www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 6 Issue 4 April 2017, Page No. 20931-20937
Index Copernicus value (2015): 58.10 DOI: 10.18535/ijecs/v6i4.24

Improving performance of Virtual Machines by Virtio bridge Bypass for PCI devices

[1]Shirley Kotian, [2]Kirti Menon, [3]Kirti Menon, [4]Utsav Mundada, [5]Neeraj Vilas Auti
[1234]PICT,
[5]PICT [CS]

ABSTRACT

Inspired by the Virtio module of virtualization, we propose an alternate method to directly communicate with PCI devices such as NIC without the use of any kernel modules. This method uses a specialized module written by us which will avoid the mechanism of bridges like the ones used in Virtio that increase latency. This module will be present in the userspace of the guest OS and we are specifically targeting the e1000 device for this purpose and later plan to make it generic for all PCI devices. Our motivation is to avoid unnecessary communication with the kernel which slows down the system. For the first step. we do resource mapping to map the PCI device memory into userspace. Then, we expose PCI configuration space through a userspace module using ACPI cables. Thus, we create a userspace PCI driver which will decrease the latency in access time and increase speed of execution. The applications in the Guest OS that request communication with the PCI devices will be redirected to our application. This will take some load off the kernel and reduce its overhead. Finally, we boot a VM that actually talks to our PCI device emulator.

GENERAL TERMS Virtio, UPCI, e1000, QEMU, virt-manager, UIO.

KEYWORDS : Emulation, Linux, QEMU

I. INTRODUCTION

Virtio is a virtualization standard for network and disk device drivers. In a paravirtualized hypervisor, it is an abstraction for a set of common emulated devices. The hypervisor can then export a set of commonly emulated devices and use a common application programming interface (API) to make them available. Latency inducing bridges are used which also increase access time for execution.

Eight different virtualization systems are supported by the Linux Kernel at the moment: IBM's System p, VMware's VMI, KVM, Xen, IBM's System z, lguest, User Mode Linux and IBMs legacy iSeries. Each of these had it's own network, console, block and other drivers with various optimizations and features until recently; and it seems likely that more such systems will appear. This is addressed by Virtio which is a series of Linux drivers which provide efficiency and can be adapted for various implementations of the

hypervisor by using a shim layer. There is an extensible feature mechanism for each driver which simply provides a least resistance path for the new hypervisors: and by providing supoort to this efficient mechanism for transport will reduce the amount of work that has to be done.

The Quick Emulator(QEMU) is an open source machine emulator as well as virtualizer and is generic in nature. If it is used as an emulator for a machine, it can run operating systems and programs made for a particular machine on a different machine by using binary dynamic translation. Thus, this mechanism renders it very efficient.If QEMU is used as a virtualizer, it achieves near native performance by executing the guest code directly on the host CPU. QEMU can virtualize server and embedded PowerPC, x86, s390, 64-bit POWER, 32-bit and 64-bit ARM, and MIPS guests and when running under two hypervisors: Xen amd KVM kernel module, it supports virtualization. We will be using the System emulation operating mode. In this particular mode, QEMU can boost various guest operating systems: Linux, Microsoft Windows, Solaris, BSD and DOS and it emulates a full computer system which includes all the peripherals. Virtual hosting is provided for several virtual computers on an individual computer. It supports emulation of many instruction sets, including MIPs, x86,ARMv8, 32-bit ARMv7, SPARC, PowerPC, MicroBlaze and ETRAX CRIS.

Inspired by this Virtio module of virtualization, we propose an alternate method to directly communicate with PCI devices such as NIC without the use of any kernel modules. This method uses a specialized module written by us which will avoid the mechanism of bridges like the ones used in Virtio that increase latency. This module will be present in the userspace of the guest OS and we are specifically targeting the e1000 device for this purpose and later plan to make it generic for all PCI devices.

II. MOTIVATION

We have found a few issues with the virtio module used currently:

1. For the KVM and QEMU guests using virtio networking, the networking breaks after a while. Continuous flow of networking constructs can be made possible by addressing this issue using direct memory mapping.

2. Virtio bridges are slow Ref:https://www.redhat.com/archives/libvirt-users/2012-June/msg00041.html

3. And, we found that while testing the throughput and latency of the network using Fedora 17 for both guest and host, using kernel 3.5.23.fc17.86 64. Pinging an external server on the LAN from the host(as shown in Fig. 1), using a gigabit interface, the results showed less RTT time. This can be seen from the following diagrams:

