Network Function Virtualization

Packet Processing Performance of Virtualized Platforms with Linux* and Intel® Architecture®

Intel® Xeon® processor E5-2600 series

Intel® 82599 10Gb Ethernet Controllers

Audience and Purpose

The primary audiences for this reference architecture (RA) are architects and engineers evaluating Intel® architecture based solutions for cloud based network services where packet processing performance is a key requirement.

Process affinity and IRQ affinity strategies can have a dramatic impact on packet handling performance including the variability of throughput. This document provides an analysis and of underlying Linux kernel tasks for KVM bridged networks. A generalized affinitization rule set is proposed for a multi-VM environment with Linux KVM.

Details for configuring and testing a virtualized packet processing environment are provided. The platform is Linux* KVM on an ATCA Reference Platform with Intel® Xeon® Processor E5-2600 and Intel® 82599 10 Gigabit Ethernet Controller. Performance results only apply to the target platform; however the test methodology and optimization learnings are generally applicable to packet handling solutions using Linux* KVM virtualization and Intel 82599 10 Gigabit Ethernet Controllers.

October 2013

Table of Contents

2 Executive Summary

Virtualized standard high volume (COTS) servers are widely used in cloud data centers. Cloud computing technologies such as virtualization will also transform Telco networks to become flexible and agile for delivering "virtualized network services" over mobile and fixed-line networks. Requirements, architectures, and performance criteria of virtualized network solutions are being defined by industry bodies such at the ETSI Network Function Virtualization (NFV) Industry Specifications Group (http://portal.etsi.org/portal/server.pt/community/NFV/367).

Evaluating performance of virtualized packet processing on COTS platforms and understanding system configuration options is fundamental to developing solutions that will meet industry requirements and transform Telco networks.

This report contains packet processing performance data for the target platform. The data and learnings are intended to help evaluate an Intel® Architecture based COTS ATCA platform with specific software configurations. The solution presented is not an optimized solution or attempt to achieve any specific performance objectives. The reader is also cautioned that the target platform is a low-power embedded ATCA reference platform and does not represent performance of higher power rack-mount server platforms that are currently available with faster processors and more system resources.

The ATCA hardware platform tested is available from ADI Engineering. Software components used include Wind River Linux and the Intel® Data Plane Development Kit (Intel® DPDK). These components are being integrated into third party solutions.

Use Cases

1. A network service running on a host or in a virtual machine (VM) consumes IP packets from the external network (terminates packets).
2. A network service running on a host or in a VM produces IP packets sent to the external network (produces packets).
3. A network service running on a host or in a VM forwards IP packets (forwards packets).
4. A network service running on a host or in a VM consumes IP packets from the external network and send replies to the external network (packet transactions).
5. VM-to-VM communication on the same system.

System under Test - Hardware

The target platform was a Next Generation Communications Platform from Intel (codename Crystal Forest). Linux* KVM was used on the ATCA Reference Platform with Intel® Xeon® Processor Series E5-2600 (formerly codename Sandy Bridge) and Intel® 82599 10 Gigabit Ethernet Controller (formerly Niantic).

Intel® Ethernet Flow Director and Receive Side Scaling (RSS) are two different features of Intel® 82599 10 GE Controller with advanced on-board network controller card features. These features can greatly improve the system level performance if configured appropriately based on traffic and application characteristics.

System under Test - Software

The OS used was Wind River Linux 4.3.0.0 with kernel 2.6.34. The platform configuration includes:

- **OS / IP Stack**: Wind River Linux (native stack), CentOS* (guest OS), Intel DPDK
- **Hypervisor / virtual interface**: KVM - E1000 emulation, VirtIO, Vhost, SR-IOV

Intel Data Plane Development Kit (Intel DPDK) is a software package which helps fully utilize the packet parsing and security features of IA to shorten time-to-market of high performance packet processing solutions.

KVM - E1000 interface emulates a physical device. This allows any operating system function to be virtualized however this emulation is very inefficient.

Virtio is a Linux standard for network and disk drivers where the guest's device driver knows it is running in a virtual environment, and cooperates with the hypervisor.

Vhost-net moves some of the network interface tasks to a kernel driver. It puts the Virtio emulation code into the kernel, allowing device emulation code to avoid performing system calls from the user-space.

SR-IOV allows a single Ethernet port to appear as multiple separate physical devices to VMs.

Packet Size and Throughput

Baseline UDP transaction tests over one 10GE interface indicated that for small packet sizes (< 512 bytes), the target platform can sustain a transaction rate of about 2M transactions per second in each direction and is CPU bound. Larger packet sizes achieve line rate and are bound by network interface bandwidth.

RSS and Flow Director

For large packets there was no significant difference between RSS and Flow Director (both achieve line rate) however for small packets where throughput is CPU bound, Flow Director produced superior performance from better cache affinity. In reality, both RSS and Flow Director have limitations making them more or less suitable in different situations such as single vs. multi-threaded applications; TCP or UDP; number of flows; etc.

Process and Interrupt (IRQ) Affinity

Process affinity and IRQ affinity strategies can have a dramatic impact on performance including the variability of throughput. In UDP stream tests with RSS, using only one stream, to get highest performance with lowest variability, IRQ and application process should be affinitized to different cores on the same CPU socket. Without careful configuration, the performance will suffer and the measured throughput can vary by over 5Gbps.

In a real-world traffic scenario with many streams, configuring one core to handle interrupts will become a bottleneck (in our setup ~6Gbps with 1472 bytes UDP payload). If interrupts are handled by multiple cores there is some performance scaling however throughput variability also increases and this depends on the pattern of flows. This is particularly evident if the cores handling an interrupt and the application process are on different CPU sockets.

Latency

Observations include that using Intel® DPDK, packets are transmitted by bursts with latency around 80 uS below 100 Mbps (for 64 bytes packets). This latency could be reduced to around 15 uS through Intel DPDK parameter tuning however this reduced the maximum achievable throughput (by up to 15%). Not surprisingly we observed that applications running in a virtual machine have much higher latency (around 3x) than when running on the host (or hypervisor). L3 forwarding in the kernel also has better latency than user space applications.

Virtual Network Interfaces

The example data below illustrates the impact on throughput performance of using different virtual network interfaces to the VM. The data is generated by a Linux user space application in the VM generating packets and sending them to an external network. Note that this configuration represents a slow-path traversing the entire Linux stack in the VM and on the host.

Vhost performs about 10 percent better than virtio for outgoing packets (VM sending packets externally), and virtio performs better in the other direction. Factors influencing performance are:

1. Number of virtual CPUs

A VM with only one virtual CPU performs very badly. Increasing the number of virtual CPUs above two does not increase throughput (even with multiple streams) as there is only one virtual CPU handling the IRQ in the VM (no multi queue virtio), or only one core handling vhost (no multi-threaded vhost).

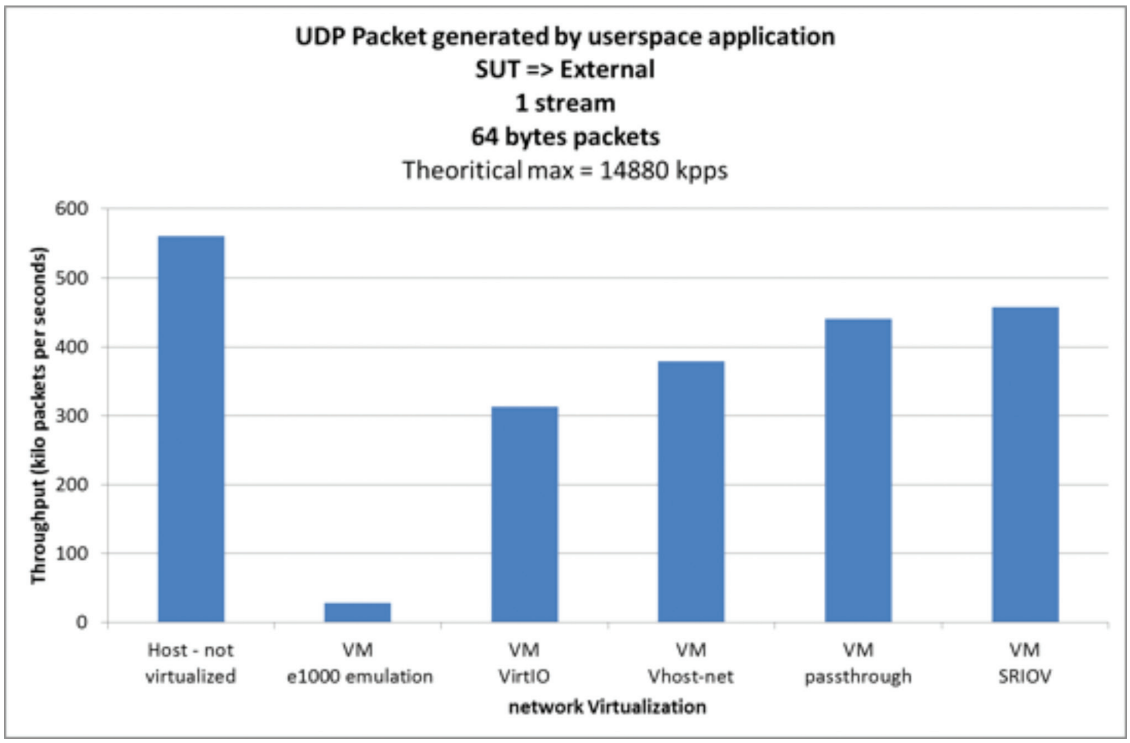


Figure 1: Packet throughput comparing KVM virtual interfaces

2. Number of VMs

When using two VMs sharing the same NIC, virtio scales well. However, when using more than one VM, vhost does not scale: the limiting factor being the vhost-net process. Wind River Linux 4.3.3 is based on a Linux 2.6.34 kernel in which vhost is a single process (i.e. single thread). In the more recent kernels (3.0.x), there is a vhost thread for each VM, and for each VM a vhost thread per interface. Vhost performance should scale with VMs in a future release of Wind River Linux.

3. Number of NICs

Today, using multiple NICs in the same VM does not scale. Using multiple NICs for different VMs gives scaling with virtio but not with vhost (expect better scaling in the future with vhost).

L3 Forwarding Performance

The following graph shows a sample of L3 forwarding performance for the following configurations:

1. L3 forwarding with host kernel [*baseline*]
2. L3 forwarding with Intel DPDK on the host
3. L3 forwarding with VM kernel using virtio
4. L3 forwarding with VM kernel using vhost
5. L3 forwarding with VM kernel using SR-IOV
6. L3 forwarding with Intel DPDK in a VM with SR-IOV

The number of cores in this result has been chosen as being the optimal for each test, i.e. for reaching the best performance for a minimum number of cores; the performance would not increase much using more cores.

SR-IOV related tests achieve their maximum using two cores (one per interface), as the Niantic does not support RSS and SR-IOV together. When using SR-IOV, the virtual interface in the VM has only one queue and cannot scale.

Vhost and virtio show very poor performance in these bi-directional tests. Vhost for instance is limited by its single threaded vhost-net task.

The Intel DPDK, whether on the host or in a VM, reaches far higher throughput than the standard ixgbe(vf) driver.

VM to VM Communication

The performance of VM to VM communication was limited to around 50 percent of line rate with large packets and only 3 percent of line rate with small packets. We observed limited scalability with virtio, vhost, and SR-IOV. Vhost and virtio are limited by single threaded tasks. SR-IOV is not compatible with RSS so a virtual NIC in a VM using SR-IOV can only use one queue. Today, the only way to scale performance is to use multiple VFs (i.e. use multiple IP addresses in each VM) and/or multiple NICs.

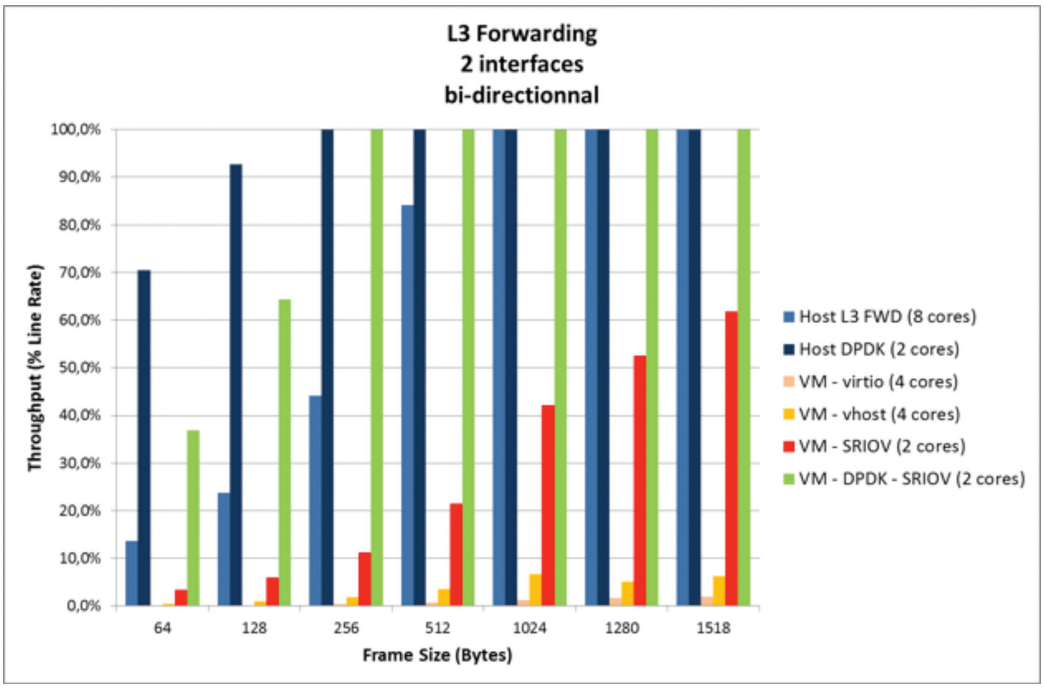


Figure 2: Host and VM L3 Forwarding Performance

Process and Interrupt (IRQ) Affinity

Process affinity determines which CPU cores can be used to execute a task. This is an OS scheduler property.

