pcidevs =
comments = This VM is not used anymore; kept for
    reference
```

Defined below are the field types used in the attribute description that follows:

{string}: any string

{int}: any integer

{bool}: **true** | **false** | **on** | **off** | **yes** | **no** | **1** | **0** (case-insensitive)

{ID-set}: a string that describes a set of ranges of integers in a human-readable form such as "0,2,4-7,12-15"

{CPUSET}: can be specified as a comma-separated list of CPUs or CPU ranges (eg. 0,1,16-19) but also as integers prefixed with 'n' for NUMA node, 'c' for core, 'd' for die, or 'p' for package. Additionally, the string may be prefixed with '~' to create an inverse set (eg. ~n0).

{PCISET}: a list of space-separated full PCI bus addresses. Each device is described by the syntax "domain:bus:device.function"(eg. 0000:21:00.00).

Each VM may be configured with the following attributes. Note that if an attribute is not set or it is missing, the default value is used.

name = { string }

This attribute sets the VM name. This is an arbitrary, user specified name that must be unique to libvrt.
There is no default value, this attribute must be set but it can be changed.

title = {string}

This attribute sets the VM title.
The default value is "".

description = {string}

This attribute sets the VM description.
The default value is "".

nr_vcpus = {int } | **auto**

This attribute defines the number of virtual CPUs in the VM. When set to **auto**, the number of virtual CPUs will automatically be set to the number of physical CPUs defined in '**cpuset**' -1. Note that when **rt** is true, hyperthreaded siblings are downed and hence not counted in the **cpuset** calculation.

The default value is **1**

cpu_topology = {int}, {int}, {int} | **auto**

This attribute defines the CPU topology that is seen by the VM.

If not **auto**, the value must be a string of three positive integers separated by commas ("sockets, cores, threads"), to describe the CPU topology. sockets is the number of CPU sockets, cores is the number of cores per socket, and threads is the number of threads per core.

When the value is **auto**, the topology is set to one socket, *nr_vcpus* cores per socket, and one thread per core.

The default value is **auto**.

NOTE

If the guest virtual machine is running a Windows operating system, the *cpu_topology* attribute may have been set to a default value that will not work well in KVM-RT. It is best to change this setting to **auto**. See the item labeled "VMs running the Windows operating system" in the Known Issues section of the KVM-RT Release Notes document.

cpuset = {CPUSET}

This attribute defines host CPUs to which all VM threads are biased. CPUSET is defined with the other field types above. See the section "Understanding the cpuset Configuration Attribute" later in this chapter for more information.

The default value is "" (no CPU biasing).

rt_memory = {bool} | **auto**

This attribute enables memory locking of all pages used by the VM.

When the value is **auto**, this option is enabled if the *rt* attribute is enabled and disabled if *rt* is disabled.

The default value is **auto**.

numatune = {ID-set} | **auto**

This attribute sets the host NUMA node(s) to be used for memory allocation to the VM.

If not **auto**, the value must describe a set of host NUMA node IDs. The set may be empty, in which case memory will not be restricted to any host NUMA nodes.

When the value is **auto**, all NUMA nodes used by *cpuset* will be used. If *cpuset* is empty then memory will not be restricted to any host NUMA nodes.

The default value is **auto**.

hide_kvm = {bool}

This attribute hides KVM from the view of the guest OS in the VM.

The default value is **false** (do not hide KVM).

rt = {bool}

This attribute configures the VM for real-time.

The *cpuset* and *rt_memory* attributes must be configured (enabled) when this attribute is enabled. It is also recommended to configure and enable *numatune* when this attribute is enabled.

The default value is **false** (not real-time).

autostart = {bool}

This attribute enables auto-starting of the VM with **kvmrt-boot**.
The default value is **false** (do not autostart).

disabled = {bool}

When set to true the VM is hidden from KVM-RT. This provides a way to save a VM configuration that is not in use.

The default value is **false** (the VM is enabled).

pcidevs = {PCISET}

A list of space-separated full PCI device bus addresses that KVM-RT will pass through to the VM. A device's bus address may be obtained by searching for the device in the output of the **pci(1)** command.

Note that all the devices in the same IOMMU group must be passed through to the same VM. The **pci** command, by default, lists only common devices and may miss other devices in the IOMMU group. To see all the devices, add the **-a** option to the **pci** command or let **kvmrt-edit-config** fail and print the device bus addresses of the missing devices.

The default value is "".

comments = { string }

A place for user comments. For multiple lines of comments, indent the additional line(s) with a space or TAB.

The default value is "".

Configuration Tools

A KVM-RT configuration can be edited by running the command:

```
$ sudo kvmrt-edit-config
```

Note that KVM-RT configuration files should not be edited directly. **kvmrt-edit-config** validates and also synchronizes the configuration with **libvirt**.

A KVM-RT configuration, as interpreted by KVM-RT, can be displayed by running the command:

```
$ sudo kvmrt-show-config
```

The **kvmrt-validate-config** and **kvmrt-sync-config** commands can be run to validate and synchronize, respectively, a configuration. Users do not normally need to run these commands directly when using **kvmrt-edit-config**.

Run any of the above commands with the **--help** option for more information and options.

Advanced Libvirt Configuration

Advanced configuration that is beyond the scope of the KVM-RT configuration file may be made to the **libvirt** XML files, using Virtual Machine Manager or 'virsh edit', but additional synchronization and validation steps are required for KVM-RT. This is also true when you remove a VM from **libvrt.**

Note that some combinations of configuration may be invalid and users are encouraged to make configuration changes by editing the KVM-RT configuration file with **kvmrt-edit-config** whenever possible.

If libvirt XML files are modified by the user outside of KVM-RT, then it is necessary to run **kvmrt-sync-config -r** and **kvmrt-validate-config**, like so:

```
$ sudo kvmrt-sync-config -r
$ sudo kvmrt-validate-config
```

Also note that **kvmrt-import -u** may be used instead of **kvmrt-sync-config -r**, as in:

```
$ sudo kvmrt-import -u
$ sudo kvmrt-validate-config
```

The **kvmrt-validate-config** command will display appropriate errors or warnings for any invalid configuration.

Run any of the above commands with the **--help** option for more information and options.

Understanding the cpuset Configuration Attribute

The *cpuset* attribute controls host-CPU-biasing of the QEMU/KVM threads of a virtual machine.

The *cpuset* attribute may be used for both real-time and non-real-time VMs.

For non real-time VMs, all the CPUs in the cpuset can be allocated to any QEMU/KVM thread. Under-provisioning of host CPUs (less CPUs in *cpuset* than *nr_vcpus* +1) results in more than one vCPU being biased to a host CPU. If *cpuset* is empty then the VM will not be bound to any particular host CPUs.

For real-time VMs, host CPUs in *cpuset* are assigned to vCPUs in order, starting with the lowest numbered CPU. The rest of the CPUs (at least one more is required) are used for non-vCPU threads. Under-provisioning of host CPUs is not allowed for real-time VMs and *cpuset* is not allowed to be empty.

Understanding KVM-RT Use of RedHawk Real-Time Features

When the *rt* configuration attribute is enabled in the configuration file, the following RedHawk real-time system features are performed:

- All the CPUs in *cpuset* are shielded. See **shield(1)**.
- Hyperthreaded siblings are downed. See **cpu(1)** and "KVM-RT Use of Threaded CPUs" in the section below.
- Memory Locking is enabled. See the **-L** option of **run(1)**.
- IRQ affinities on the host may be modified. See below.

For each real-time VM using PCI passthrough devices, the affinity of all related vfio IRQs are changed to bind the IRQ to the last host CPU used by the real-time VM. This is done constantly in the background whenever VMs are running. All other device IRQs on the host have their affinity changed as needed in order to guarantee that they will not be bound to any of the host's CPUs used by any real-time VMs in the configuration.

It is recommended that when the *rt* configuration attribute is enabled, that *numatune* also be enabled. When *numatune* is enabled NUMA nodes specified are to be used for memory allocation to the real-time VM. See **NUMA(7)**.

KVM-RT Use of Threaded CPUs

On host systems having a threaded-CPU architecture such as Intel's Hyper-Threading or AMD's SMT, KVM-RT gives special treatment to multi-threaded CPU cores when a real-time VM is in use.

Real-time demands that only one threaded sibling CPU be in use to avoid contention of CPU core resources (e.g. caches, etc.). To ensure this, KVM-RT shuts down all but one threaded sibling CPU for each CPU core allocated to a real-time VM. This requires some consideration when assigning VM *cpuset*s.

A real-time VM will be given ownership of all threaded sibling CPUs that are related to the CPUs specified in its *cpuset*. This may result in the VM consuming but not using more CPUs than it has specified in it's *cpuset*. Only one CPU per threaded core will be used for real-time and the others will be shutdown.

No special treatment is given to threaded cores hosting non-real-time VMs.

Configuring Real-Time Virtual Machines

Perform the following steps to configure a VM for real-time:

- enable the *rt* configuration attribute
- enable the *rt_memory* attribute (**auto** is recommended)
- consider enabling the *numatume* attribute (**auto** is recommended)
- configure the *cpuset* attribute as described below

Configuring the *cpuset* attribute for a real-time VM requires some understanding of the host system's CPU topology. Use the **hwtopo** or the **cpustat** command to see a display of the host system's CPU topology. **hwtopo** displays the layout of NUMA nodes, CPU cores, and logical CPUs. The following example shows the command output for a multi-threaded architecture with multiple NUMA nodes:

```
$ hwtopo -v --no-io
Machine 0 (Supermicro M12SWA-TF, "TEST_MACH1"):
   Package 0 (AMD Ryzen Threadripper PRO 5975WX 32-
                Cores):
      L3 Cache (32MiB):
         NUMA Node 0 (31GiB)
         Core 0:
            CPU 0
            CPU 32
         Core 1:
            CPU 1
            CPU 33
         Core 2:
            CPU 2
            CPU 34
         Core 3:
            CPU 3
```

```
            CPU 35
         Core 4:
            CPU 4
            CPU 36
         Core 5:
            CPU 5
            CPU 37
         Core 6:
            CPU 6
            CPU 38
         Core 7:
            CPU 7
            CPU 39
      L3 Cache (32MiB):
         NUMA Node 1 (31GiB)
         Core 8:
            CPU 8
            CPU 40
         Core 9:
            CPU 9
            CPU 41
         Core 10:
            CPU 10
            CPU 42
         Core 11:
            CPU 11
            CPU 43
         Core 12:
            CPU 12
            CPU 44
         Core 13:
            CPU 13
            CPU 45
         Core 14:
            CPU 14
            CPU 46
         Core 15:
            CPU 15
            CPU 47
      L3 Cache (32MiB):
         NUMA Node 2 (31GiB)

...
```

The following rules should be observed when configuring a real-time VM for optimal performance. The KVM-RT tools will display appropriate errors or warnings when any of the rules are violated. Errors must be corrected to continue, but warnings serve as reminders that your configuration may not be optimal.

- The *cpuset* of a real-time VM cannot overlap the *cpuset* of any other VM.
- The *cpuset* of a real-time VM must not be under-provisioned for the number of CPUs configured in the *nr_vcpus* attribute.

- Careful consideration should be given if the *cpuset* of a real-time VM spans multiple NUMA nodes.
- Careful consideration should be given if the *cpuset* of any other VM shares NUMA nodes with a real-time VM.
- Careful consideration should be given if *numatune* is not enabled for a real-time VM, or if the *numatune* node set is not contained within the NUMA nodes used by the *cpuset*.
- Careful consideration should be given if the *numatune* node set of any other VM overlaps with the NUMA nodes used by a real-time VM's *cpuset*.
- The *cpusets* of all real-time VMs must not consume all host CPUs. This is because some CPUs must be available for the KVM-RT host OS.

Adhering to the following recommendations will help simplify real-time VM configuration:

- Always configure *cpuset* with a least *nr_vcpus* + 1 host CPUs.
- Do not configure the *cpuset* of any other VM to conflict with this VM's *cpuset*, or to use any other CPUs in a NUMA node used by this VM.
- Do not let the *cpuset* span multiple NUMA nodes.
- Set *numatune* to **auto**.
- Do not configure the *numatune* of any other VM to include the NUMA node used by this VM.
- Use the **kvmrt-show-config** command to view the real-time policy configured for all VMs.
- Use the **kvmrt-stat -t** command to display the CPU-biasing of all currently running VM threads.

4
KVM-RT Tools

RedHawk's real time tools included in the **ccur-rttools** package are shipped with both RedHawk Linux and KVM-RT. Both sets of tools, RedHawk's Real-time tools and KVM-RT tools are described here.

RedHawk's Real-time Tools

These commands can be used to examine and modify the system to achieve the best performing KVM-RT configuration possible.

Only a very brief description is listed here. For more information see the man pages or use the *--help* option provided for each tool. There is also a **rttools(7)** man page that lists all these commands.

Command line interfaces

cpus
Displays or changes the state of CPUs.

irqs
Displays information about IRQs on the system.

tasks
Displays information about tasks (process threads) on the system.

hwtopo
Displays system hardware topology.

pci
Displays PCI device information.

irq-affinity
Displays or changes the CPU affinity of IRQs.

task-affinity
Displays or changes the CPU affinity of tasks (process threads).

cpustat
Combines hardware topology, CPU state, IRQ and task execution and CPU affinity into one display.

cpi
Displays per-CPU interrupt counts of IRQs.

Graphical user interfaces

hwtopo-gui
Displays system hardware topology with a GUI.

cpustat-gui
Combines hardware topology, CPU state, IRQ and task execution and CPU affinity into one display with a GUI.

interview
Displays interrupt counts per-CPU in real-time with a GUI.

KVM-RT Tools

These commands are KVM--RT specific. There is a man page and a *--help* option associated with each command. There is also a **kvmrt(7)** man page that lists all these tools.

Most of the KVM-RT tools use the **/etc/kvmrt.cfg** file by default, however, a different configuration file may be specified via the **-f** option.

Start-Up Commands

kvmrt-validate-host:

Verifies if the current system configuration is valid for a KVM-RT host. It will provide suggestions on changes to be made if not.

kvmrt-import:

Imports **libvirt** virtual machines into a KVM-RT configuration file. By default, all libvirt VMs on the current system will be imported, but individual VMs may be specified instead. Any VMs already listed in the KVM-RT configuration file will be skipped, unless the **-u** option is used.

Configuration Commands

kvmrt-edit-config:

Allows a user to edit, validate, and synchronize the KVM-RT configuration file **/etc/kvmrt-edit-config**. This is the default configuration file but another may be specified with the **-f** option.

kvmrt-show-config:

Displays the configuration of virtual machines in a KVM-RT configuration. Options are available to control the information to display.

kvmrt-sync-config:

Synchronizes **libvirt** VM configuration **XML** files with a KVM-RT configuration file. By default, all VMs in the KVM-RT configuration file are

synchronized, but individual VMs may be specified instead. Optionally you can just query the state.

kvmrt-validate-config:

All VMs in the configuration are individually validated as well as the combined VMs are evaluated for conflicts. By default, only VMs in the configuration that are not disabled are evaluated but **-all** option may be used to override.

Boot/Shutdown Commands

kvmrt-boot:

Boots virtual machines in a KVM-RT configuration, after validating the configuration. By default, all VMs in the configuration with the "autostart" configuration parameter enabled are booted, however the **--all** option can be used to boot all VMs in the configuration. Individual VMs may be also be specified instead.

If any VMs are running it simply re-tunes them as required for real-time and boot errors are ignored. When the option **--clean** is specified, no virtual machine may already be running and no boot errors are tolerated.

kvmrt-shutdown:

Shuts down virtual machines and removes any real-time policy used by those VMs. By default, all VMs in the configuration are shutdown in parallel, but individual VMs may be specified instead. Using the **-v** verbose option will serialize the shutdown so that the output from the shutdown is not garbled. A **--force** option is available.

kvmrt-stat:

Displays the status of virtual machines in a KVM-RT configuration. By default, all enabled VMs are shown but the **--all** option will also show disabled VMs. Individual VMs may also be specified instead.

5
Virtual Machine Time Synchronization

Chrony is a versatile time synchronization implementation of NTP. It is designed to perform well in a wide range of conditions and can be run on virtual machines. Specific instructions are included here on how to configure and start the chrony system on virtual machines. The host system is assumed to be already configured with time synchronization. See **chronyd(1)**, **chrony.conf(5)** and on-line documentation for more information.

Complex applications may depend on the time of day to be synchronized between two or more VMs or with the host. It is also required that the time of day on the virtual guests be synchronized with the host when using RedHawk tracing to analyze performance issues or debug system problems with real-time VMs.

Instructions to run chrony

There are various techniques to synchronize the time of day clock on the virtual guests but we recommend kvm_clock synchronized with chrony via the **ptp_kvm** module.

The process of configuring **chronyd** to use **ptp_kvm** differs slightly depending on the base distribution.

If you are using Ubuntu as your base distro, use these settings:

```
service=chrony
conf=/etc/chrony/chrony.conf
drift=/var/lib/chrony/chrony.drift
```

If you are using a Rocky-compatible distro, use these settings:

```
service=chronyd
conf=/etc/chrony.conf
drift=/var/lib/chrony/drift
```

The following instructions should help in configuring chrony on a virtual guest. Substitute the variable settings below for the appropriate distro settings above.

1. If not already installed, install chrony.

   For Rocky-compatible systems:

   ```
   dnf install chrony
   ```

   For Ubuntu systems use:

   ```
   apt install chrony
   ```

2. Load the ptp_kvm module on boots.

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

3. See if there are any lines that reference 'refclock', 'server', 'pool' or 'peer' in the configuration file. If there is, edit the file and comment them out (put a # sign in front).

```
egrep 'refclock|server|pool|peer' $conf
[ "$?" = 0 ] && vi $conf
```

4. Configure 'refclock".

```
echo "refclock PHC /dev/ptp_kvm poll 3 dpoll -2 \
    offset 0" >> $conf
```

5. Comment (place a # at the front) any lines with PEERNTP and append PEERNTP=no to the /etc/sysconfig/network file.

```
grep PEERNTP /etc/sysconfig/network && \
    vi /etc/sysconfig/network
echo "PEERNTP=no" >> /etc/sysconfig/network
```

6. Remove the appropriate $drift file.