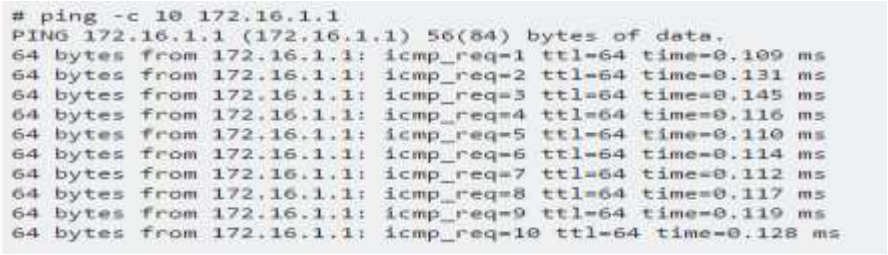

Fig. 1

Pinging the same external host on the LAN from the guest, the latency seems to be much higher(as

shown in Fig. 2).

```
# ping -c 10 172.16.1.1
PING 172.16.1.1 (172.16.1.1) 56(84) bytes of data.
64 bytes from 172.16.1.1: icmp_req=1 ttl=64 time=0.206 ms
64 bytes from 172.16.1.1: icmp_req=2 ttl=64 time=0.352 ms
64 bytes from 172.16.1.1: icmp_req=3 ttl=64 time=0.518 ms
64 bytes from 172.16.1.1: icmp_req=4 ttl=64 time=0.351 ms
64 bytes from 172.16.1.1: icmp_req=5 ttl=64 time=0.543 ms
64 bytes from 172.16.1.1: icmp_req=6 ttl=64 time=0.387 ms
64 bytes from 172.16.1.1: icmp_req=7 ttl=64 time=0.348 ms
64 bytes from 172.16.1.1: icmp_req=8 ttl=64 time=0.364 ms
64 bytes from 172.16.1.1: icmp_req=9 ttl=64 time=0.345 ms
64 bytes from 172.16.1.1: icmp_req=10 ttl=64 time=0.334 ms

--- 172.16.1.1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 8999ms
rtt min/avg/max/mdev = 0.206/0.374/0.543/0.093 ms
```

Fig. 2

III. ARCHITECTURE

As of now, if an application in the guest OS wants to access a PCI device, then it sends out a request to the hypervisor(in this case QEMU). The QEMU with the help of vhost driver in virtio emulates the device needed. This vhost model makes use of many virtio bridges, virtio queues, PCI bus and structures like Vring and VIRT QUEUE to perform this emulation. This results in high latency and overhead on the host kernel. For multiple VMs the performance of the host machine will be significantly compromised.

A. PROPOSED ARCHITECTURE

We propose a new model to go about this problem. We write our own module(UIO DRIVER) which is placed in the host userspace. This module will map the device memory into userspace allowing the applicaiton to directly access the device. The mapping of device memory allows us to directly make changes in the device memory. This technique is a lot faster than accessing the PCI address space to handle the device. We modify QEMU to redirect all virtio calls towards our UIO DRIVER. The UIO DRIVER also has PCI code implementaion to access the device via device driver. Now, UIO DRIVER gives us an advantage of DMA and speeds up the processing. This driver will also have all the respective functionalities to handle various calls targeted towards the device.