IRQ affinity is the ability to assign certain IRQs to specific cores.

Process affinity and IRQ affinity strategies can have a dramatic impact on performance including the variability of throughput. If interrupts are handled by multiple cores performance scales however throughput variability also increases and this depends on the pattern of flows. This is particularly evident if the cores handling an interrupt and the application process are on different CPU sockets.

When handling packets in one VM with two virtual cores, the best performance and stability is obtained when appropriately selecting affinities using five different cores in the correct socket and assigning respectively to qemu threads (x3), vhost thread and IRQ handling thread.

If affinities are not carefully selected the performance and/or stability of the results can dramatically decrease. In addition, in the VM itself, the virtual core handling the IRQ must be different from the virtual core running the user process.

All cores are not equal as logical cores can belong to the same or different physical cores (hyper-threading). Physical cores can also belong to the same or to different CPU sockets. Synchronization between two cores of the same physical processors has a far lower overhead than between two cores in different CPU sockets.

The relationship of cores in a system is dependent on the processor and board architecture and can be determined by viewing /proc/cpuinfo. The Linux scheduler is responsible for choosing which core is used to execute a task and is aware of the hardware architecture.

However, Linux is a general purpose OS and the scheduler is not optimized for VM workloads.

PCI Pass-through, SR-IOV and (Input/ Output Memory Management Unit (IOMMU)

SR-IOV and PCI pass-through are techniques used to pass packets into the **guest** OS with low overhead. (Input/ Output Memory Management Unit (IOMMU) must be enabled when using SR-IOV or pass-through.

Intel's IOMMU is called VT-d. This translates DMA addresses to (virtual) machine addresses. So, in order to assign physical devices to the guest, VT-d must be supported and enabled in the BIOS, and IOMMU support must be enabled in the kernel.

Enabling the IOMMU can affect **host** I/O performance significantly.

3 Introduction

3.1 Test Configuration

The diagram below depicts the hardware components used. Software is described in section 3.4. An Xeon E5-2600 processor-based system was used as a reference platform due to availability of recent Linux kernels and provided base-line performance benchmarks. Each system was configured with two Intel 82599 10 Gigabit Ethernet Controllers. See BOM below for detailed hardware configuration.

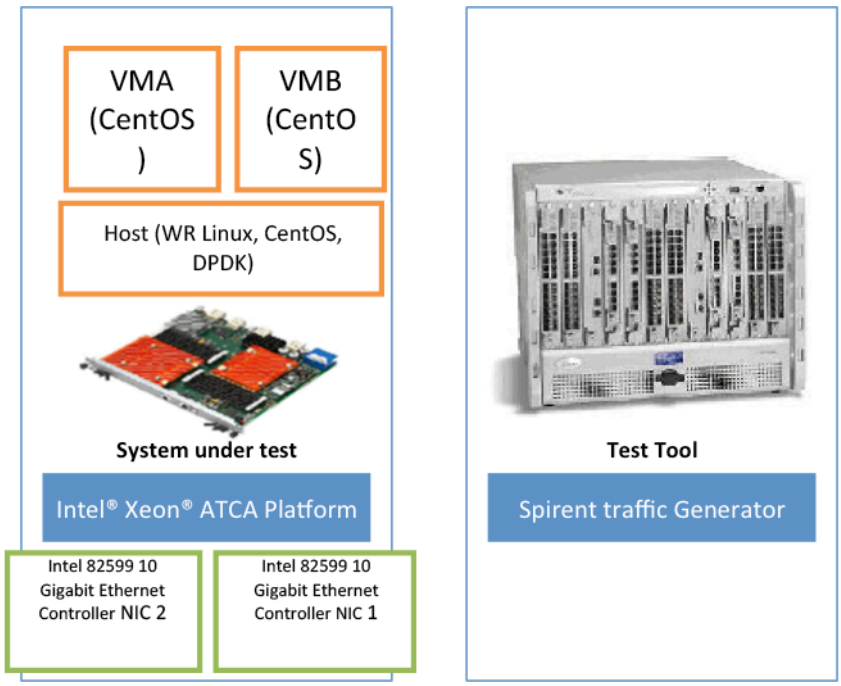


Figure 3: Detailed Hardware Configuration

3.1.1 Use Cases

The focus is packet processing performance in a virtualized COTS environment. The following use-cases are useful when evaluating or architecting a software based network services platform.

- Network service consumes IP packets from the external network. An example use-model is DPI (Deep Packet Inspection) service where packets are inspected and terminated.
- Network service produces IP packets sent to the external network. An example use-model is a video server in an edge node streaming to a client device through the access network.
- Network service does L3 packet forwarding. An example use-model is a media transcoding service that forwards transcoded packets to a client device.
- VM-to-VM communication (located on the same system). An example use-model is a packet processing pipeline where services are hosted in VMs e.g. NAT and packet forwarding.

These use-cases are intended to serve as building blocks to explore higher level network service use-models. Following this "recipe" should dramatically reduce the effort to instantiate a POC platform for exploring new "network cloud" use-models. This RA specifically targets Intel® Xeon® processor class platform known as Crystal Forest (Intel Xeon processor E5-2600, formerly code named Sandy Bridge utilizing the Next Generation Communications Chipset from Intel, codename Cave Creek) with Wind River Linux. Testing started with the Intel Xeon processor E5-2600 based platform formerly code named Romley (Intel Xeon processor E5-2600 utilizing the Intel® X79 Chipset, formerly code named Patsburg) because Crystal Forest and Wind River Linux was not available and the Intel Xeon processor E5-2600 based system, is a suitable reference platform. Small (64B) to large (1518B) packet sizes were investigated in all tests (no packet fragmentation).

3.1.2 Configuration Options

Hardware and software configurations include:

- **Hardware platform**:
  i. Next Generation Communications Platform from Intel, codename Crystal Forest. Dual Intel® Xeon® ATCA Reference Platform.
  ii. A 2-way rack-mount server platform with Xeon® E5-2600 family of processors (formerly code named Romley) was also used for reference to compare certain performance results and to generate/consume packets for certain tests.
- **NIC**: Intel 82599 10 Gigabit Ethernet Controller
  i. RSS
  ii. Flow Director
- **Host OS / IP Stack**
  i. Wind River Linux with native Linux stack
  ii. CentOS with native Linux stack
  iii. Wind River Linux with Intel DPDK
- **Guest OS**: CentOS
- **Hypervisor**: Linux kernel virtualization (KVM)
  i. Stack interface configurations with: E1000 emulation, Virtio, Vhost, SR-IOV
  ii. SMP Interrupt affinity
  iii. Process affinity

3.2 Hardware

3.2.1 Compute Platform

Item	Description	Notes
Platform	Next Generation Communications Platform from Intel, codename Crystal Forest.	http://www.intel.com/content/www/us/en/communications/global-communications-network.html
Form factor	ATCA	Bladed telco server
Processor(s)	2x E5-2648L	20480KB Cache, with Hyper-threading
Cores	8 core/CPU, up to 70W SKU	16 Hyper-threaded cores per CPU for 32 total cores
Memory	2 GB or 16 GB RAM	Quad channel 1600 DDR3 w/ECC. Small and large memory configurations used in tests
NICs (Niantic)	Intel® 82599 10 Gigabit Ethernet Controller	2 ports on front panel and 2 on RTM
PCH	4x Intel, codename Cave Creek	With Intel® QuickAssist Technology
BIOS	2.14.1219	BIOS rebuilt to enable Intel® Virtualization Technology for Directed I/O (Intel® VT-d)

3.2.2 Intel 82599 10 Gigabit Ethernet Controller Network Interface

The platform network interfaces are Intel® 82599 10 Gigabit Ethernet Controller (formerly Niantic). This is the latest Intel 10GE network card that provides dual-port 10Gb/s throughput with improved end-to-end latency and lower CPU utilization using TCP Segmentation Offload (TSO), Receiving Side Coalescing (RSC), and flow affinity filters.

Due to the enhanced onboard intelligence of the Intel 82599 10 Gigabit Ethernet Controller , it can also provide advanced features such as:

- Offload IPsec for up to 1024 Security Associations (SA) for each of Tx and Rx
- AH and ESP protocols for authentication and encryption
- AES-128-GMAC and AES-128-GCM crypto engines
- Transport mode encapsulation
- IPv4 and IPv6 versions (no options or extension headers)

The Intel 82599 10 Gigabit Ethernet Controller uses Linux the ixgbe Driver for 10 Gigabit Intel Network Connections. The ixgbe driver also supports Intel® 82598 and Intel® 82597 10 GE Ethernet Controllers (has fewer features compared to Intel® 82599). Tests here used ixgbe-3.7.21 which by default is not enabled for the ATR mode and UDP.

Niantic default flow-affinity options for UDP:

- No RSS and no Flow Director (default configuration)
- RSS only

For testing the driver was modified (i.e. recompiled) to enable:

- Flow Director only
- Flow Director plus RSS

When considering UDP note that for a few flows with specific packet sizes and well-defined five tuples (IP dst/src, Port dst/src, Protocol) Flow Director will work well. In reality UDP may be used more in a multicast way which could ruin the affinity that Flow Director provides plus create more CPU utilization due to the fact that the hash needs to be calculated multiple times. Also in lab tests there are no fragments; however Flow Director does not filter fragmented packets, so this adds more "noise" (which is a case for both TCP and UDP in real life).

The tests here also document other virtualization related performance features, such as RX/TX queues filtering, SMP interrupt affinity, Physical Function (PF) direct mapping, and Virtual Function (VF) utilization.

3.3 Software Components

Component	Function	Version / Configuration
Wind River Linux*	Host OS	Wind River Linux 4.3.0.0 with kernel 2.6.34.10
CentOS*	Host OS	6.2 is used as a reference host OS to compare with Wind River Linux. ▪ Kernel was updated to 3.2.2 because of known updates for Intel® Xeon® processor E5-2600, formerly code named Sandy Bridge. ▪ CentOS6.2 distribution original configuration file was used to build the patched kernel. ▪ Receive Packet Steering (RPS) disabled. This requires net configuration file modifications. Additional information is available to Intel customers on request. ▪ System Services - firewall disabled, irqbalance service disabled.
CentOS	Guest OS	Also used as the guest OS for virtualization configurations.
Intel® Data Plane Development Kit (Intel® DPDK)	IP stack acceleration	Version 1.1.0.27 used in Wind River Linux tests.

4 Test Methodology

4.1 Measuring Network Throughput

When calculating data throughput, packet overhead needs to be considered as this significantly impacts throughput based on the size of packets. For instance, when sending or receiving 64 bytes Ethernet frames, there are 20 additional bytes consisting of an inter-framing gap, start of frame delimiter and preamble. Therefore using 64 bytes frames, the maximum theoretical data throughput on a 10Gbps interface is 7.619 Gbps. For 1518 bytes Ethernet frames, the maximum theoretical throughput is 9.870 Gbps.

Another consideration is to know which throughput network performance tools such as Netperf (http://www.netperf.org/netperf/) report. For example the Netperf UDP stream test reports UDP payload throughput. A 64 bytes Ethernet frame contains 18 bytes UDP payload., The maximum theoretical throughput with such frames is 2.143 Gbps of UDP payload, or 7.619 Gbps of Ethernet payload; for 1518 bytes Ethernet frames (containing 1472 bytes UDP payload), it is 9.571 Gbps of UDP payload.

4.1.1 Packet Size Definition

There is a difference between an Ethernet frame, an IP packet, and an UDP datagram. In the seven-layer OSI model of computer networking, 'packet' strictly refers to a data unit at layer 3 (Network Layer). The correct term for a data unit at layer 2 (Data Link Layer) is a frame, and at layer 4 (Transport Layer) is a segment or datagram.

4.1.1.1 Ethernet Frame (Ethernet II, IEEE 802.3)

An Ethernet frame is a data packet on an Ethernet link. The length depends of the presence of the IEEE 802.1Q optional tag (indicating VLAN membership and IEEE 802.1p priority) in the Ethernet Header.

If the optional 802.1Q tag is absent:

7 bytes	1 byte	14 bytes	46 to 1500	4 bytes	12 bytes
	Start of frame delimiter	**Ethernet Header**	**Payload**	**FCS/CRC**	**Interframe gap**

If it is present:

7 bytes	1 byte	18 bytes	42 to 1500	4 bytes	12 bytes
	Start of frame delimiter	**Ethernet Header**	**Payload**	**FCS/CRC**	**Interframe gap**

In this document, for the sake of simplicity, we will use Ethernet frames where the IEEE 802.1Q is absent. We will also focus on IEEE 802.3 and Ethernet II frames.