```
rm -f $drift
```

7. Enable the appropriate chronyd service but do not start it.

```
systemctl enable $service
```

8. Reboot for a clean start with the new configuration.

```
reboot
```

6
I/O Device Utilization in Virtual Environment

This chapter covers key aspects of virtualization within the context of Linux Quick Emulator (QEMU) and Kernel-based Virtual Machine (KVM). It aims to provide users with a better understanding of the underlying techniques within virtualization so that informed decisions can be made in terms of performance and reduced latency when configuring common I/O devices.

The chapter covers different virtualization techniques, including device emulation, paravirtualization, and PCI passthrough, which provide varying levels of performance and host hardware interaction. This provides a foundation for understanding the device types.

Three common device types, networking, storage and graphics, are also discussed including several implementations for each which can be separated into virtual and physical hardware solutions.

Introduction

In Linux, QEMU/KVM make up the hypervisor. QEMU provides hardware emulation including CPU, memory, network cards, disk controllers, display adapters and more.

KVM provides core virtualization capabilities by exposing hardware virtualization features such as Intel VT-x and AMD-V to user-space. It relies on libraries such as libvirt to manage virtual machines. KVM also handles the virtualization of the CPU, allowing virtual machines to execute instructions directly on the host's physical CPU.

Virtualization Techniques

There are three common techniques for making I/O devices available to a virtual machine: Device Emulation, Paravirtualization and PCI Passthrough.

Device Emulation

Full device emulation works by simulating the hardware devices the guest operating system uses. This allows the guest OS to run in a state where it believes it is interacting with real hardware.

The guest OS will interact with I/O devices using standard I/O operations, except device interrupts are emulated and QEMU intercepts and handles them. The work is often forwarded to the host OS CPU for processing or handled directly by the emulator.

Device emulation does incur some performance overhead since interrupts are handled via software instead of by the hardware device it would have normally been intended for on a physical system. This method can be useful for older guest operating systems where paravirtualization is not an option or the drivers are not available.

Paravirtualization

Paravirtualization improves the performance of the virtual machine by modifying the guest OS to be aware that it is running in a virtual environment. By being aware of an underlying hypervisor, the guest OS can bypass the emulation layer for certain operations and interact directly with the hypervisor.

Virtio is the interface with which QEMU/KVM provides paravirtualization support. It provides a standardized API for devices. The guest OS will need support for drivers such as virtio-pci for various devices. For example: networking (VirtioNet), storage controllers (Virtio Block, Virtio SCSI), and graphics (Virtio GPU).

The virtio driver is responsible for accepting I/O requests from user processes within the guest OS, forwarding those I/O requests to the appropriate virtio device within QEMU, and retrieving completed requests from the virtio device.

The virtio device will accept the I/O requests from the corresponding virtio driver, offload those I/O requests to the host's physical hardware, and make the result available to the virtio driver.

While there is a significant improvement in virtual machine performance when utilizing the optimizations present in virtio, there is still some overhead. All virtual machines utilizing paravirtualization still contend for host resources and there can still be some latency expected when compared to dedicating physical hardware completely to the VM using PCI passthrough.

For further reading:

https://blogs.oracle.com/linux/post/introduction-to-virtio

PCI Passthrough

PCI passthrough offers the best performance by reducing host kernel involvement in executing virtual machine I/O operations. PCI passthrough may require some kernel boot parameter configuration depending on your system. See Appendix C, PCI Passthrough Boot Parameters.

Some benefits of PCI passthrough include secure isolation from host memory and other VMs along with direct hardware access providing near bare-metal performance.

One of the drawbacks to PCI passthrough is complete resource allocation. This solution does not scale well due to resource limitations and prevents the host and other guests from using the device assigned to a specific VM. While this is a drawback, PCI passthrough provides the best real-time performance over the methods mentioned above.

To understand how PCI passthrough works we must discuss the mechanisms with which the host OS prepares an environment for a physical device to be allocated to a virtual machine. These mechanisms include IOMMU and VFIO.

IOMMU

Input-Output Memory Management unit (IOMMU) provides secure and efficient mapping of device memory access. This provides memory isolation such that the VM can only access the memory for the assigned device. IOMMU also provides interrupt remapping to ensure interrupts generated by the device are properly routed to the VM's CPUs.

Further reading:

https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/sect-iommu-deep-dive#sect-iommu-deep-dive

VFIO

The Virtual Function I/O (VFIO) driver is an IOMMU/device agnostic framework for exposing direct device access to userspace in a secure, IOMMU protected environment. This allows VMs to utilize their device drivers to directly interact with physical disk devices and bypass the host kernel.

VFIO depends on IOMMU to safely assign a PCI device to a VM by setting up the necessary memory mappings and providing isolation from the host. IOMMU isolation is not always at the individual device level. Properties of devices, interconnects, and IOMMU topology can reduce isolation to a group of devices. It should be noted that VFIO operates at the IOMMU group level and all devices within the same IOMMU group must be passed through to the same VM.

Further reading:

https://docs.kernel.org/driver-api/vfio.html

Networking

A virtual machine can use virtual and physical network devices. Virtual network devices can be configured for NAT or MacVTap. Physical network devices are passed through from the host.

Virtual Networking

Virtual networking can be configured for NAT or MacVTap on virtual device models such as virtio, e1000e and rt18139.

NAT

By default libvirt installations use Network Address Translation (NAT) with forwarding for network configuration on virtual machines. The host system will have an isolated bridge device, 'virbr0', without any physical interfaces added.

Libvirt adds iptables rules to manage traffic to and from the guest. Outgoing connections from the guest pass through the virtual network and are forwarded via the host IP address to their destination. External connections cannot initiate communication with the guest, however the guest can accept incoming communication from other guests, as well as the host, using virbr0.

For a guest to be accessible from outside the bridge an explicit static destination network address translation (DNAT) with port forwarding has to be configured on the host.

Further readings:

https://wiki.libvirt.org/Networking.html
https://wiki.libvirt.org/VirtualNetworking.html

MacVTap

MacVTap provides a simplified solution for virtualized bridged networking. This is not to be confused with network bridges created via **`brctl(8)`**, **`nmcli(1)`**, or other methods of creating a Linux ethernet bridge.

MacVTap combines the Macvlan driver and a tap device. It is an instance created on top of a physical interface along with a character device directly used by KVM/QEMU. It extends an existing host network interface along with providing its own MAC address on the same ethernet segment for the guest to use.

Guests will appear on the same switch the host is connected to. This offers an advantage over NAT networking where incoming connections cannot reach the guest. With MacVTap, incoming connections can reach the guest.

MacVTap can be set up in one of three modes which define the communication between endpoints:

Virtual Ethernet Port Aggregator (VEPA)
Bridge
Private

Most modern libvirt installations now use bridge mode and it's the mode discussed here.

Bridge allows all endpoints to be directly connected to each other. While guests using MacVTap can communicate on the same network the host uses, they can also exchange frames directly with other guests using the same MacVTap bridge connection without routing through the external network. Inter-guest transfer speeds are similar to what NAT networking provides while external network speeds are limited to the physical network setup.

One limitation of MacVTap is that it cannot readily enable network communication between the KVM host and any of the KVM guests. One possible workaround is to have

multiple network interfaces in the KVM host and configure another host interface to communicate with the guest.

Further reading:

https://virt.kernelnewbies.org/MacVTap
https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces-for-virtual-networking#macvlan
https://docs.kernel.org/networking/tuntap.html
https://wiki.libvirt.org/TroubleshootMacvtapHostFail.html

Virtual Networking Device Models

Virtual Network device models are virtualized network cards presented as PCI devices within the guest. These device models utilize the backend setup of either NAT of MacVTap as described above.

These common device models are available for both MacVTap and NAT backend setups:

virtio
e1000e
rtl8139 (hypervisor default)

Virtio is a paravirtualized driver that works in guest operating systems that support virtio. The guest knows it is running within a virtual machine and allows direct calls to the hypervisor. This method greatly improves transfer speeds and avoids the need for fully emulated network devices.

The remaining device models, e1000e and rtl8139, are emulated Intel and Realtek network cards respectively. Depending on the limitations of what network cards the guest OS can support, one of these options might be needed in lieu of virtio.

Examples of PCI device listing in the guest for each device model:

virtio:
Ethernet controller: Red Hat, Inc. Virtio 1.0 network device (rev 01)

e1000e:
Ethernet controller: Intel Corporation 82574L Gigabit Network Connection

rtl8139:
Ethernet controller: Realtek Semiconductor Co., Ltd.
RTL-8100/8101L/8139 PCI Fast Ethernet Adapter (rev 20)

Device Model Performance Comparison

When configuring virtual network devices it is important to note the differences in performance.

Each device model was compared using communication between 2 guests on the same bridge in both NAT and MacVTap configurations. The sampling was done using **`iperf3`** TCP tests over a 5 minute period with measurement intervals being one second (60 secs/min x 5 minutes= 300 samples). Tests were completed both with and without stress

on the guest OS. Below is the average giga bits per second measured over the sampling period:.

Device Model	Without Stress (Gbps)	With Stress (Gpbs)
virtio	10.3	4.44
e1000e	2.12	1.12
rt18139	0.55	0.44

Speeds between NAT and MacVTap configurations, when directly comparing device models, are nearly identical and therefore not distinguished in the results.

Test results for guests configured with MacVTap communicating outside of the host would depend upon the user's network limitations. The same can be said for guests configured with NAT sending data outside the virbr0 device.

Note that these measurements were taken on systems not utilizing any real-time features. The use of KVM-RT on the host, along with a RedHawk guest shielding both network interrupts and iperf tasks, yields dramatic improvements in network performance and consistency.

Further reading:

https://wiki.libvirt.org/Virtio.html

Physical Networking

Physical network cards can be used to provide dedicated networking devices to virtual machines. Single Root IO Virtualization (SR-IOV) technology can make a physical network card appear as multiple, individual devices, called virtual functions, on the PCI bus. Alternatively, the entire network card can be dedicated to a single virtual machine via complete PCI passthrough.

SR-IOV

Single root I/O virtualization (SR-IOV) allows a single root function to appear as multiple, separate, physical devices. A single ethernet port on a network card could appear as multiple functions each with its own configuration space, bus address, and IOMMU group. Multiple functions appear as separate devices in the output of commands that list PCI device information.

NOTE

The KVM-RT qualified platform supports SR-IOV. Users of a non-qualified system should check with the corresponding manufacturer documentation on whether their system supports this feature and ensure it is enabled in the BIOS.

NOTE

The network card must also be SR-IOV capable. Refer to the manufacturer's specifications to verify SR-IOV capabilities.

SR-IOV enabled devices use two PCI functions, Physical Functions (PFs) and Virtual Functions(VFs).

Physical Functions

Physical Functions (PFs) are the complete PCIe device that contain the SR-IOV capabilities. They operate as normal PCI devices while also configuring and managing SR-IOV functionality.

Virtual Functions

Virtual Functions (VFs) are derived from a Physical Function. Virtual Functions are simple PCIe functions that process I/O. A single, physical ethernet port can map to many Virtual Functions up to the limitations of the device.

Unlike the virtual network methods mentioned in the previous section, some setup is required before a Virtual Function can be utilized by a guest. See Appendix B, SR-IOV Setup for help.

Once Virtual Functions are set up, proceed to pass through the PCIe device associated with the VF to the guest as one would with any other PCIe device.

Guests on the same host utilizing Virtual Functions on the same Physical Function benefit from transfer speeds greater than the limitation of the network since they do not get routed to the switch and back to the card.

Guests using a Virtual Function cannot communicate with the host's corresponding Physical Function. A separate interface would need to be configured to enable guest-to-host communication.

Further reading:

https://www.intel.com/content/www/us/en/developer/articles/technical/configure-sr-iov-network-virtual-functions-in-linux-kvm.html
https://access.redhat.com/documentation/enus/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/sect-pci_devices-pci_passthrough

SR-IOV vs MacVTap Performance

The similarities between Virtual Functions (SR-IOV) and MacVTap warrant a comparison in performance.

Two KVM-RT hosts with identical guests and 10Gb ethernet cards were setup to communicate with each other over a dedicated ethernet connection isolated from any other network.

The guests first communicated over MacVTap in bridge mode with a virtio device model using the first interface of the 10Gb network card.

After data was collected, Virtual Functions were configured for the same interface and the guests utilized PCI passthrough to use a Virtual Function.

During a round trip network test MacVTap was shown, on average, to use 34.6% more kernel CPU time than SR-IOV when transferring GB of data for each interval and 69.3% more kernel CPU time, on average, when transferring MB of data for each interval.

Along with lower CPU utilization, SR-IOV also provides advantages in throughput and scalability when compared to virtual networking methods.

Network PCI Passthrough

Complete passthrough of a network card is the best method to provide full functionality of a network card to a guest while also isolating its use from both the host and any other guests.

This method also allows the other interfaces (ports) of the card to be configured for guest use. This method is best for any use cases where networking performance is critical to a real-time application.

This method is the least scalable with respect to the maximum number of guests and network interfaces given the limitation of PCIe slots on the host motherboard.