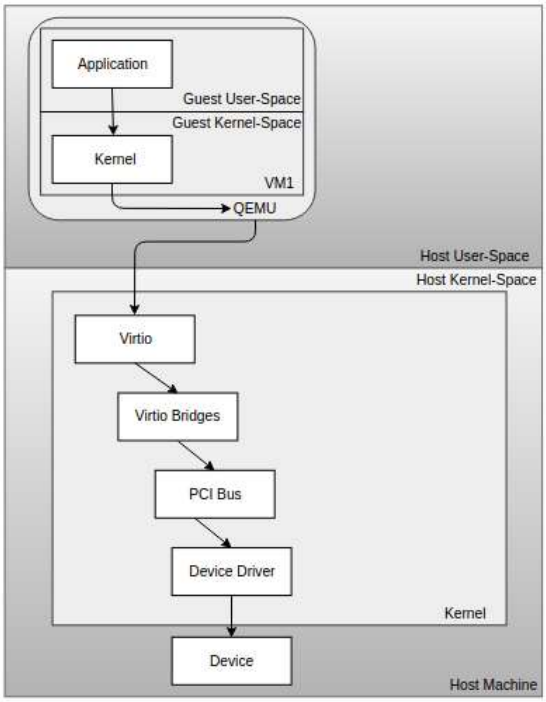

Fig. 3: Current Architecture

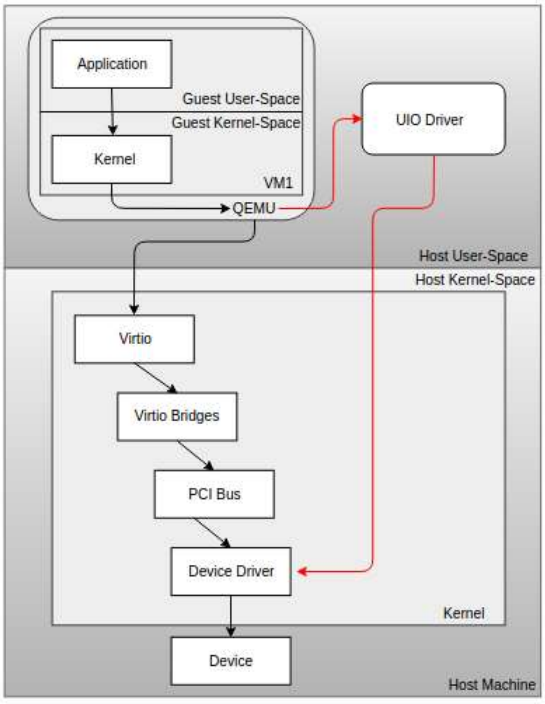

Fig. 4: Proposed Architecture

IV. IMPLEMENTATION

A. MEMORY MAPPING

Mapping into PCI memory region in the userspace :-

1. Run the lspci command and check if the device id which is of the format 0x:00.0(We tried specific to the network controller for the E1000 device)

2. check the given device on sysfs : ls -l /sys/bus/pci/devices/0000\:00 1c.4/0000\:09\:00.0/ starting from /sys/bus/pci/devices/

3. After compiling the code we can run it as ./a.out /sys/bus/pci/devices/0000\:00\:1c.4/0000\:09\:00.0/resource0 0x100 w 0x00 which is explained as:-

resource0 is the file which should be mapped for pci devices, 0x100 is the

offset‘ The command parameters are:-

./a.out sys file offset [type [data]]

sys file: sysfs file for the pci resource to act on

offset : offset into pci memory region to be acted

upon type : the type of access operation which

are:

b: byte

h: halfword

w: word

data: data to be written

== mmap() ==

The sysfs resource can be used along with mmap() to map the PCI memory into a userspace applications memory space. The application can then read/write values directly by placing a pointer to the start of the PCI memory region. Some more operations are associated with the memory pointers but are executed by the kernel.

```
fd = open$"/$sys/devices/pci0001\:00/0001\:00\:07.0/resource0",
O_RDWR); ptr = mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
0);
printf("PCI BAR0 0x0000 = 0x%4x\n",  *((unsigned short *) ptr);
```

B. TRACING

Tracing E1000 system calls:

QEMU’s tracing structure and how to use it for profiling, debugging and observing execution is explained as follows:-

1. first, building with the ’simple’ trace backend:

./configure –enable-trace-backend=simple make

2. Creating one file with all the events that you wish to trace. Eg: readv, writev

3. A trace file is produced by running the virtual machine:

qemu -trace events=/tmp/events ... # your normal QEMU invocation

4. Finally, pretty-print the binary trace file:

./simpletrace.py trace-events trace-*

Using the software emulation(”TCG”), when the QEMU executes a guest, it translates many blocks of guest code into native code and then executes the code.

Initially, QEMU needs to be compiled from source. Before that, docs/tracing.txt needs to be understood thoroughly and also edit the trace-events and remove a particular keyword-’disable’- from the following lines in that file.

The next step is to add these trace events into the /tmp/events file. It is also useful to put the entire qemu command line into a script alongwith the -trace events=/tmp/events parameter in case we have to rerun the trace.

Using the scripts/simpletrace.py script one can then analyze the log as it is describe in the tracing documentation of the QEMU. The output will be huge. The output can be aligned to make it easier to read:

./scripts/simpletrace.py trace-events trace-4491 — head

V. PRACTICAL APPLICATION

Our module will be integrated in the PCI codes. Thus, presumably decreasing latency in access time of commands and kernel overhead.

VI. SUMMARY AND CONCLUSION

Thus, we will be able to map the PCI memory in userspace, expose the PCI configuration space through our module and create a userspace PCI driver. Finally, we boot a VM that actually talks to our PCI device emulator.

VII. REFERENCES

[1] Jiuxing Liu, Wei Huang, Bulent Abali, Dhabaleswar K. Panda et al., High Performance VMM-Bypass I/O in Virtual Machines

[2] J. Liu, J.Wu, S. P. Kini, P.Wyckoff, and D. K. Panda et. al. High Performance RDMA-Based MPI Implementation over InfiniBand

[3] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd et.al. The Virtual Interface Architecture., IEEE Micro, pages 66-76, March/April 1998.

[4] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer Xen and the Art of Virtualization, October 2003.

[5]P. M. Chen and B. D. Noble et. al. When virtual is better than real, 2001

[6]K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. et. al. Safe hardware access with the xen virtual machine monitor, OASIS ASPLOS Workshop, 2004

[7]I. Pratt. Et. al.Xen Virtualization, Linux World 2005 Virtualization BOF Presentation

[8]M. Rosenblum and T. Garfinkel. Et. al. Virtual Machine Monitors: Current Technology and Future Trends, May 2005.

[9]J. Sugerman, G. Venkitachalam and B. H. Lim. et.al. Virtualizing I/O Devices on VMware Workstations Hosted Virtual Machine Monitor, 2001

[10]C. Waldspurger et. al.Memory resource management in vmware esx server, In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, 2002.