Frame Size

RFC 1242 defines the Data Link frame size as "The number of octets in the frame from the first octet following the preamble to the end of the FCS, if present, or to the last octet of the data if there is no FCS." (RFC 1242). This means that the frame size can vary from 64 to 1518 bytes (when 802.1Q tag is absent) or 64 to 1522 bytes when this tag is present.

Ethernet Header

6 bytes	6 byte	4 bytes	2 bytes
MAC destination	**MAC source**	**802.1Q (optional)**	**EtherType**

The Ethertype is a two-byte field. It takes one of two meanings, depending of its value.

- If the value is less or equal than 1500 bytes, it indicates a length.
- If the value is greater or equal than 0x600 (1536), then it represents a Type, indicating which protocol is encapsulated in the payload of an Ethernet Frame). Typical values are 0x8000 for IPv4 and 0x0806 for an ARP frame.

So, the Ethernet Header length is (when 802 1Q tag is absent) 14 bytes.

Any frame which is received and which is less than 64 bytes (18 bytes Header/CRC and 46 bytes data) is illegal.

4.1.1.2 IP Packet

The IP packet is transmitted as the payload of an Ethernet frame. Its structure is (with each line representing 32 bits):

8 bits		6 bits	2 bit	16 bits	
Version	**Length**	**DSCP**	**ECN**	**Total Length**	
Identification				**Flags**	**Fragment Offset**
TTL	**Protocol**			**Header CheckSum**	
Source IP Address					
Destination IP Address					
Options if header length is greater than 5					
Data					

The "total length" 16-bit field defines the entire packet (fragment) size, including header and data, in bytes. The IP header length is 20 bytes (when options are absent).

The total length of the IP packet varies from 20 to 65535 bytes.

The MTU is the maximum size of IP packet that can be transmitted without fragmentation - including IP headers but excluding headers from lower levels in the protocol stack (so excluding Ethernet Headers).

If the network layer wishes to send less than 46 bytes of data, the MAC protocol adds a sufficient number of zero bytes (0x00, is also known as null padding characters)

4.1.1.3 UDP Datagram

A UDP datagram is transmitted as the data of an IP Packet:

16 bits	16 bits
Source Port	**Destination Port**
Length	**CRC**
Data/ UDP Payload	

Length is the length in octets of this user datagram including this header and the data. This means that the minimum value of the length is eight.

A UDP Payload of 1 byte would result of a 9 bytes UDP datagram (8 bytes UDP header), a 29 bytes IP packet (20 bytes IP header) and a 64 bytes Ethernet frame (14 bytes Ethernet Header, 4 bytes CRC plus 17 bytes padding).

A UDP payload of 18 bytes results in a 26 bytes UDP datagram, a 46 bytes IP packet, and a 64 bytes Ethernet frame (14 bytes Ethernet Header, 4 bytes CRC).

4.1.1.4 TCP Segment

8 bits	8 bits	8 bits	8 bits
Source Port		**Destination Port**	
Sequence Number			
Acknowledgement Number			
Offset	**Reserved**	**Flags**	**Windows Size**
Checksum		**Urgent Pointer**	
Options (+padding)			
Data/ TCP Payload			

The header size varies from 5 words of 32 bits to 15 words of 32 bits, so from 20 to 60 bytes. Option varies from 0 bytes to 40 bytes.

A TCP payload of 6 bytes results in a 26 bytes UDP segment (when no options are present in the TCP header), a 46 bytes IP packet, and a 64 bytes Ethernet frame (14 bytes Ethernet Header, 4 bytes CRC).

4.2 Network Test Tools

Two types of networking tools were used:

1. L3 Forwarding tests were performed using stand-alone traffic generators (Ixia and Spirent). This provides full control on defining the Ethernet frame (TCP or UDP). Be sure to pay attention to how the field length is defined as the CRC may not be taken into account depending on the particular tool being used.
2. Linux networking tools: Information on Linux networking and the basic tools used in these tests is described below.

4.3 Linux Networking

You can send and receive packets from Linux applications using send(), sendto(), recv(), recvfrom() socket API. Both send() and sendto() functions have a "len" field specifying the length of data in bytes passed to the Linux stack.

When using TCP, this len field might not have any link with the size of the packets sent on the line. This is because the Linux stack might use a technique like Nagle's algorithm to determine the packet size of a transmission. This algorithm was developed to avoid congestion by small packets by combining a number of small outgoing messages and sending them all at once. Hence, when an application sends, for instance, 10 messages with 20 bytes of TCP payload, the stack might only send one packet with 200 bytes TCP payload on the wire. If necessary, this algorithm can usually be disabled using TCP_NODELAY as a socket option.

4.3.1 Netperf

When you generate UDP datagrams using netperf in stream tests, you can specify the UDP payload using the "-m" option. It is up to you to calculate the equivalent IP packet size and the Ethernet frame size:

	Size in Bytes							
UDP Payload Size	1	18	82	210	466	978	1234	1472
IP Packet Size (UDP traffic)	29	46	110	238	494	1006	1262	1500
Ethernet Frame Size	64	64	128	256	512	1024	1280	1518

When you generate TCP packets using netperf, you only specify (using –m option) the message size used by netperf in the send() function. As explained earlier, this is not the TCP payload size as sent on the line, because of different algorithms and optimization used by the Linux stack (like Nagle algorithm and TCP segmentation offload). The stack will usually group multiple TCP segments and send them as one packet to the network interface. The network interface might segment the packet if it is too big to be transmitted.

When using TCP_NODELAY (or –D option in netperf), the size specified using –m option in netperf will control the TCP payload size of the packet sent to the NIC. As Linux usually adds TCP timestamp (12 bytes) to the optional part of the TCP header, the maximum TCP payload before the packet is fragmented by the NIC is (1500 – 20 – 20 – 12) = 1448 bytes.

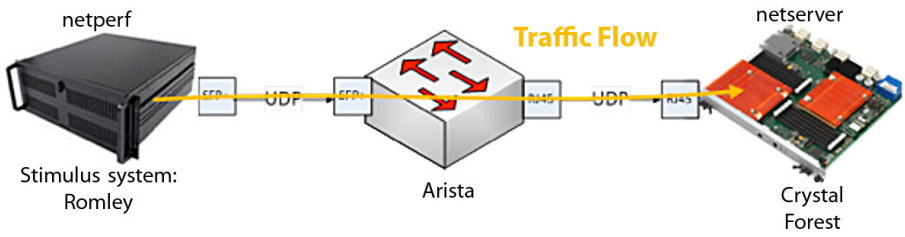
4.3.1.1 UDP Tests

Two different tests from the UDP netperf suite have been used:

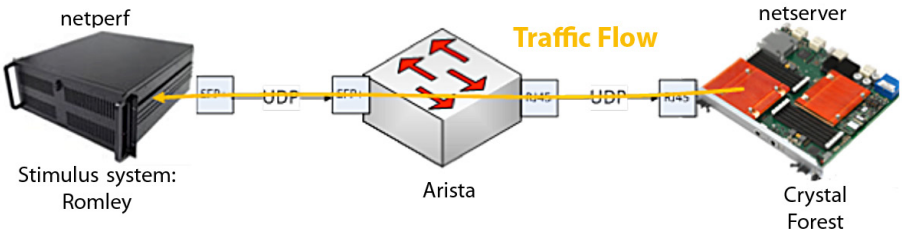
- UDP_STREAM:

  UDP packets of a pre-defined size are generated by the "netperf" application running on the client side. Netserver application, running on the server, counts the packets it receives. Both applications are running in userspace. Different tests are usually run with different payload sizes, reflecting the frame sizes from RFC 1244. This test can be run in both directions:

  "External => Host" simulates an application receiving packets



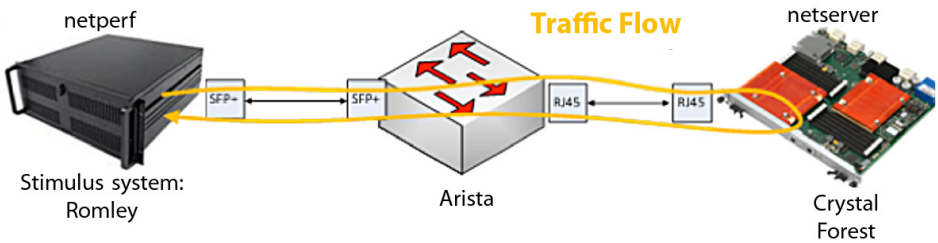
  and "Host => External", simulating an application generating packets.



  UDP_RR:

  This is a request/response test: a transaction is initiated by netperf, sending a packet of a pre-defined size. Netserver running on the server responds to this packet. When netperf receives this response, the transaction is completed and a new transaction can start. In our testing, each netperf was running 100 transactions in parallel.

  In our testing, each netperf was running 100 transactions in parallel:



One pair of netperf/netserver corresponds to one stream, as it uses one source IP, one destination IP, one source port, and one destination port. When running on the host (no virtualization), one stream is usually handled by up to two cores: one handling the IRQ, and one running netperf or netserver.

Some tests were performed using only one stream. This "1 stream" test is not really realistic, as much more than one source/destination IP/ports are used on a real network; but it is useful to highlight some characteristics of the system under test.

Other tests were performed using 32 streams (thus corresponding to 32 instances of netperf and netserver, and 32 different source/destination ports). Using many streams is more realistic, and help to understand how the system behaves when all cores are in use.

Reference http://www.netperf.org/netperf/training/Netperf.html

4.3.1.2 TCP Tests

Similarly, two tests from the TCP netperf suite have been used: TCP_STREAM and TCP_RR. They work just like the UDP tests, but using TCP instead of UDP.

Reference http://www.netperf.org/netperf/training/Netperf.html

4.3.2 Tcpdump

Tcpdump is the well-known tool used to print out a description of the contents of packets on a network interface. It is important to note that usually tcpdump does not capture the CRC. So, a 64 bytes frame will be captured by tcpdump as 60 bytes.

Also, tcpdump does not capture the frame sent or received on the wire, but the frames sent by the linux stack to the network interface, and sent by the network interface to the linux stack. The network interface might still segment a packet, for instance when TCO (TCP segmentation Offload) is enabled. When TCO is enabled, tcpdump might hence capture TCP/IP packets longer than the 1500 bytes MTU.

5 Packet Optimization with Intel 82599 10 Gigabit Ethernet Controller

5.1 Host OS Packet Throughput

The objective was to establish a baseline of packet throughput that the host OS (i.e. no virtualization) can sustain using a default Intel 82599 10 Gigabit Ethernet Controller (ixgbe) driver.

Using netperf UDP_RR, a transaction is initiated by the stimulus system and a response is sent by the SUT. Only one 10Gbps link is used. Netperf ensures that there are 100 transactions active at any time (as soon as a transaction completes, netperf starts a new one). 32 flows, i.e. 32 netperf/netserver instances, are run in parallel. UDP_RSS is enabled (using Linux ethtool) so that different flows (corresponding to different source and destination ports) will be received on different NIC queues. Set_irq_affinity.sh script (provided with ixgbe NIC driver sources) is run so that the IRQ of each NIC queue is handled by a different processor core.

An example of these UDP test results is shown below for different frame sizes.

Ethernet frame sizes of 64B to 512B (UDP payload of 18B to 466B) sustain a similar transaction rate of about 2M transactions/s. For 512B frames this corresponds to a throughput of about 8.2 Gb/s in each direction.

Ethernet frame sizes 1024B to 1518B (UDP payload of 978B to 1472B) sustain a smaller transaction rate of between 800k and 1.2M transactions/s. These correspond to 10GE line rate in each direction.

The transaction rate drops significantly for 1024B and larger Ethernet frames

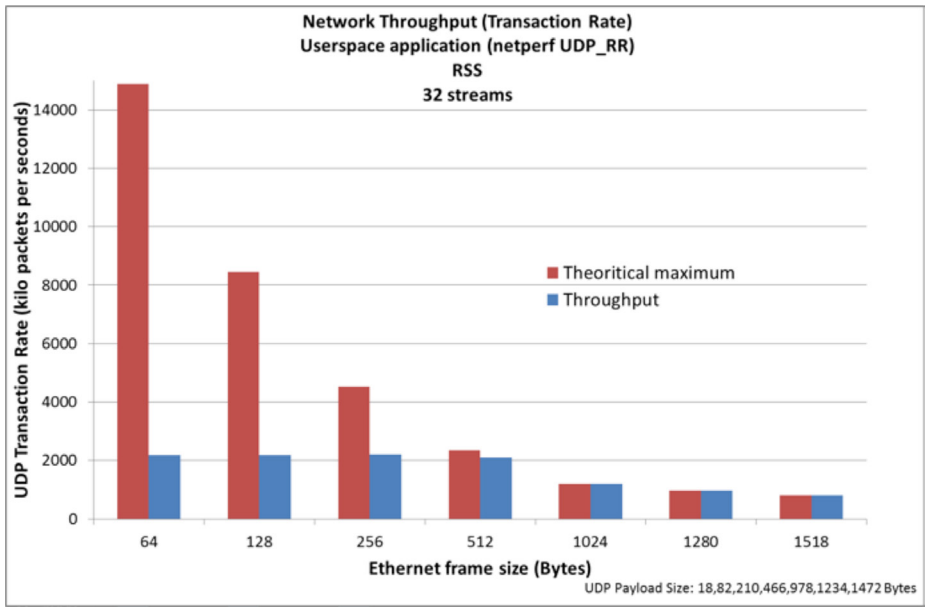


Figure 4: Packet throughput of Linux host as a function of frame size

however throughput at these frame sizes is similar (line-rate) indicating that performance is limited by network interface bandwidth.