Storage

Storage options include virtual disk formats like qcow2 and raw images, alongside physical disk configurations including the assignment of physical disk partitions and PCI passthrough of physical storage controllers.

Virtual Disks

Virtual machines see the virtual disk image file as a physical disk, however I/O is actually managed by the host system via the hypervisor (QEMU/KVM).

When the VM makes an I/O request to the virtual disk, the hypervisor intercepts the request and translates it in a manner appropriate to the disk format being used before forwarding the request to the host's file system.

The translation varies depending upon the virtual disk format in use.

With qcow2 images, there is some overhead as they involve the management of additional metadata, snapshots, and compression.

Raw images, however, have a one-to-one mapping of disk sectors. This allows the hypervisor to directly translate the request to the corresponding offset in the raw image file.

Once forwarded to the host's file system, the I/O request appears as a standard file operation. The complexity of the file operation will vary for qcow2 depending upon the structure of the image and the state of the virtual disk. The process for raw images is a straightforward read or write at a given offset.

The host kernel schedules the I/O operations and disk interrupts are generated as they normally would be for the host system.

Once completed, the hypervisor sends a result to the VM. The VM interprets this as though the I/O operation on its virtual disk has been completed.

Qcow2 Images

QEMU Copy-on-Write 2 (qcow2) images offer several features:

- Sparse allocation. The disk space on the host is allocated dynamically, allowing the disk to take up only the amount of space needed by the data stored within.
- Compression. Images can be compressed to reduce host disk usage.
- Snapshots. Allows the state of the VM to be saved and revert back to later.

As mentioned before, the benefits qcow2 offers also introduces some overhead which impacts performance.

Raw Images

Raw images are simple, unstructured disk images that occupy the entire space allocated when created. Because of this simplicity, raw images provide better read/write performance when compared to qcow2 images.

Virtual Disk Drawbacks

While virtual disk images are easier to migrate and require fewer resources when compared to allocating physical disks to VMs, there are drawbacks which affect VM and host disk performance.

Virtual disk images are placed on the host file system. This requires VM disk I/O operations to be communicated via the hypervisor and completed by the host kernel.

There is contention between scheduling I/O operations for the VM and I/O operations for the host. This can negatively impact read and write operations for both the VM and the host. Performance degradation is more apparent when managing multiple VMs; each requiring the host to schedule I/O operations on its behalf.

With virtual disk images, flushes for dirty pages to write to the physical host disk are dependent upon the host. Virtual disk write performance is reduced when the host decides to perform a sync.

While VMs with virtual disk images can benefit from fast cached read speeds, the host controls when to evict pages from memory. This has been shown to dramatically reduce read performance.

Further reading:

https://docs.redhat.com/en/documentation/red_hat_virtualization/4.3/html/technical_reference/qcow2
https://github.com/qemu/qemu/blob/master/docs/interop/qcow2.txt
https://www.qemu.org/docs/master/system/images.html
https://docs.kernel.org/admin-guide/mm/concepts.html#page-cache

Physical Disks

Physical hardware storage can be allocated to virtual machines through partition assignment and PCI passthrough.

Partition Assignment

When a physical disk partition on the host disk is assigned to the VM, I/O requests are directly mapped to the host disk partition and there is no longer a need for the hypervisor to translate the I/O request for a virtual disk image file format.

When the VM makes an I/O request, the hypervisor will intercept the request and directly pass the request to the assigned host disk partition.

As with virtual disk images, the I/O request is still handled by the host kernel as a standard file operation, however there are no intermediate files system layers.

Once complete, a disk interrupt is generated, signaling to the host that the operation is complete. The hypervisor then sends a signal to the VM.

Partition assignment offers many benefits over virtual disk images such as reduced complexity, overhead, and improved disk performance. There are drawbacks, however, when compared to PCI passthrough. Partition assignment still involves host kernel support to perform I/O operations and there is a notable performance impact due to contention between scheduling I/O operations for both VM and host. The effects of the host cache flushing and evicting pages from memory mentioned in the previous section still apply here to disk partition assignment.

Note that partition assignment is not considered passthrough. The distinction being that partition assignment still relies on the host kernel and disk driver to process I/O operations to/from the physical disk on behalf of the hypervisor intercepting signals from the VM.

Storage PCI Passthrough

PCI passthrough offers the best performance by reducing host kernel involvement in executing VM I/O operations.

On IOMMU enabled systems, the disk controller can be passed directly to the guest VM and controlled by the guest VM's disk driver. This operation is made possible due to the host's disk driver being unloaded and instead loading the VFIO driver to provide full direct, device access to the VM.

One of the drawbacks to PCI passthrough is complete resource allocation. Depending upon the topology of the host system, passthrough of a disk controller can prevent the host from having access to disks under that controller. This is more evident with SATA controllers where multiple disks might be part of the IOMMU group associated with the controller and passthrough allocates all of the disks to the VM. The performance advantages of PCI passthrough, which include allowing the VM direct control over the disk device and freeing the host from processing VM I/O operations, outweigh the cost of complete resource allocation.

Graphics

Virtual graphics implementations within virtual environments include VGA, QXL, and Virtio GPU. VNC and SPICE are display protocols used to deliver a virtual graphics display. Full GPU PCI passthrough provides full access to a physical graphics card on the host and provides the best graphical performance and reduced latencies.

VGA

VGA offers broad compatibility with older OS versions, especially those that might not have specific drivers for more advanced virtual graphics adapters such as QXL or Virtio GPU.

QEMU emulates a simple VGA card with Bochs VESA BIOS Extensions (VBE) with the guest OS using the bochs-drm driver.

The emulated VGA graphics adapter supports low-resolution graphics modes and can provide higher resolutions and color depths with VBE.

This device uses a simple framebuffer. The hypervisor (QEMU) reads this framebuffer and displays it on the screen or sends it to a remote client via a display protocol such as VNC.

Emulated VGA does not support hardware acceleration and all graphics rendering is handled by the CPU which can limit high-resolution displays or graphically intensive applications.

Overall, it has lower performance when compared with the other methods mentioned in this doc, but offers broader compatibility. For example, both QXL and VirtIO GPU have optimizations which offer some level of 2D acceleration even without the use of the host GPU, but require that the guest OS support and use paravirtual drivers for graphical devices.

QXL

QXL is a paravirtual graphical adapter with 2D acceleration support that utilizes the SPICE protocol (see SPICE below).

It is designed to provide better performance than traditional emulated graphics by leveraging the capabilities of the host machine and improving the efficiency of the graphics operations in VMs. The graphical adapter utilizes a paravirtual graphics 'qxl' driver within the VM and communicates with a QXL device on the backend via QEMU. The guest OS needs this qxl driver to translate the guest's graphical operations into commands that the QXL device can process efficiently.

QXL operates within the VM and handles drawing commands, screen updates, and other graphics-related tasks. Unlike emulated graphics (VGA), QXL is designed to work in virtual environments, allowing it to offload tasks to the host system and SPICE server. This makes it more efficient in terms of host CPU and memory usage. It also supports single monitor resolutions up to 3840x2160.

Virtio GPU

Virtio GPU is a modern paravirtualized graphics driver. Part of the broader Virtio framework, Virtio GPU provides efficient I/O virtualization by exposing simplified device interfaces to the guest OS. This reduces the overhead that is normally associated with fully emulated hardware. Virtio GPU provides better performance and flexibility over full device emulation and older paravirtual solutions such as QXL.

The guest OS will need support for the paravirtual drivers. While the driver is lightweight and optimized for virtual environments, it may not be supported by older operating systems. Full emulation would be necessary in this case.

Virtio GPU supports 2D acceleration. 3D acceleration support via OpenGL is not stable at this time. For more complete graphical capabilities, we recommend PCI passthrough.

Further reading:

https://www.kraxel.org/blog/2019/09/display-devices-in-qemu/

Display Protocols

The VNC (Virtual Network Computing) protocol and the Simple Protocol for Independent Computing Environment (SPICE) protocol are described and compared in this section.

VNC

VNC (Virtual Network Computing) is a cross-platform protocol that provides remote access to a GUI by transmitting screen updates, keyboard, and mouse events between a server running on the VM host and a client running on the user's system. It is based on the

Remote Framebuffer (RFB) protocol which is simple compared to more advanced protocols like SPICE.

VNC provides a broader compatibility than SPICE which is beneficial for older operating systems running within virtual machines, however it does not perform as well as SPICE.

Further reading:

https://web.mit.edu/cdsdev/src/howitworks.html

SPICE

The Simple Protocol for Independent Computing Environment (SPICE) protocol is designed to handle remote access to virtualized desktops, enabling high performance graphics and input redirection between VM and a client.

SPICE manages the guest framebuffer, efficiently streaming graphical updates to the client. It tracks changes to the framebuffer and only sends the updated regions to the client. It also caches frequently used graphical elements on the client, reducing the need to retransmit them.

Like VNC, SPICE operates as a client-server protocol with several components working together to deliver a remote desktop. The server runs on the host machine (within the hypervisor) and manages graphical, input, and multimedia data streams. The client will connect to the server and receive the guest VM's display while also handling input events from mouse, keyboard, etc.

SPICE also has a software component running inside the guest to provide additional features such as clipboard sharing, dynamic resolution changes, and more. Some features might not perform as well depending on the graphics adapter in use (QXL vs. Virtio GPU).

SPICE generally offers better performance, especially for multimedia heavy workloads and when interacting with the desktop. This is in part due to features like client-side caching, and advanced compression methods.

Further reading:

https://www.spice-space.org/documentation.html

Graphics PCI Passthrough

GPU passthrough provides full access of a physical graphics card on host to the virtual machine. IOMMU will provide secure memory isolation and access while VFIO will allow the graphics driver in the guest OS to directly access the device.

The virtual machine will then have access to all of the physical GPU's features as though it were a physical system. Depending on the GPU used, this includes multiple displays, 3D acceleration, CUDA, and more.

Note that with PCI passthrough, this will prevent the GPU from being used by the host and any other virtual machines. This method, while not necessarily scaling well, provides the best graphical performance and reduced latencies.

7
Analysis and Debugging

This chapter covers the system tools that can be used to analyze performance issues or debug system problems in virtualized environments.

A new multi-merge tracing feature is included in the latest release of the RedHawk operating system. It allows the merging of multiple system trace dumps into one view organized by timestamp. This new feature is crucial to debugging virtualized environments that often produce cross-VM and host interactions that can impact the performance of real-time applications.

In order to take advantage of the multi-merge tracing feature, all the guest VMs to be traced must be synchronized using the time of day clock (TOD). See the section “Instructions to run chrony” on page 5-1 to start-up chrony on each of the guest VMs to be traced.

NOTE

The time stamp counter (TSC) cannot be synchronized, therefore, only the TOD timestamp type should be used when tracing multiple systems. Be sure to select the TOD timestamp clock option in the trace tools.

In this chapter, the following information is presented:

- the KVM trace events supported in RedHawk.
- a brief description of the RedHawk tracing tools collectively known as **`xtrace`**. These tools use a simple command line interface. An example of tracing the host and one guest VM using xtrace and the new multi-merge feature is included.
- a new service named KVM-RT Guest Services. KVM-RT Guest Services is a collection of application programmer interfaces which give guest userspace applications access to functions exposed by the host hypervisor.

NightTrace is an optional product offered by Concurrent Real-Time. NightTrace is part of the NightStar family and consists of an interactive debugging and performance analysis tool, trace data collection daemons, and two Application Programming Interfaces (APIs) allowing user applications to log data values as well as analyze data collected from user or kernel.

For information on how to use NightTrace with KVM-RT see the "Kernel Tracing with KVM-RT" section in the NightTrace User's Guide.

KVM Trace Events

Following are the KVM traceable events supported by the RedHawk operating system.

`KVM_ENTER_VM_PID`

This is a generic catch-all event which will be triggered any time execution/control is transferred from the host kernel to the guest VM. It is produced by the KVM module on the host system, right before the host-guest transition.

`KVM_EXIT_VM_PID`

This is a generic catch-all event which will be triggered any time execution/control is transferred from guest VM to host kernel. It is produced by the KVM module on the host system, right after the guest-host transition.

`KVM_EXIT_HANDLER`

This event is logged to provide additional information about VM exits, such as when MSR reads and writes occur along with the MSR type and data that is read or written. This event is logged shortly after KVM_EXIT_VM_PID.

`KVM_GUEST_HC_START`

This event is logged by the guest VM right before it makes a hypercall to the host.

`KVM_GUEST_HC_END`

This event is logged by the guest VM right after control returns from a hypercall.

`KVM_HOST_HC_ENTER`

This event is logged by the host system right after execution reached the generic hypercall handler.

`KVM_HOST_HC_EXIT`

This event is logged by the host system right before execution exits the generic hypercall handler.

Kernel Tracing with xtrace

`xtrace` is a command line interface used in the tracing and analysis of dumps.

xtrace comes with the RedHawk Operating system in the **ccur-xtrace** package and contains several tools named **xtrace-**<function>. To see all the commands and libraries provided by this package, on a RedHawk system execute:

```
rpm -ql ccur-xtrace
```

The following are the tools directly called in the example that follows. A brief description and only a few options are mentioned below. For more information and to see more options, use the **--help** option:

xtrace-run:

captures xtrace data during the execution of a shell command. The command must be specified in the command line. When the command exits **xtrace-run** stops. The **-o** option specifies the output directory name where the xtrace data will be saved. The **-m** overwrite option may be used when the tracing will go for long periods of time and the xtrace data will grow very large.

xtrace-multi-merge:

merges into one multi-merge directory the xtrace-data directories specified in the command line. These are the directories created when **xtrace-run** was invoked. In the command line specify one directory for the host and one for each guest VM traced. The **-o** option lets you specify the directory name of the multi-merge directory to be created. The **-t** option sets the xtrace time-stamp clock to be used. Note that only the time of day clock (TOD) can be synchronized.

xtrace-view:

merges and displays xtrace data in a user-readable format. The xtrace data directory must be specified.

xtrace-ctl:

provides control of the kernel xtrace module on one or more CPUs. In the non-interactive mode, commands such as FLUSH, PAUSE, RESUME are specified in the command line.

Example: multi-merge Tracing with xtrace

This example captures a trace dump on the host system and a guest VM simultaneously, and then merges the two trace dumps into one. The example assumes that the user application is known to fail within the first five minutes.

NOTE

Time of day synchronization must be configured and running before guest VMs are traced. Refer to the section "Instructions to run chrony" on page 5-1 to start-up chrony on each of the VMs to be traced.

1. Trace host system in the background and sleep for a span of time greater than it takes the user application to fail:

   ```
   rm -rf xtrace-host
   xtrace-run -m overwrite -t tod -o xtrace-host \
       sleep 600 &
   ```

2. Start tracing remotely from the host. When the user application fails on the guest VM, the trace buffer is flushed:

   ```
   ssh guest_vm "rm -rf xtrace-vm;
   xtrace-run -m overwrite -t tod -o xtrace-vm \
       bash -c '( userapp || xtrace-ctl flush)'"
   ```

3. Flush the trace buffer and stop tracing on the host:

   ```
   xtrace-ctl flush stop
   ```

4. Copy the trace data directory from the guest VM to the host system:

   ```
   scp -r guest-vm:xtrace-vm .
   ```

5. Merge the guest and host trace directories into one:

   ```
   xtrace-multi-merge -o xtrace-merged xtrace-host xtrace-vm
   ```

6. View the merged trace arranged according to time stamp:

   ```
   xtrace-view xtrace-merged
   ```

The fields displayed are controlled by options to **xtrace-view**. The fields in the following example output are: timestamp (TOD), hostname, CPU and event.

Note that CPUs are local to each host so in the excerpt that follows, "vm1 0" denotes virtual CPU 0 in the guest VM whose hostname is "vm1".