For smaller packet sizes, higher transaction rates do not sustain line-rate indicating that performance is CPU bound.

5.2 RSS and Flow Director

Recent NICs like Intel 82599 10 Gigabit Ethernet Controller can classify incoming packets into different NIC receive queues. Classification can be based on 5-tuple i.e. 1): Source IP Address, 2) Source IP Port, 3) Destination IP Address, 4) Destination IP Port, and 5) Data type (UDP, TCP, etc.) The kernel assigns each RX queue to an interrupt through a Message Signaling Interrupt (MSI-X). Each interrupt in turn can be "bound" to a different core (configured by the user) i.e. SMP IRQ affinity. This guarantees that packets belonging to the same flow are processed in-order by the same core. To benefit from cache affinity, a network service handling a particular flow should be run on that same core.

Flow Director and RSS (Receive Side Scaling) are two different features of Intel 82599 10 Gigabit Ethernet Controller with advanced on-board network controller card features. These features can greatly improve the system level performance if configured appropriately based on traffic and application characteristics.

The objective was to observe the effect of RSS and Flow Director on packet throughput.

A transaction is initiated by the stimulus system (running netperf). The system under test (SUT) (running netserver) replies. Netperf ensures that there are 100 transactions active at any time. 32 flows, i.e. 32 netperf/netserver instances are run in parallel.

UDP_RSS is enabled (through "ethtool –N ethX rx-flow-hash udp4 sdfn") so that different flows, corresponding to different source and destination ports, will be received on different NIC queues.

Set_irq_affinity.sh script (provided with ixgbe NIC driver sources) is run so that the IRQ of each NIC queue is handled by a different processor core.

Note that the ixgbe NIC driver was modified to enable Flow Director for UDP as this is not the default configuration.

An example of test results is shown below (expressed in packets per second) using UDP, for different frame sizes comparing the baseline (no NIC receive queues) with RSS and Flow Director.

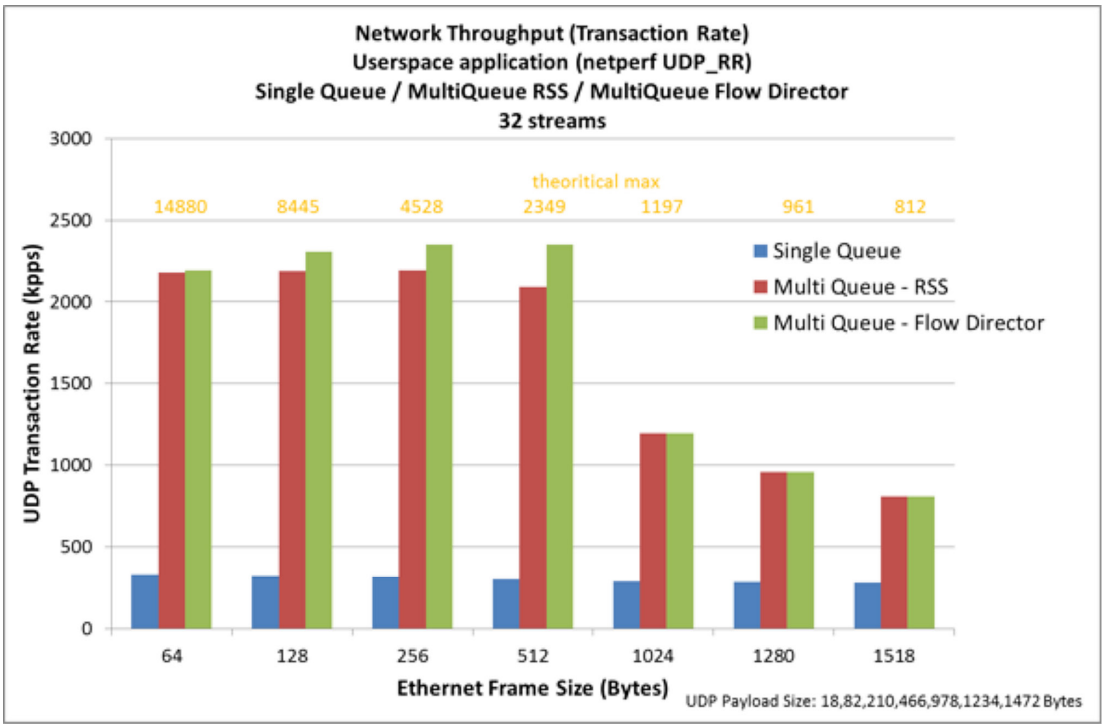


Figure 5: Packet throughput of Linux host as a function of frame size comparing NIC queue schemes

5.2.1 RSS Overview

In Receive Side Scaling (RSS), a hash function is applied to the header fields of the incoming packets (IP Source, IP destination, Port Source, Port Destination, Protocol) and the resulting hash is used to select one of the RX queues (the 7 LSB of the 32-bit hash result are used as an index into a 128 entry redirection table, each entry providing a 4-bit RSS output). Because of the 4-bit output, the maximum number of queues selectable by RSS is 16. With a large number of flows, there will be a statistical distribution of traffic across queues.

With RSS, the RX queue is based only on a hash of the incoming packet; this queue is assigned to a dedicated interrupt which is handled by a core. As the NIC has no knowledge which core will process the packet, the core handling the NIC queue interrupt will usually be different from the core running the applicable network function.

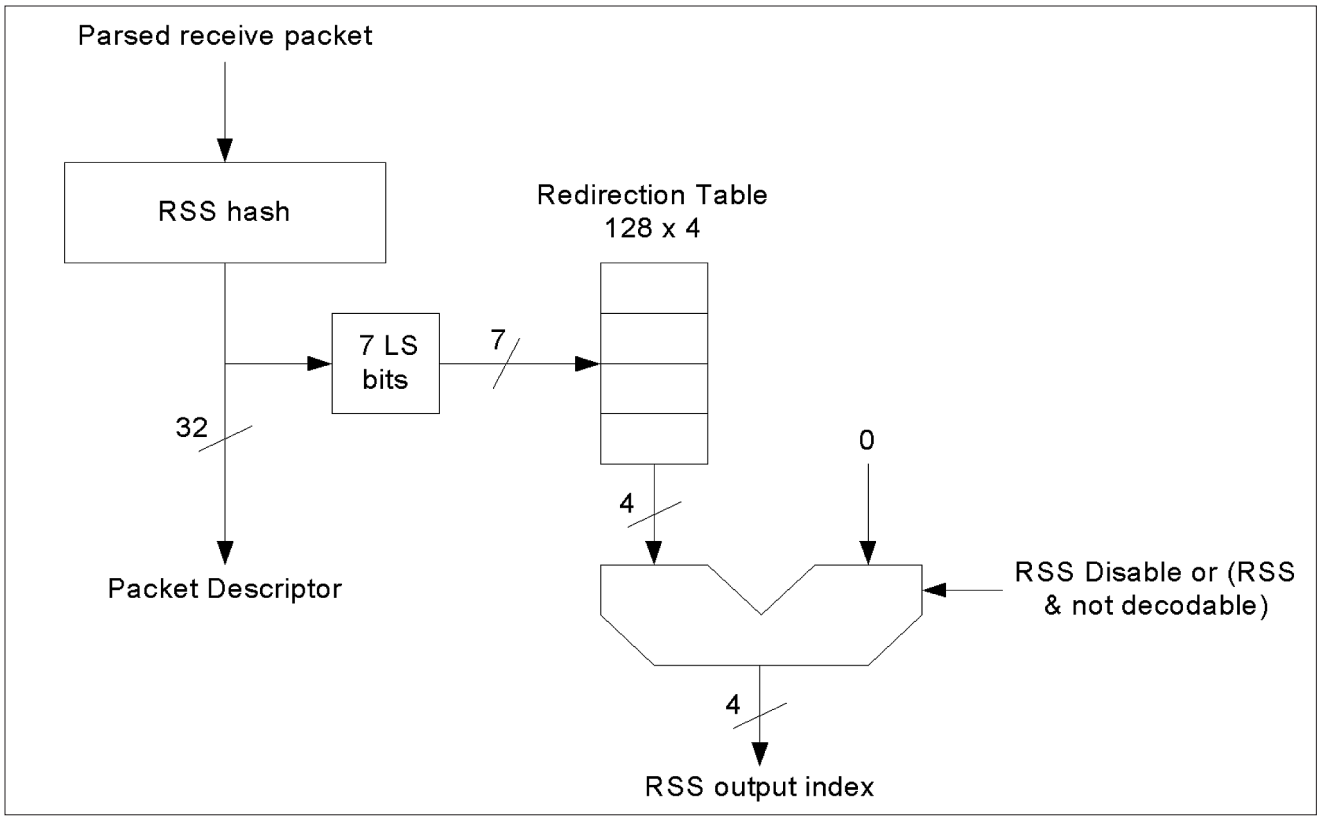


Figure 6: RSS Queue Logic

5.2.2 Flow Director Overview

Flow Director supports two different modes: hash based filtering (ATR) and perfect filter mode. Only ATR has been used in the tests described in this document and is described below. In perfect filter mode, the hardware checks a match between the masked fields of the received packets and programmed filters; refer to "Intel® 82599 10 Gigabit Ethernet Controller Datasheet" for more details.

During transmission of a packet (every 20 packets by default), a hash is calculated based on the 5-tuple. The (up to) 15-bit hash result is used as an index in a hash lookup table to store the TX queue. When a packet is received, a similar hash is calculated and used to look up an associated Receive Queue.

For uni-directional incoming flows, the hash lookup tables will not be initialized and Flow Director will not be effective. For bi-directional flows, the core handling the interrupt will be the same core processing the network flow (see figure 7).

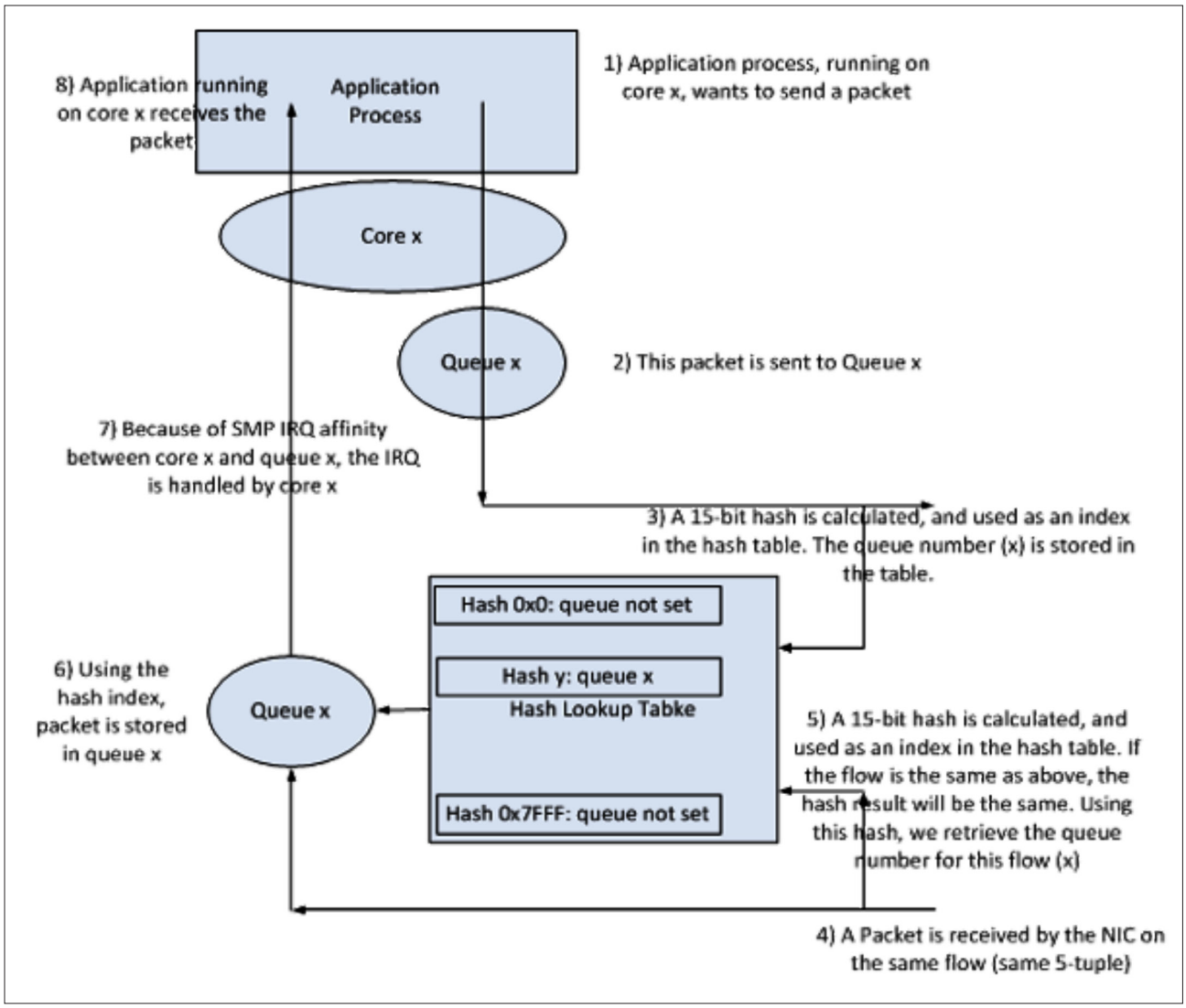


Figure 7: Flow Director Queue Logic

5.2.3 RSS versus Flow Director Priority

Figure 8 shows that what happens when RSS and Flow Director are both enabled. If a packet matches Flow Director filters, then the Rx queue is defined by the Flow Director rules. If the packet misses the Flow Director Filter, then the Queue is defined by the RSS index.