```
23.404455270 host 3 INTERRUPT_ENTER [apic_timer]
23.404455720 host 3 HRTIMER_CANCEL [0xffffffff8e8f84e0]
23.404455898 host 3 HRTIMER_EXPIRE [0xffffffff8e8f84e0]
23.404456627 host 3 SCHED_WAKEUP [740216]
23.404456854 host 3 HRTIMER_EXPIRE_DONE[0xffffffff8e8f84e0]
23.404456971 host 3 HRTIMER_START [0xffffffff8e8f84e0]
23.407646071 vm1 0 SYSCALL_EXIT [openat]
23.407646321 vm1 0 SYSCALL_ENTER [read]
23.407646512 vm1 0 FILE_READ [3]
23.407647171 vm1 0 SYSCALL_EXIT [read]
```

KVM-RT Guest Services

Virtualized environments can produce complex cross-VM and host interactions which can have detrimental effects on the performance of hard real-time applications running on the VMs. Some of these interactions might be infrequent and/or hard to reproduce. In these cases the standard approach of tracing may not suffice.

KVM-RT Guest Services is a collection of application programmer interfaces which give guest userspace applications access to functions exposed by the host hypervisor.

One of the ways to reduce complexity is to leverage the implied domain knowledge contained within each of the applications. Applications know the state the application should be in at any particular time and when any timing or state violations occur. In that context, KVM-RT Guest Services gives the application developer the ability to:

1. log relevant events/data from an application running on a guest-VM directly to a central logging/tracing facility (i.e. syslog, NightTrace, xtrace) on the host.

2. flush xtrace buffers on the host. This can be combined with local flushing of xtrace buffers on the guest to flush both guest and host buffers at about the same time.

3. log explicit pre-defined sequence of events, in the context of the host's clock, to establish ordering of events; "bracketing". For example the following was logged by two different guest VMs on the host:

   On VM1:

   ```
   host: "VM1 is about to start A"
   ...
   host: "VM1 just finished A"
   ...
   ```

   On VM2:

   ```
   host: "VM2 is about to start B"
   ...
   host: "VM2 just finished B″
   ```

   On the host, you will be able to see the order of events in the context of the host's clock:

   ```
   host: "VM1 is about to start A"
   ...
   host: "VM2 is about to start B"
   ...
   host: "VM2 just finished B"
   ...
   host: "VM1 just finished A
   ```

The KVM-RT Guest Services functions provided in the command line interface **`kvmrt-gs`** and the library **`libccur_kvmrt_gs`** are briefly discussed in the sections that follow.

Also discussed below are the traceable KVM-RT Guest Services events and the kernel boot parameters that must be enabled in the host and guest VMs.

KVM-RT Guest Services Library Interface

The following functions are provided via the library **libccur_kvmrt_gs**. See the **libccur_kvmrt_gs(3)** man page for more information on options and usage.

Note that the man page can be invoked using the names of any of the functions listed below. For example: **man kvmrt_gs_available**.

```
bool kvmrt_gs_available(void);
bool kvmrt_gs_ping_available(void);
bool kvmrt_gs_log_msg_available(void);
bool kvmrt_gs_xtrace_flush_available(void);
bool kvmrt_gs_xtrace_log_data_available(void);
long kvmrt_gs_ping(unsigned long cookie);
long kvmrt_gs_log_msg(char * msg);
long kvmrt_gs_xtrace_flush(unsigned long scope);
long kvmrt_gs_xtrace_log_data(void * data, long size);
```

`kvmrt_gs_available`

Returns true if the KVMRT_GS interface is present, enabled, and permitted. Smilarly, kvmrt_gs_<function>_available returns true if each individual KVMRT_GS function is present, enabled, and permitted.

Note that availability of the interface does not imply availability of any function. Further, an invocation of a function which is available may still fail due to variety of reasons.

`kvmrt_gs_ping`

Ping a hypervisor with a *cookie*. The purpose of this function is to provide a guest with a simple light-weight mechanism with no copying or allocation to explicitly cause a VMEXIT event on a hypervisor in a way which can easily be traced and matched from guest and host sides. This interface produces corresponding xtrace events, when xtrace is available.

`kvmrt_gs_log_msg`

Log a short ASCII text message via the standard kernel logging mechanism on a hypervisor side. *msg* is a pointer to a standard Zero-terminated C string. The hypervisor and any of the intermediate layers may restrict the maximum length of the string, and/or truncate the message. See also kvmrt_gs_xtrace_log_data below.

`kvmrt_gs_xtrace_flush`

Trigger FLUSH xtrace event on host OS.

scope controls which CPUs are affected by the FLUSH:

```
KVMRT_GS_XTRACE_CPU_CURRENT
KVMRT_GS_XTRACE_CPU_VM
KVMRT_GS_XTRACE_CPU_ALL
```

issue FLUSH to the current CPU, all the CPUs servicing current VM, and all the CPUs active on the host system respectively.

`kvmrt_gs_xtrace_log_data`

Log arbitrary binary *`data`* buffer containing *`size`* bytes as two matching xtrace events on guest and host sides. The hypervisor and any of the intermediate layers may restrict the maximum size of, and/or truncate the data logged. See also kvmrt_gs_log_msg above.

KVM-RT Guest Services Command Line Interface

The following commands are provided via the **`kvmrt-gs`** command line interface. See the **`kvmrt-gs(1)`** man page for more information on options and usage.

`kvmrt-gs [OPTIONS] [COMMAND [ARGUMENTS] ...] ...`

`available`

Return SUCCESS if KVM-RT Guest Services are available.

`ping_available`

Return SUCCESS if 'ping' command is available.

`ping` *`COOKIE`*

ping a hypervisor with a *`COOKIE`* - an arbitrary user-selected integer (unsigned long int).

`log_msg_available`

Return SUCCESS if the 'log_msg' command is available.

`log_msg` *`MESSAGE`*

Log a message on hypervisor. *`MESSAGE`* can be either a regular quoted ASCII string or a hex-encoded byte sequence.

`xtrace_flush_available`

Return SUCCESS if the 'xtrace_flush' command is available.

`xtrace_flush` *`SCOPE`*

Flush xtrace buffers on host OS. *`SCOPE`* can be one of the following: {0: the current CPU; 1: all the VM CPUs; 2: all host CPUs}

`xtrace_log_data_available`

Return SUCCESS if the 'xtrace_log_data' command is available.

`xtrace_log_data` *`DATA`*

Log xtrace event with binary data. *`DATA`* can be either a regular quoted ASCII string or a hex-encoded byte sequence.

KVM-RT Guest Services Trace Events

KVM-RT Guest Services logs various trace events. Every event type comes as a pair, where the *_GUEST part is logged on the guest side and the *_HOST is logged on the host.

The purpose behind such double-logging is to provide predictable reference points within the trace logs for cases where the host and the guest VM clocks might not be synchronized or have drifted in relation to each other.

`KVMRT_GS_PING_GUEST`
`KVMRT_GS_PING_HOST`

> These are produced by the "ping" function of KVM-RT Guest Services. See **`kvmrt_gs_ping(3)`** for details.

`KVMRT_GS_FLUSH_GUEST`
`KVMRT_GS_FLUSH_HOST`

> These are produced by the "xtrace_flush" function of KVM-RT Guest Services. See **`kvmrt_gs_xtrace_flush(3)`** for details.

`KVMRT_GS_LOG_DATA_GUEST`
`KVMRT_GS_LOG_DATA_HOST`

> These are produced by the "xtrace_log_data" function of KVM-RT Guest Services and is similar to `XTRACE_EV_CUSTOM`. See **`kvmrt_gs_xtrace_log_data(3)`** for details.

KVM-RT Guest Services Kernel Boot Parameters

KVM-RT Guest Services requires that the following kernel parameters must be enabled at boot time. Note that one is specific to the host system and the others to the guest VMs.

`kvm.kvmrt_gs_hc_host_enabled=`

> [KVM,x86] Enable KVM-RT Guest Services Hypercall on KVM Host. Setting this to 1 (Enabled) permits host to advertise KVMRT_GS hypercall and related GS functions to the guests. This is a host-side parameter for KVM module. Default is 0 (Disabled).

`kvmrt_gs_hc_guest_enabled=`

> [KVM_GUEST,x86] Enable KVM-RT Guest Services Hypercall on KVM Guest. Setting this option to 1 (Enabled) permits guest kernel to discover and use KVMRT_GS Hypercall and its functions if such is offered by the Host. This is a guest-side kernel parameter. Default is 0 (Disabled).

`kvmrt_gs_syscall_enabled=`

> [KVM_GUEST,x86] Enable KVM-RT Guest Services Syscall on KVM Guest. Setting this option to 1 (Enable) permits guest kernel to advertise

KVMRT_GS syscall and its functions to the userspace applications running on the Guest. This is a guest-side kernel parameter. Default is 0 (Disabled).

A
NUMA mapping the Supermicro M12SWA-TF

This appendix describes the NUMA node mapping of device slots and I/O ports for the Supermicro M12SWA-TF platform.

The KVM-RT Product features the Supermicro M12SWA-TF motherboard. This board supports both the AMD Ryzen Threadripper PRO 5975WX and 5965WX. Both systems have been pre-qualified for the KVM-RT product.

For more information on the KVM-RT product, see the section “The KVM-RT Product” in the KVM-RT Release Notes.

NOTE

The board described in this document has the latest BIOS revision provided by Supermicro as of this release, Supermicro M12SWA-TF BIOS Revision 2.1. The BIOS must have this revision or be updated in order to use these mappings.

Importance of NUMA Mappings in KVM-RT

Leveraging the topology of the system will lead to a more efficient KVM-RT configuration. It will reduce the need to reassign device interrupts to different NUMA nodes and will decrease latency for the real-time VMs.

NUMA node mapping is important in considering the placement of devices for host, real-time and non real-time guest VMs. Placing devices on the same NUMA node as the real-time VM ensures faster access to data and lower latencies when those devices are used. On the other hand, devices not used by the real-time VM but allocated to the same NUMA node as the real-time VM, increase the risk of interference and an increase in latency for the real-time VM.

The first step in achieving an optimal configuration is selecting the best NUMA node for the real-time VM by first examining the devices the VM will use, then selecting a NUMA node where the devices to be used by the VM can be placed. For example, a real-time VM that requires 2 PCIe slots, can be placed in NUMA node 1 or NUMA node 2; while one that requires 1 PCIe slot and 1 USB port would be best placed in NUMA node 3. Refer to diagrams in section below.

Note that compromises may have to be made since device slots and ports are not divided evenly between all NUMA nodes. In addition, there is a preference for NUMA node 0 which is *not* an ideal choice for real-time VMs. If more PCIe slots and more devices are needed beyond those mapped to the NUMA node chosen for the real-time VM, then the IRQs currently handled by another NUMA node will need to be reassigned to the NUMA node chosen for the real-time VM.

NUMA Node Mappings of devices and I/O ports

In an effort to optimize KVM-RT hardware configurations, the following diagrams are provided that map device slots and I/O ports to their respective NUMA nodes.

NOTE

These mapping are only relevant to the Supermicro M12SWA-TF running BIOS revision 2.1.

The bus addresses of on-board devices (e.g., NVME, SATA) and ports (e.g., USB) have *not* been included in the diagrams. This is because those addresses vary depending on which devices or ports are in use.

The PCIe diagram below contains a color-coded mapping of device slots to NUMA nodes. Note the bus addresses displayed next to each PCIe slot are persistent across reboots. This makes them easy to identify when selecting devices for PCIe passthrough to VMs.

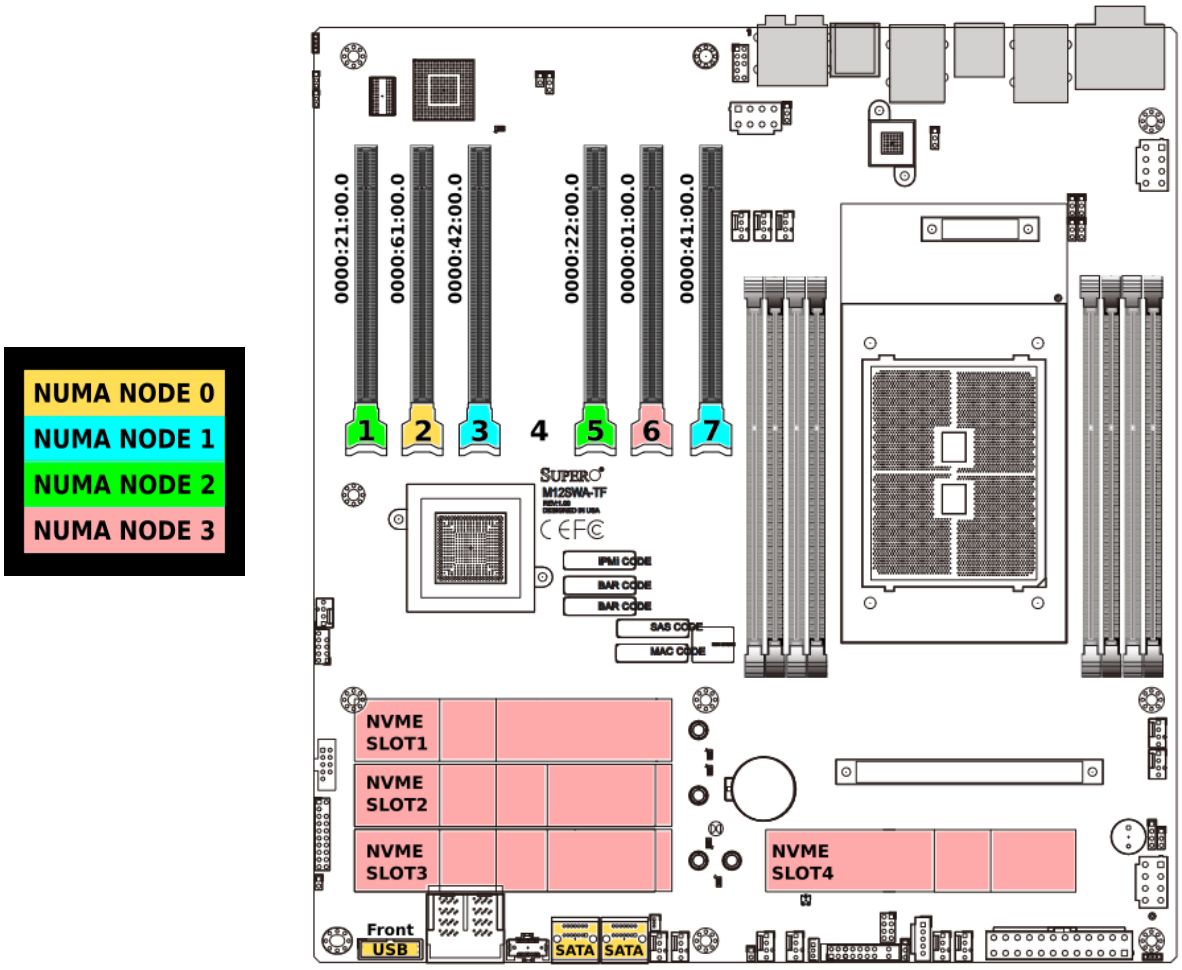
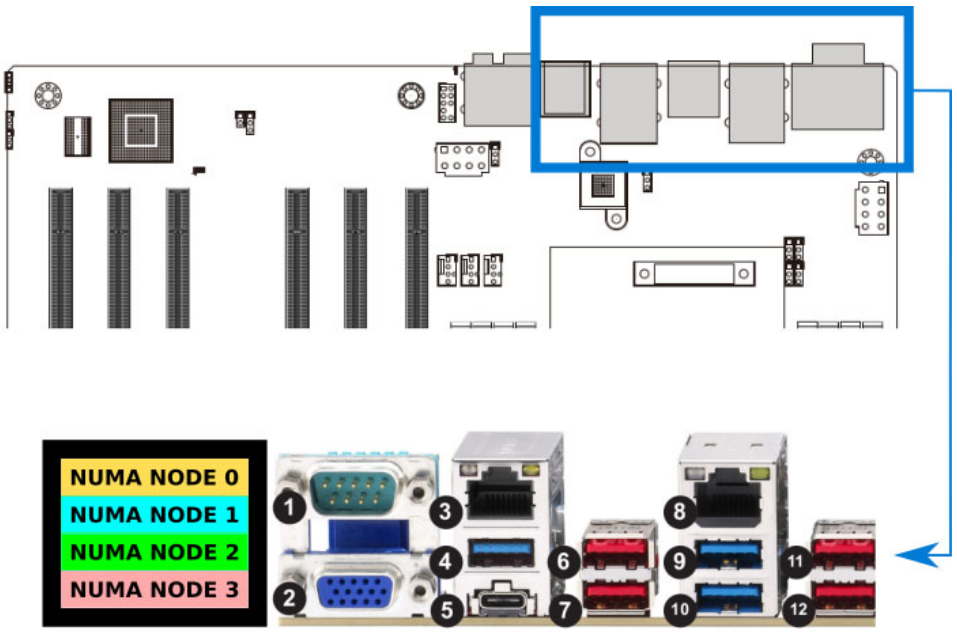


Figure A-1 NUMA node mapping of devices in the Supermicro M12SWA-TF

The rear panel diagram below contains a table with NUMA node color-coded entries. The numbers correspond to the port numbers (in black circles) indicated in the diagram. This table also contains details about each port which can prove helpful when determining which ports or device bridges will be passed through to a VM.



	Rear Panel I/O Ports	NUMA Node
1	COM1	0
2	VGA Port	0
3	1Gb LAN Port (i210)	0
4	USB3.2 Gen1 Type A, 5Gb/s	2
5	USB3.2 Gen2x2 Type C, 20Gb/s	0
6	USB3.2 Gen2 Type A, 10Gb/s	3
7	USB3.2 Gen2 Type A, 10Gb/s	3
8	10Gb LAN port (AQC113C)	0
9	USB3.2 Gen1 Type A, 5Gb/s	0
10	USB3.2 Gen1 Type A, 5Gb/s	0
11	USB3.2 Gen2 Type A, 10Gb/s	0
12	USB3.2 Gen2 Type A, 10Gb/s	0

Figure A-2 NUMA node mapping of I/O Ports in the Supermicro M12SWA-TF

B
SR-IOV Setup

This section covers SR-IOV setup. The example in this section utilizes an Intel Ethernet Converged Network Adapter X550-T2.

Some setup is required before a Virtual Function can be utilized by a guest. Once an SR-IOV capable card is installed on the host, begin SR-IOV setup as

1. Verify that the SR-IOV capable card has been installed on the host and identify the card of interest with the **pci** command output:

   # pci | grep -i ether