5.2.4 Comparing RSS and Flow Director Performance

We observed no significant difference between RSS and Flow Director for large frame sizes (both reach line rate). For the smaller frames sizes, where throughput is CPU bound, Flow Director had better performance than RSS because the thread handling the network flow and the thread handling the interrupt were running on the same core, indicating better cache affinity.

5.2.5 RSS and Flow Director Limitations

5.2.5.1 One Core Handling the IRQ

For single threaded applications, Flow Director performance can be lower than RSS as the thread handling the interrupt, and the application thread handling the network flow competes for the same core. When using RSS, both threads will usually be on two different cores.

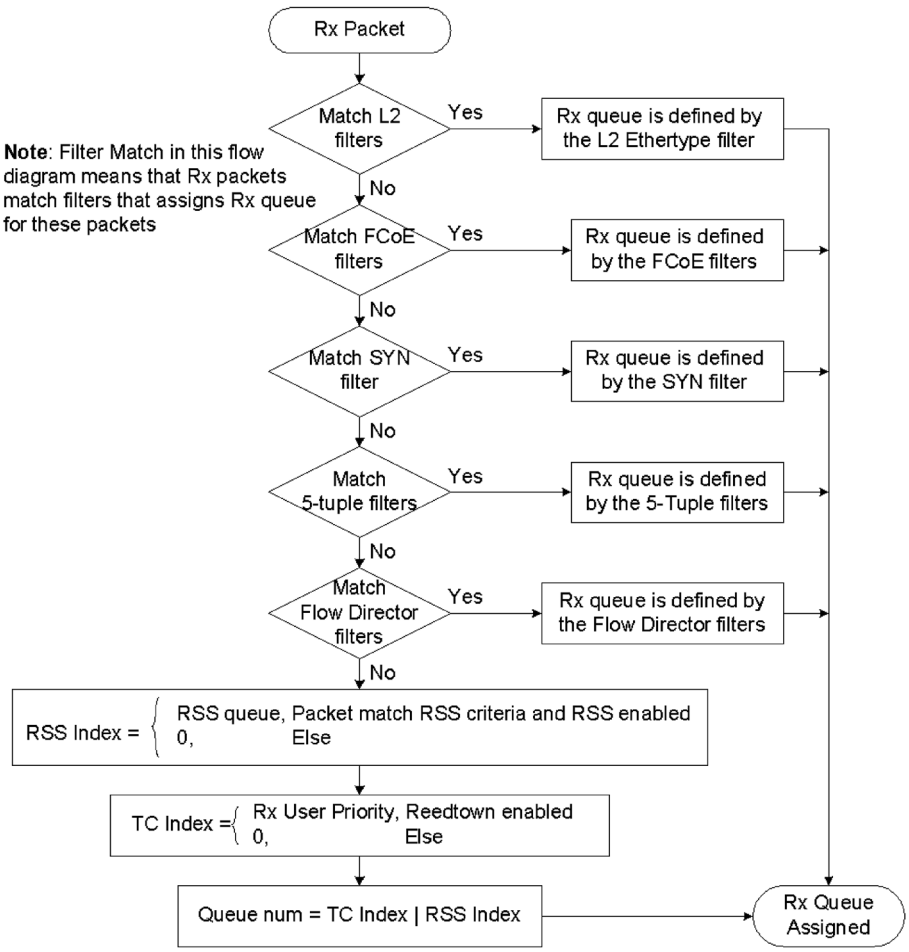


Figure 8: Logic when RSS and Flow Director are enabled

5.2.5.2 Flow Director for TCP

Currently, the IXGBE device driver only enables Flow Director for TCP.

Flow Director is not enabled for UDP by default. There are a limited number of hash entries for Flow Director and a large number of UDP flows typical in real systems would disrupt established TCP flows (i.e. which queue the flow is received), until the flow affinity is reestablished.

5.2.5.3 Flow Director for UDP

UDP Flow Director can provide uniform processing in the case of a UDP netperf to netserver test, where a limited number of flows exist, and the flow information is not disrupted. For reported results a patch was developed to specifically test the performance of flow director in UDP and compare it to RSS.

5.2.5.4 Number of Flows

Flow Director uses a hash table lookup, hence the number of flows supported by Flow Director is limited. If a higher number of flows are received, the performance will decrease.

5.2.5.5 Packet Reordering

Wu et al. (see references) have shown that Flow Director can cause packet reordering. When using flow director, the RX queue is chosen so that an incoming packet will be handled on the same core as the one running the application thread handling the network flow. If, for any reason, the Linux scheduler schedules this thread on a different core, when a packet is being sent by the thread, the new core ID is passed to the NIC, and the corresponding entry in the "flow=>core" table is updated. When a new packet comes in, it will be sent to the new RX queue. If there were still packet of that stream in the previous RX queue there is a risk of packet reordering.

Note that, if the application processes are affinitized to run on specific cores, then this problem will not occur.

6 Packet Performance with Virtualization

6.1 Emulated NIC / VhostNet / Virtio

Our objective is to measure the throughput handled by an application (netperf/netserver) running in a VM, using KVM hypervisor. Both outgoing (VM towards external system) and incoming (external system toward a VM) traffic are measured. Three different Linux network virtualization configurations are compared: e1000 emulation, virtio, and vhost-net.

Traffic (packets consumed on the SUT) is generated using netperf on an external system and consume in the VM. Outgoing traffic is generated using netperf in the VM and forward to the external system. Only one stream of UDP traffic is generated for this test.

The VM was configured to run with two virtual CPUs. A VM is handled by a qemu-kvm process running three threads (Note: a N virtual CPU VM would be handled by N+1 threads).

Results were obtained using:

- One core for handling the IRQ,
- One core for vhost-net (when enabled), and
- Three cores running the three qemu-kvm threads.

When using virtio for incoming traffic, CPU usage is close to 400 percent (4 cores used at 100 percent): the three qemu threads and the IRQ handling thread are almost at 100 percent CPU.

When using vhost-net, CPU usage is similar, i.e. around 400 percent. This time two qemu threads, the vhost thread, and the IRQ handling thread are closed to 100 percent.

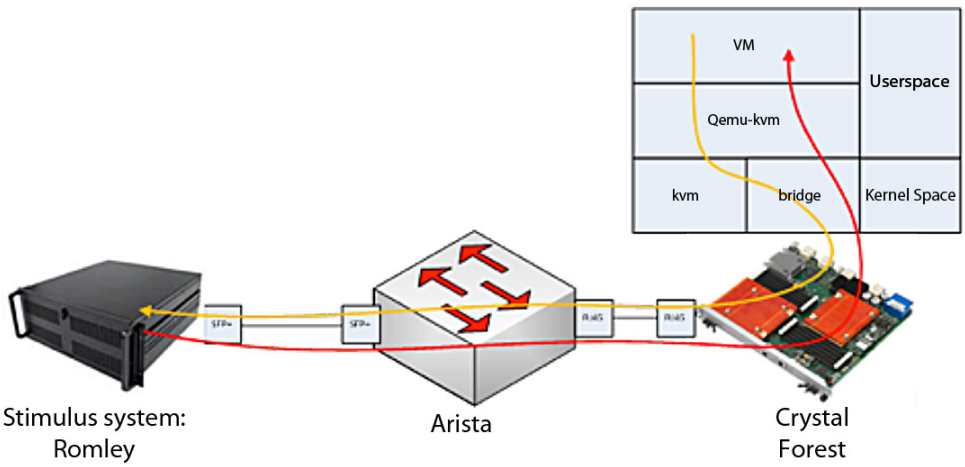


Figure 9

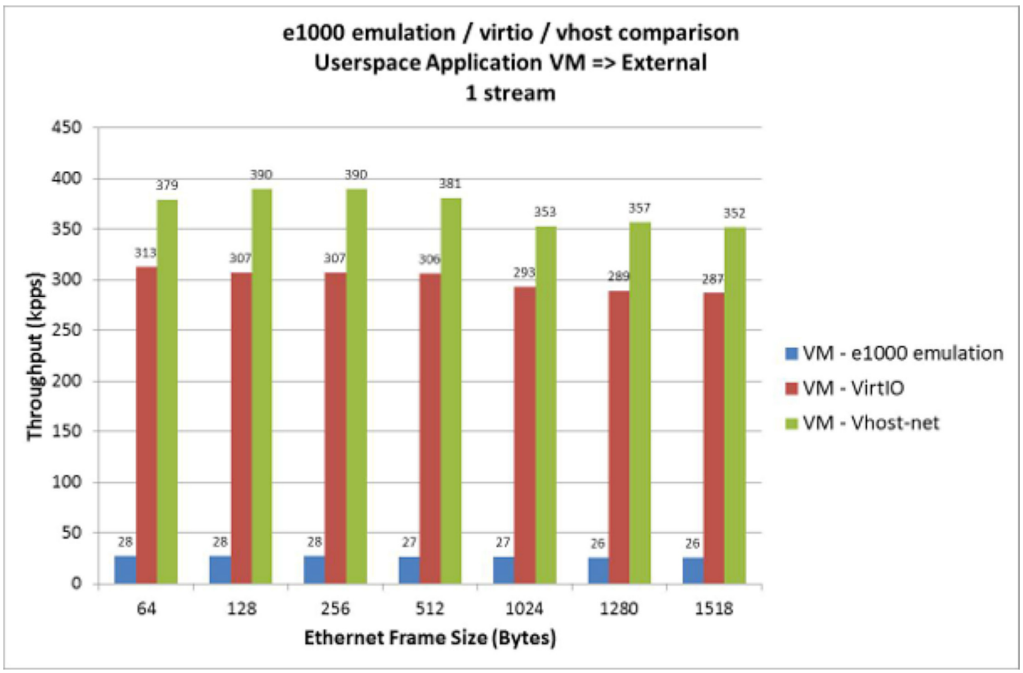


Figure 10: Virtualization: packets sent to an external system from a VM

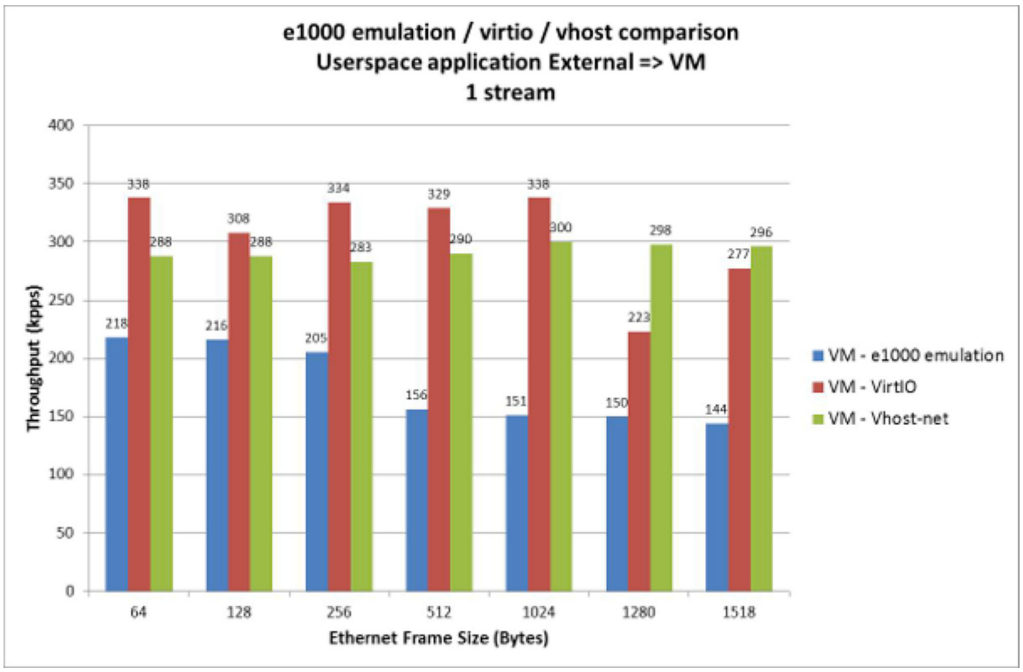


Figure 11: Virtualization: VM receives packets from an external system

When using full virtualization (i.e. e1000), the hypervisor emulates a physical device. This allows any operating system function to be virtualized however this emulation is very inefficient.

Virtio is a Linux standard for network and disk drivers where the guest's device driver knows it is running in a virtual environment, and cooperates with the hypervisor. This is architecturally equivalent to Xen para-virtualized drivers and VMware's Guest-Tools.

Vhost-net moves some of tasks (virtio descriptors to skbs conversion and back) from the qemu userspace to a kernel driver. Basically, it puts the virtio emulation code into the kernel, allowing device emulation code to avoid performing system calls from the userspace.

The above graphs show the huge impact of using virtio and/or vhost, compared to e1000 emulation. Vhost performs 10 percent better than virtio for outgoing packets (VM sending packets externally), and virtio performs better in the other direction. Factors influencing performance are:

Number of Streams

The previous tests were done using only one stream. Real life use cases usually involve many streams. In a non-virtualized case, when using multiple streams, we have seen than having multiple cores handling the different queues improves the performance.

In a virtualized system, this is different, as we see that the IRQ handling core is not the only limiting factor: when using vhost, the vhost core is also at 100 percent CPU. So, having two cores handling the RX queues may not improve the performance, as we hit the vhost limitation. Using virtio is similar: increasing the number of cores handling the IRQ on the host does not improve the performance.

Number of Virtual CPU

A VM with one virtual CPU had very bad network throughput. So, a recommendation is to use at least two virtual CPUs.

Increasing the number of virtual CPUs above two does not increase throughput (even with multiple streams) as there is only one virtual CPU handling the IRQ in the VM (no multi queue virtio), or only one core handling vhost (no multi-threaded vhost).

Number of VMs

When using two VMs sharing the same NIC, virtio scales well and each VM can send/receive the throughput shown in the graph above (i.e. 2x ~3.3 Gbps outgoing and 2x~4Gbps incoming). Note that results were obtained with only one core handling the interrupts.

However, when using more than one VM, vhost does not scale: the limiting factor being the vhost-net process. Wind River Linux 4.3.3 is based on a Linux 2.6.34 kernel. In those kernels, vhost-net is a single process (i.e. single thread). In the more recent kernels (3.0.x), there is a vhost thread for each VM, and in each VM a vhost thread per interface. So, the expectation is that vhost performance should scale with the number of VMs in a future release of Wind River Linux.

Number of NICs

Today, there is no scaling with multiple NICs in the same VM.

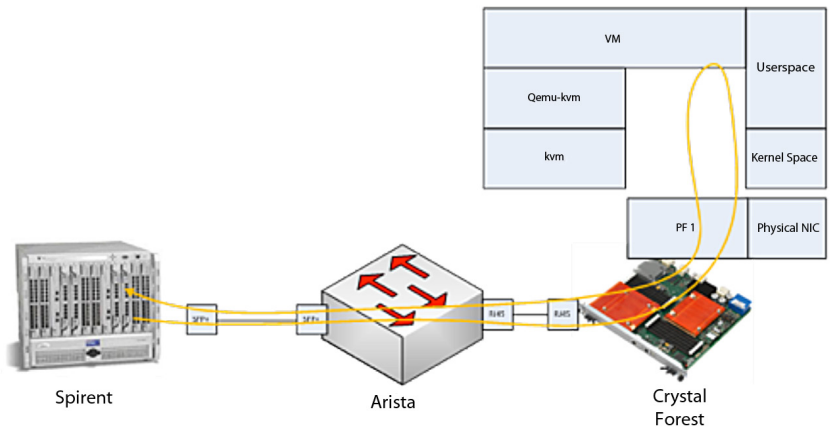
Using multiple NIC in different VMs we have similar results as described above i.e. scaling with virtio, no scaling with vhost (expect scaling in the future with vhost).

6.2 SR-IOV, RSS

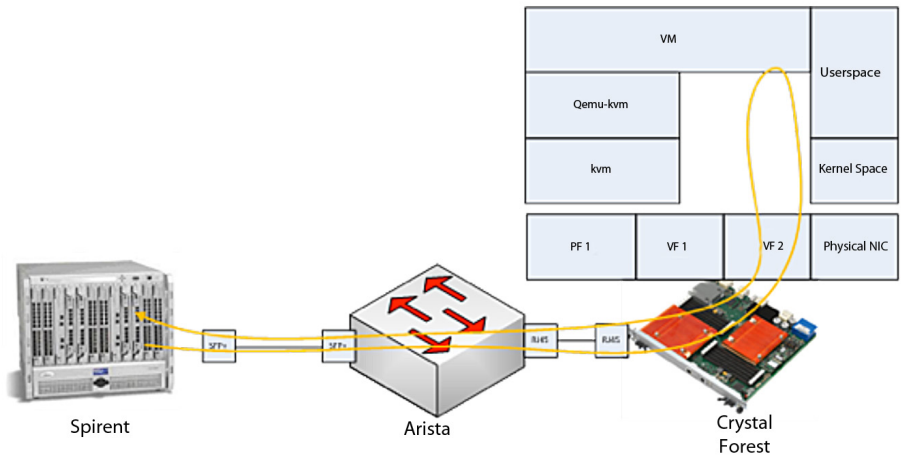
PCI pass-through allows PCI devices to appear as if they were physically attached to the guest operating system. In addition, SR-IOV allows a single Ethernet port to appear as multiple separate physical devices.

The objective is to measure the L3 forwarding throughput of the VM kernel, using KVM hypervisor to compare PCI pass-through and SR-IOV.

Spirent generates 32 flows of UDP data. Frames are received and forwarded by the VM kernel. Note that only uni-directional L3 forwarding has been configured here, i.e. the Spirent generated traffic on only one of its interfaces, and received traffic on the other interface. In pass-through, the full NIC access is given to the VM and the host has no access to the NIC:



With SR-IOV the VM has access to some virtual functions on the NIC however the host still has access to Physical functions and to other virtual functions:



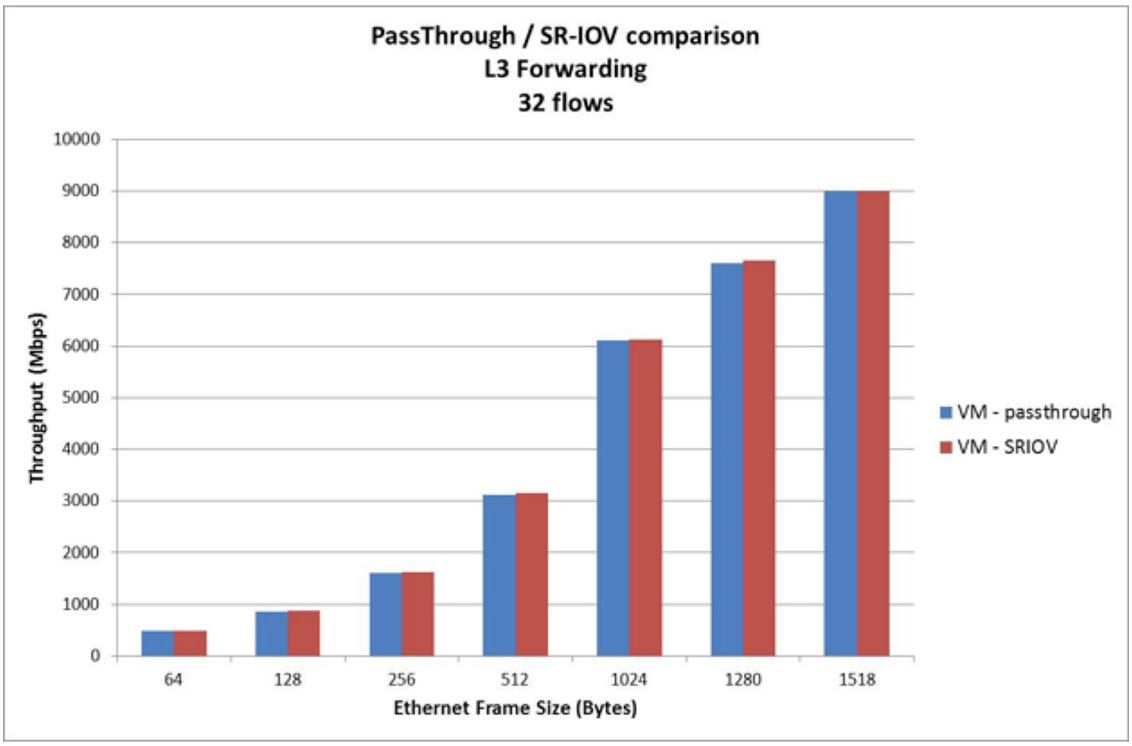


Figure 12: PCI Pass-through and SR-IOV L3 forwarding

The PCI-SIG Single Root I/O Virtualization and Sharing (SR-IOV) specification defines a standardized mechanism to create natively shared devices. SR-IOV enables a single Ethernet port to appear as multiple, separate, physical devices. A physical device with SR-IOV capabilities can be configured to appear in the PCI configuration space as multiple functions. Each device has its own configuration space complete with Base Address Registers (BARs). SR-IOV is about identifying, associating, and managing NIC partitions. There are two basic components:

Physical Functions (PFs):

- These are full PCIe functions that include the SR-IOV Extended Capability.
- The capability is used to configure and manage the SR-IOV functionality.

Virtual Functions (VFs):

- These are "lightweight" PCIe functions that contain the resources necessary for data movement but have a carefully minimized set of configuration resources.

For direct PCI pass-through, a host registers a pass-through interrupt handler for IRQ on behalf of the guest. Interrupts received by the host are injected into the guest and the guest acknowledges the virtual APIC. Guests can dynamically change the IRQs of the device handle with trapping writes to PCI configuration space.

Intel VT-d and IOMMUbased SR-IOV devices can share a single physical port with multiple virtualized guests.

Additional tests should be done to verify how a VM scales with the number of streams - measure scaling when using multiple VMs using the same interface, and measure scaling when using multiple VM using multiple interfaces.

6.3 (Input/Output Memory Management Unit) IOMMU

To use SR-IOV or Direct Pass-through, an IOMMU must be enabled. Intel's IOMMU is called VT-d. It translates DMA addresses to (virtual) machine addresses.

So, in order to assign physical devices to the guest, you need a system which supports VT-d (VT-d must be enabled in the BIOS), and enable IOMMU support in the kernel. Here we see how enabling IOMMU (necessary for enabling PCI pass-through or SR-IOV, and giving direct access to physical or virtual NIC functions to a guest) affects host performance. We measure the throughput handled by an application (netperf) running on the host system and observe the influence on the host performance of enabling the IOMMU.

A UDP stream is initiated by the stimulus system (running netperf). SUT (running netserver, on the host) counts the packets it can receive. Only one stream is generated by the Romley system.

Three cases are tested. They all represent traffic flowing from an external system towards a host system.

1. IOMMU disabled in the kernel.
2. IOMMU enabled in the kernel (intel_iommu=on)
3. IOMMU enabled in the kernel, and IOMMU pass-through enabled (intel_iommu=on, iommu=pt).

The option name iommu=pt (with pt for pass-through) is confusing. In the context of this document, the term pass through was used as the fact of giving to the guest direct access to a physical device. We needed to enable IOMMU support for that. The term pass-through in "iommu=pt" has a different meaning – we talk about host behavior here, and it means that DMA remapping is bypassed in the Linux kernel.

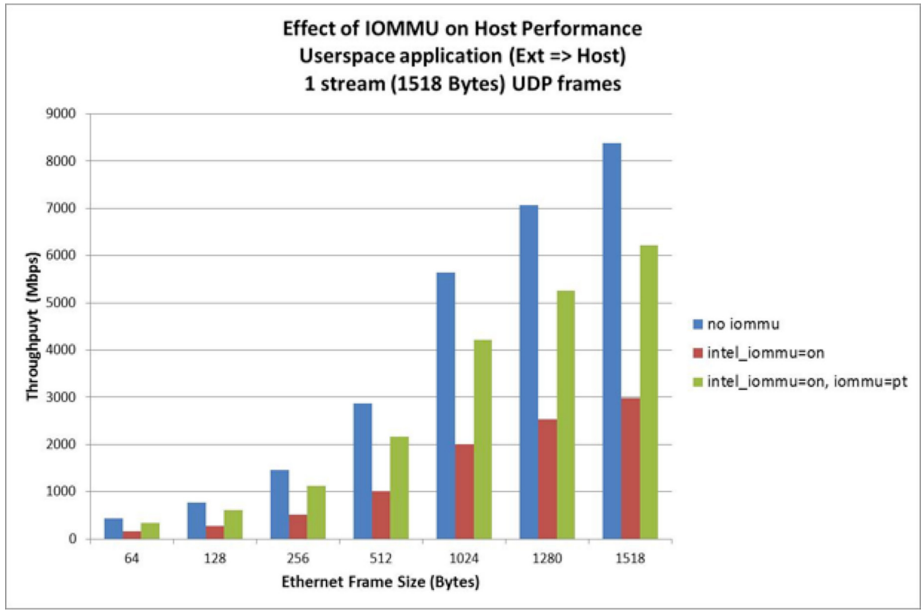


Figure 13: IOMMU effect on Performance

SR-IOV and direct pass-through are techniques used in virtualization to get high network performance in the guest OS. IOMMU must be enabled when using SR-IOV or Direct Pass-through. This results in host performance degradation (more than 50 percent performance drop on some workloads).

Enabling "iommu=pt" on the grub command line partially mitigates this performance degradation issue. This option has no influence on VM performance.

If you want to use SR-IOV to get good performance on the guest OS but still have good performance on the host, you must enable both "intel_iommu=on" and "iommu=pt".

7 Packet Optimization with Affinitization

When using the Linux KVM, the host OS schedules processing but this will not be optimal. Packet throughput can usually be significantly improved by using affinitization i.e. execute certain tasks on predefined cores or core sets. Trade-offs include executing related tasks on the same core, to optimize cache affinity and reduce synchronization overhead versus executing tasks on different cores to benefit from parallel processing on the multicore architecture.