```
0000:21:00.0 Ethernet: Intel Corporation Ethernet Controller 10G X550T
0000:21:00.1 Ethernet: Intel Corporation Ethernet Controller 10G X550T
```

2. Verify the device has SR-IOV capabilities as well as the total number of VFs available and number of VFs in use.

   # lspci -vvv -s 0000:21:00.0

```
[...]
Capabilities: [160 v1] Single Root I/O Virtualization (SR-IOV)
  IOVCap: Migration-, Interrupt Message Number: 000
  IOVCtl: Enable+ Migration- Interrupt- MSE+ ARIHierarchy+
  IOVSta: Migration-
  Initial VFs: 64, Total VFs: 64, Number of VFs: 0, Function Dependency Link: 00
  VF offset: 128, stride: 2, Device ID: 1565
  Supported Page Size: 00000553, System Page Size: 00000001
  Region 0: Memory at 00000000b0d00000 (64-bit, non-prefetchable)
  Region 3: Memory at 00000000b0c00000 (64-bit, non-prefetchable)
  VF Migration: offset: 00000000, BIR: 0
```

   Note that this device is capable of configuring up to 64 Virtual Functions and currently has 0 configured.

3. Create VFs while the system is running. Determine the interface associated with the Physical Function:

   # lshw -class net -businfo

```
Bus info          Device     Class          Description
=======================================================
pci@0000:21:00.0  enp33s0f0 network      Ethernet Controller 10G X550T
pci@0000:21:00.1  enp33s0f1 network      Ethernet Controller 10G X550TG
```

   In this example we are interested in the first port of the X550T card with PCI address '0000:21:00.0' and interface 'enp33s0f0'.

Update the number of Virtual Functions to be configured for the desired interface. For this example:

```
# echo 2 > /sys/class/net/enp33s0f0/device/sriov_numvfs
```

View the configured VFs:

```
# lshw -class net -businfo
```

```
root@thor:~# lshw -class net -businfo
Bus info          Device         Class         Description
=========================================================
pci@0000:21:00.0  enp33s0f0      network       Ethernet Controller 10G X550T
pci@0000:21:00.1  enp33s0f1      network       Ethernet Controller 10G X550T
pci@0000:21:10.0  enp33s0f0v0    network       X550 Virtual Function
pci@0000:21:10.2  enp33s0f0v1    network       X550 Virtual Function
```

Two VFs (enp33s0f0v0 and enp33s0f0v1) were created for the interface specified above.

It is important to note that VFs are separated into their own IOMMU group, thus making PCI passthrough of individual VFs sharing the same PF to different guests possible.

```
# pci -G
```

```
[...]
    IOMMU Group 45:
       0000:21:00.0 Ethernet: Intel Corporation Ethernet Controller 10G X550T
    IOMMU Group 46:
       0000:21:00.1 Ethernet: Intel Corporation Ethernet Controller 10G X550T
    IOMMU Group 56:
       0000:21:10.0 Ethernet: Intel Corporation X550 Virtual Function
    IOMMU Group 57:
       0000:21:10.2 Ethernet: Intel Corporation X550 Virtual Function
```

Note also how the Virtual Function will utilize a different driver.

```
# pci -v 0000:21:00.0
```

```
0000:21:00.0 Ethernet: Intel Corporation Ethernet Controller 10G X550T
                 (numa=2, iommu=45, driver=ixgbe, irq=48, msi_irqs=265-269)
                NIC "enp33s0f0" (ec:e7:a7:07:79:44)
```

```
# pci -v 0000:21:10.0
```

```
0000:21:10.0 Ethernet: Intel Corporation X550 Virtual Function
                 (numa=2, iommu=56, driver=ixgbevf, irq=0, msi_irqs=270-272)
                NIC "enp33s0f0v0" (76:da:1f:8f:a3:c5)
```

4. Make Virtual Functions persistent across reboot. There are two methods to ensure Virtual Functions are configured when the system boots.

a. Boot line parameter.

   Identify the Physical Function's driver as shown above and add a boot line parameter with the maximum number of Virtual Functions to configure per interface for each network card utilizing that driver. For this example:

   ixgbe.max_vfs=2

   Add the boot option to the end of GRUB_CMDLINE_LINUX in **/etc/default/grub** and update the grub file using the appropriate method for your base distribution.

   Ubuntu systems:

   # update-grub

   RHEL-compatible systems:

   # grub2-mkconfig -o /boot/efi/EFI/redhat/grub.cfg

b. Modprobe config file

   Create a conf file in **/etc/modprobe.d** using the name of the Physical Function's driver and add an option with the maximum number of Virtual Functions to configure per interface for each network card utilizing that driver.

   For this example:

   echo 'options ixgbe max_vfs=2' >> **/etc/modprobe.d/ixgbe.conf**

5. Once Virtual Functions are set up, proceed to pass through the PCIe device associated with the VF(s) to the guest, as one would with any other PCIe device.

C
PCI Passthrough Boot Parameters

The parameters listed in this section may be added to the RedHawk system's boot time parameters. Use the **`blscfg(1)`** command to add and remove kernel boot parameters and then reboot the system for the changes to take effect.

Required in RedHawk Release 8.x

On systems running RedHawk release 8.x, you will need to set:

`intel_iommu=on`

Intel-based systems running RedHawk 8.x must enable *intel_iommu=on* in the kernel to enable passthrough.

Optional for enhancing host performance

To enhance host performance, you may also enable:

`iommu=pt`

This option only enables IOMMU for devices used in passthrough and will provide better host performance.

NOTE

This option is not supported on all hardware. If passthrough fails, remove this option.

Required for Graphic Cards

Most PCI-e (PCI Express) cards can be used for PCI passthrough without requiring any additional kernel boot parameter to be set. Graphic cards are the exception. The graphics card must be claimed by the VFIO driver early in the boot before it is claimed by the host's drivers.

The following boot parameters may be used to passthrough a graphics card. Note that **`lspci -nnk`** command can be used to obtain the necessary information about the device.

The PCI vendor and device id codes are listed at the end of the line in brackets ([]). The BUS:SLOT.FUNCTION information is listed at the very beginning. If more than one device is listed for your graphics card, you must include all the devices.

You can use one of the two following boot parameters to specify the device(s). While the first boot parameter is the simplest to use, the second one is needed when there are multiple cards on the system with the same vendor and device IDs.

1. *vfio-pci.ids* =[vendor:device,...]

   This parameter can be set to a comma-separated list of pci devices that will be assigned to the VFIO driver. Each device is specified by vendor:device.

NOTE

If you use this boot parameter and there are multiple cards with the same vendor and device IDS on the system, the host will not be able to properly initialize and use any of the devices.

2. *vfio-pci.addrs*=[BUS:SLOT.FUNCTION,…]

   This parameter is set to a comma-separated list of pci devices that will be assigned to the VFIO driver. Each device is specified by BUS:SLOT.FUNCTION.

   This boot parameter should be used when you have multiple cards on the system with the same vendor and device IDs and the host wants to be able to use one or more of the cards.

NOTE

If there are changes to the physical placement of cards in the system, the BUS:SLOT.FUNCTION setting will need to be re-evaluated as it may have changed. If it has changed, then the kernel boot parameter settings will have to be updated and the system rebooted.