Additional complexity comes from the fact that synchronization cost is different when two logical cores represent one physical core (hyper-threading) compared to two physical cores (no hyper-threading), and whether the physical cores are on the same CPU socket or not.

This discussion is based on use-cases that were tested with Linux KVM (CentOS) using a bridged network configuration common on all Linux kernel versions. Other network software configurations such as a Virtual Switches require additional study.

7.1 Core Process Affinitization

Core process affinitization determines which CPU cores can be used to execute a program and/or to handle interrupts. The process's CPU affinity is a scheduler property that bonds a process to a given set of cores on the system. This is distinct from IRQ affinity which is a scheduler property that bonds certain IRQs to specific cores.

All cores in a system are not equal:

- Some logical cores can belong to the same physical cores (hyper-threading).
- Physical cores can belong to the same or to different CPU sockets. Synchronization between two cores of the same physical processors has a lower cost than synchronization between two cores in different physical CPU sockets.

The relationship of cores in a system is dependent on the processor and board architecture and can be determined by viewing /proc/cpuinfo.

The Linux scheduler is responsible for choosing which core is used to execute a task and is aware of the hardware architecture. Linux is a general purpose OS and the scheduler is not optimized for a particular VM workload configuration.

Process affinitization to the cores is done with the taskset utility either during task creation or after task creation using the task's process id (e.g. taskset 0x3 ./ my_application to start my_application on core 0 and 1).

7.2 Interrupt Affinitization

Setting affinity for interrupt processing is different to core process affinitization. The Intel 82599 10 Gigabit Ethernet Controller supports multiple receive and transmit queue pairs which are assigned unique interrupt request numbers. The core affinity for interrupt processing can be set based on the interrupt number.

By default, the software interrupt routine handling the packets can be run on any core: Irqbalance daemon distributes interrupts automatically to every core in the system. Interrupt affinity can be used instead to restrict interrupt handling to some cores, preserving cache affinity and decreasing inter-socket communications.

Reviewing the default interrupt affinity can be done by looking in /proc/interrupts to find the interrupt number of interest for the target interface and /proc/irq/(irq number)/smp_affinity to see the current interrupt affinity. A new interrupt to CPU core affinity can be set in /proc/irq/(irq number)/smp_affinity.

The interrupt process balance daemon irqbalance often needs to be disabled before setting the interrupt affinitization since it can move the interrupt processing to a different core.

7.2.1 IRQ Affinitization Example

One UDP stream (1472 bytes UDP payload) is initiated. SUT (running netserver) counts the packets it can receive. The default ixgbe driver (i.e. without the patch for supporting Flow Director) is used.

Only one stream (one instance of netperf/netserver) is used in this case. For this test, the application on the SUT (netserver) is affinitized to one specific core (core 8). We study the impact of different IRQ affinity schemes on the results:

- Using set_irq_affinity.sh script, set interrupt affinity so that the interrupts are handled by 32 different cores (core 0 – 31). Note that, when using RSS, only 16 queues are used (Queue 0 to 15) so only cores 0 to 15 will handle interrupts.
- All 16 RSS queues (IRQs) are handled by one core: core 9, i.e. a different core that the core running netserver. Both cores use the same processor (platform socket).

- All 16 queues (IRQs) handled by one core: core 0, i.e one core on the other platform socket.
- All 16 queues (IRQs) handled by one core: core 8, i.e the core already running netserver.

Tests were run multiple times and the minimum and maximum results are shown in Figure 14.

The graph below shows the effect of affinities on performance. It represents UDP stream performance from an external system towards the WindRiver SUT (netperf udp_stream test), using 1500 bytes packet size (1472 bytes UDP payload, 1518 bytes Ethernet Frame). Flow Director was not used for this test.

The first bar is a test where IRQ has been affinitized to 16 cores, using set_irq_affinity.sh script provided with ixgbe sources. Netserver always runs in this test on core 8. Results vary significantly from one test-run to the next, with throughput varying from around 1Gbps to 6 Gbps! Depending on the stream queue, the packets will be handled in netserver by the same core already handling the IRQ, by a different core running on the same socket, or by a different core running on a different socket.

The second bar is a test where the core handling the IRQ and the core on which netserver is running are different, but reside on the same socket. The third bar is a test where both cores are on different sockets. The fourth bar represents the worst case: the application and the interrupts are handled by the same core. For all results, minimum and maximum of ten identical test executions are shown and illustrate the possible variability of the results.

In real-world situation (i.e. non lab-benchmarks), the system is dealing with many streams. In this case, the configuration options are more complex. If only one core handles the interrupts, this core becomes a bottleneck at about 6Gbps. If we configure multiple cores to handle the interrupts (for instance one core per queue i.e; 16 cores for the 16 RSS queues or 32 cores for the 32 Flow director queues), then there is some scaling, but the variability of the results increase, depending on whether the different flows are equally spread to the different Rx queues, and whether the core handling the interrupt and the core handling the application are on the same CPU socket.

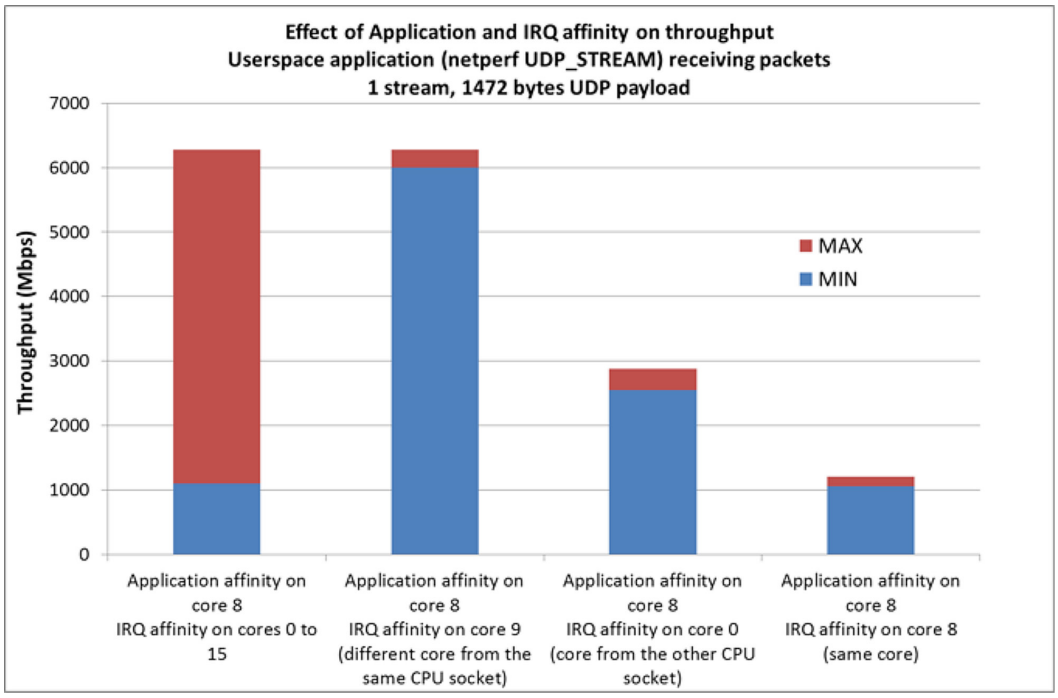


Figure 14: Interrupt Affinity Schemes

7.3 VM Affinitization Methods

See appendix for tasks involved when a VM handles packets. This indicates which component must be affinitized to obtain the best performance. The main tasks are described below. Whether a KVM virtual machine starts using the command line or through tools such as virt-manager, each VM corresponds to a different QEMU process.

7.3.1 Affinitization of QEMU Virtual Processors

When QEMU is started using KVM, the main QEMU task creates a processing thread on the host for each virtual processor of the guest operating system. These threads are independently scheduled on the host. Hence for each VM with N virtual cores there are usually N (children) + 1 (parent) QEMU threads running on the host. Each of those threads can be affinitized to one or more hyper-threaded cores.

7.3.2 Affinitization of vhost

If vhost is enabled in the QEMU arguments during startup, for more recent kernels, a vhost task is created for each sub-network connected to the VM and is unique to the QEMU instance for a particular VM (and terminates when QEMU exits).

For older versions of vhost, only one global vhost task is created to service all VMs. If network data rates are moderate or high, the vhost global task can quickly become the bottleneck for systems with more than one VM and/or a VM with more than one network interface.

The vhost task/s can also be affinitized to one or more hyper-threaded core.

Vhost has no affinitization by default even when QEMU is affinitized at startup. When the VM is started by QEMU, taskset can be used to affinitize the VM to the host cores.

QEMU and vhost tasks are usually assigned to different physical CPU sockets by the Linux scheduler and this behavior seems to be consistent for most configurations. Having vhost and QEMU tasks running on different physical CPUs results in poor performance since vhost and QEMU tasks transfer all packets between each other. To correct this issue affinitize the associated vhost tasks to cores in the same socket as the QEMU tasks.

7.3.3 Affinitizing Tasks Within a VM

Affinitizing the VMs tasks in Linux guest OS is done in the same manner as on the host using taskset, however the virtual core ID is used. In order to have a specific VM task affinitized to a specific physical core, the associated VM virtual CPU (QEMU thread) must also be affinitized to a physical core.

7.3.4 Affinitizing Interrupts Within a VM

For Linux VM guests, interrupt processing is affinitized to virtual CPUs in the same manner as with interrupt affinitization on the host (including irqbalance considerations as discussed above). In order for the VM interrupt processing to be affinitized to a physical core, the VM virtual CPU (QEMU thread) must also be affinitized to a host CPU core.

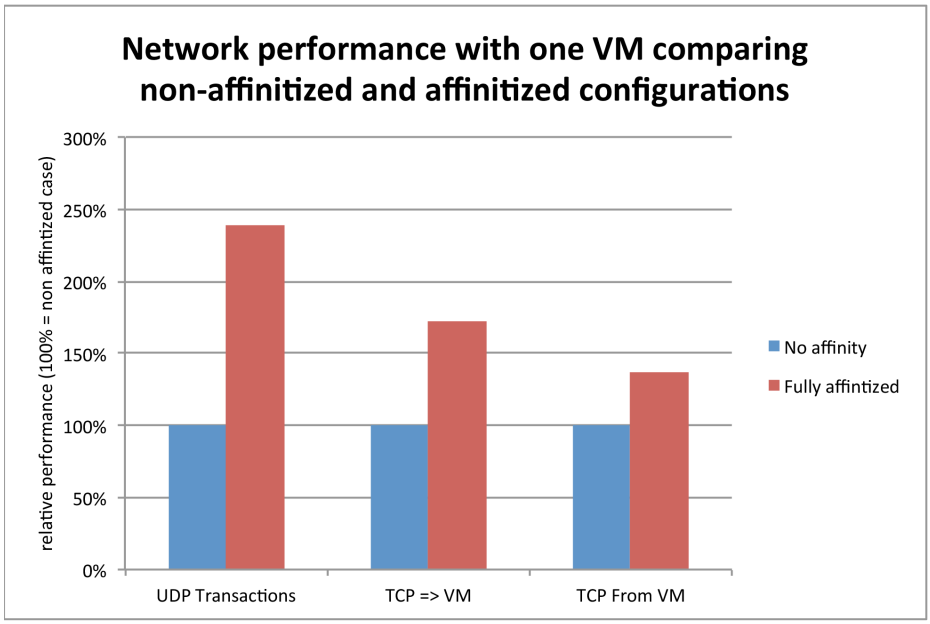


Figure 15: Relative packet throughput for a single VM

7.4 Optimal Affinitization with a Single VM

The best overall network performance for a single task on one VM is achieved when each individual task and the network interrupt process runs on different cores on the same CPU socket.

The following graph shows the improvement when all tasks involved in processing (IRQ handling on host, QEMU threads, vhost thread, IRQ handling in VM, netperf in VM) are affinitized to different cores on the same CPU socket. Results shown have been normalized to the non-affinitized performance in each case.

This strategy is not optimal for a "real" solution since it cannot be effectively applied to multiple VMs and does not scale since all NIC interrupt processing is handled by one core.

7.5 Optimal Affinitization with Multiple VMs

To achieve good performance for a realistic application with multiple VMs, you must:

1. Find a combination of VM affinitization rules that could be applied to a general system and applied to a moderate number of independent VMs running on the system to optimize network performance.
2. The task scheduler should have as much freedom as possible to balance the processing across the cores.
3. Network Interrupt processing should scale to allow multiple cores to handle the network RX and TX SWISR processing.

7.5.1 QEMU and vhost Affinitization

For each VM, affinitize QEMU tasks, Virtual Processor QEMU threads and vhost tasks to all of the first logical cores (of hyper-threaded physical cores) of the same physical CPU. Multiple VMs should be evenly distributed between the two physical processors. For instance on a 2-way 8-core Sandy Bridge system configure VM1 to run on cores [0-7] and VM2 to run on cores [8-15].

7.5.2 IRQ Affinitization

Affinitize the ixgbe driver interrupt processing to all of the second logical cores (of hyper-threaded physical cores). For instance, configure interrupts to be handled by cores [16-31].

Ideally all network interrupt processing should be on the same physical CPU as vhost, but this would make it difficult to map VMs across physical CPUs. The poorest interrupt related performance will occur when the NIC RX queue is being processed by a core on a different physical CPU than that which the vhost is running on. This will occur about 50% of the time when Flow Director is not being used to direct the flows appropriately.

7.5.3 VM Affinitization

Affinitize the VM virtio driver, RX interrupt task, TX interrupt task to a different virtual processor than where the network application runs (to use an idle core). Of course this will have lower network performance when two network applications run in a VM with only two virtual processors (i.e. no idle cores). It was also found that the un-affinitized operation works better for receiving TCP, since receive is improved when the processor used for RX interrupt routine is the same as the user application processor reading the socket queue.

Figure 16 compares network performance with the following schemes:

1. QEMU only affinitization:
   - **Host**: QEMU on core [8-15], IRQ services on cores [0-31], vhost on cores [0-31]
   - **VM**: no affinitization
2. General affinity scheme (QEMU, vhost, IRQ affinitization, VM affinitization):
   - **Host**: QEMU on cores [8-15], IRQ on cores [16-31] and vhost on cores [8-15]
   - **VM**: Netperf on vcpu 0 and IRQ on vcpu 1
3. **Partial affinity** (QEMU, vhost, IRQ affinitization, no VM affinitization) :
   - **Host**: same as with general affinity
   - **VM**: no affinitization

Throughput of TCP to and from the VM was limited by the network interface speed in the General Affinity case. The TCP throughput increase from the VM was small since the baseline throughput was already approximately 8 Gb/S.

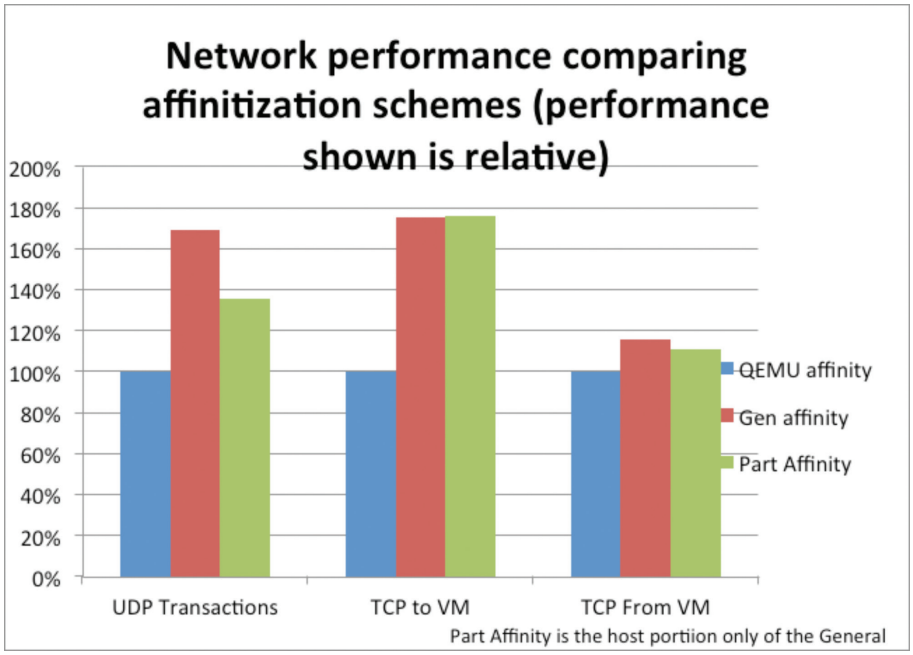


Figure16: VM network performance comparing affinity schemes

7.6 Alternative Affinitization Schemes

For heavily loaded systems the general affinitization scheme proposed above might not be optimal, as the second logical cores (of hyper-threaded physical cores) are only handling the NIC interrupt routines and may be lightly loaded. Hence total system performance may not be at maximum. Depending on the work load other rule sets might give better results.

For instance, another affinitization scheme allows all of the hyper-threaded cores to be used while maintaining the vhost to network HWISR and SWISR to vhost hyper-threaded pair relationship.

This mainly applies to use-cases where TCP traffic is predominant, as it's based on flow director.

It might cause higher interactions between the performances of different VMs, as there is an impact when a process is running on a CPU core that has its hyper-threaded core pair in use. This is probably not an issue on hyper-threaded core aware operating system task schedulers, but will have a performance transition when all of the core pairs have one task and more tasks need to run. However, when the number of tasks ready to run or

running exceeds the hyper-threaded core count, the total system performance should exceed the first affinitization rule set.

7.6.1 QEMU and vhost Affinitization

For each VM affinitize QEMU tasks, Virtual Processor QEMU threads, and vhost tasks to all logical cores (of hyper-threaded physical cores) of one physical CPU. Multiple VMs should be evenly distributed between the two physical processors. For instance, configure VM1 to run on core [0-7 and16-23] and VM2 to run on cores [8-15 and 24-31].

7.6.2 IRQ Affinitization

Affinitize the ixgbe driver interrupt processing to the second logical cores (of hyper-threaded physical cores) associated to the Niantic queue number. For example NIC queues 0 to 15 would be processed by second logical cores (of hyper-threaded physical cores) 16 to 31 and the NIC queues 16 to 31 would be processed by the first logical cores (of hyper-threaded physical cores) 0 to 15.

7.6.3 VM affinitization

Affinitize the VM virtio driver, RX interrupt task, TX interrupt task as in the general affinitization rule set.

8 Throughput Performance Considerations

When looking at the Crystal Forest architecture, we see that the two 10GBE fabric interfaces are connected through the Intel 82599 10 Gigabit Ethernet Controller chips to socket 0, and the two front 10GBE interfaces (with RJ-45) are connected to socket 1. We investigate how this asymmetrical design can influence performance.

8.1 CPU Socket

There may be a difference in performance when using cores from socket 0 instead of cores from socket 1 for a packet processing workload.

In this test 32 UDP streams are initiated by the Spirent test generator on two NIC interfaces. The SUT is configured to forward frames received on NIC interface 1 to NIC interface 2, and frames received on NIC interface 2 to NIC interface 1. This means there are two 10Gbps streams entering the platform, and two streams leaving the platform.

Four cases are tested with 2 cores configured to handle traffic from interface 0, and two other interfaces configured to handle traffic from interface 1.

1. The 4 cores are located on socket 0.
2. The cores handling interface 0 are located on socket 0. The cores handling interface 1 are located on socket 1.
3. Interface 0 and interface 1 are each handled by one core of socket 0 and one core of socket 1.
4. The 4 cores are located on socket 1.

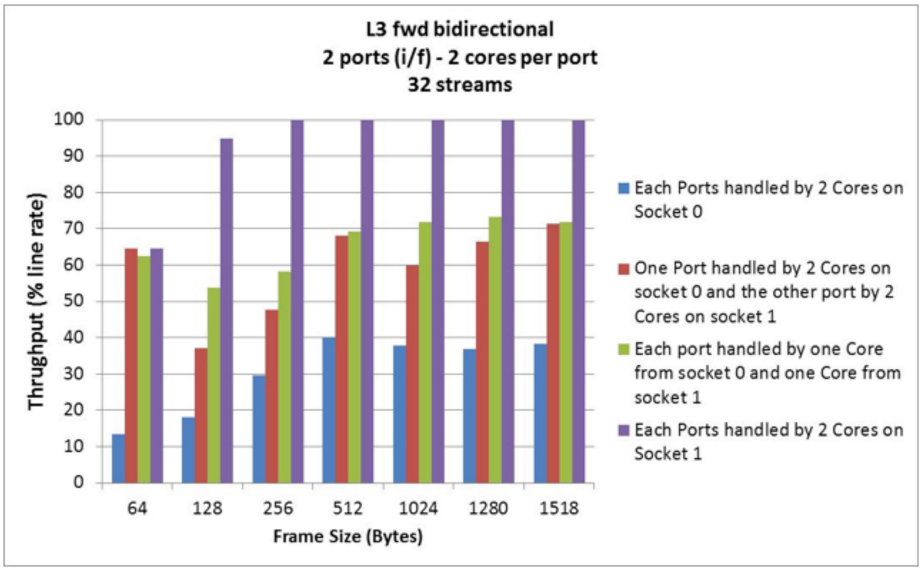


Figure 17: Impact of CPU socket choice on packet throughput performance

When looking at the Crystal Forest architecture, we see that the two 10GBE fabric interfaces are connected through the Intel 82599 10 Gigabit Ethernet Controller chips to socket 0, and the two front 10GBE interfaces (with RJ-45) are connected to socket 1. This means that if a packet arriving on an external RJ-45 interface is handled by a core on socket 0, it will have to cross the QPI link. The results show the effect of this and illustrates that using cores on different CPU sockets can have a dramatic impact on performance.

8.2 CPU Utilization and RSC, LRO, GRO

CPU utilization might be influenced by hardware acceleration techniques such as RSC (Receive Side Coalescing), LRO (Large Receive Offload) and GRO (generic Receive Offload). To test this we measure CPU utilization when receiving a TCP stream using RSC, LRO, and GRO.

One 10Gbps TCP stream is sent by a system (running a netperf TCP_STREAM test) towards the SUT (running netserver). In all cases, the SUT can receive all packets but depending on the acceleration technique being used, the CPU usage varies.

TCP Segmentation Offload (TSO)

TSO is when a TX buffer much larger than the MTU is passed by the Linux stack to the NIC. The work of dividing the large packet into sizes smaller or equal to the MTU is offloaded to the NIC. Using this technique can drastically reduce the CPU usage for a given load, or increases the throughput. TSO is usually enabled on Intel Niantic cards by default.

Receive Offloads

This is similar to TSO but in the other direction. When incoming packets are received by the NIC, they are merged before being sent to the Linux stack. This merging can be done in the NIC itself (in which case, it's called **Receive Side Coalescing**) or in the driver (in which case it's called **Large Receive Offload**).

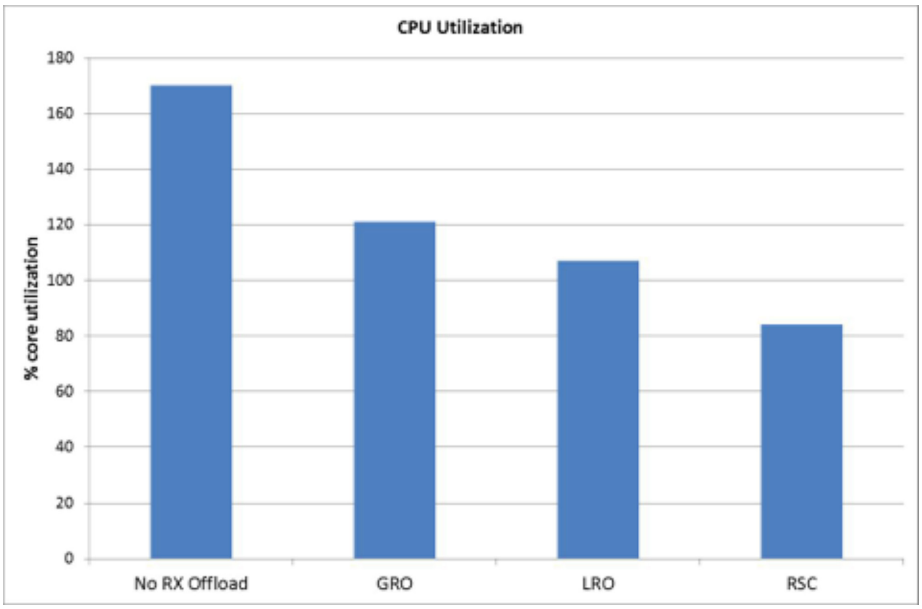


Figure 18: CPU utilization handling comparing hardware acceleration techniques (single TCP stream at 10Gbps)

Large Receive Offload (LRO)

LRO is a technique for increasing inbound throughput of high-bandwidth network connections by reducing CPU overhead. It works by aggregating multiple incoming packets from a single stream into a larger buffer before they are passed higher up the networking stack, thus reducing the number of packets that have to be processed. LRO combines multiple Ethernet frames into a single receive, thereby potentially decreasing CPU utilization for receives. LRO cannot be used when bridging or routing packets. LRO is also incompatible with iSCSI. LRO can be disabled by recompiling the ixgbe driver with the proper flags (IXGBE_NO_LRO), or, runtime, using ethtool -K ethX lro off. Statistics can be collected using ethtool -S ethX | grep lro.

Receive Side Coalescing (RSC)

Hardware based coalescing can merge multiple frames from the same IPv4 TCP/IP flow into a single structure that can span one or more descriptors and works similarly to the software large receive offload technique.

By default RSC is enabled for 82599-based adapters and LRO is disabled. RSC can be disabled by recompiling the ixgbe driver with the proper flags (in which case LRO might be used). RSC is also incompatible with bridging or routing packets and iSCSI. Both RSC and LRO can be disabled runtime using ethtool ethX -k lro off.

Both LRO and RSC are automatically disabled when bridging is used, which is the case when using Virtual machines (this is done automatically by the kernel when brctl addif command is launched).

Generic Receive Offload (GRO)

In-kernel software implementation of GRO is supported by Niantic drivers. It has been shown that by coalescing RX traffic into larger chunks of data, CPU utilization

can be significantly reduced under large Rx load. GRO is an evolution of LRO. GRO is able to coalesce other protocols beside TCP.

Documentation about GRO is conflicting: ixgbe.txt from 3.7.21 ixgbe driver specifies that GRO should be disabled when using bridging/routing. Ixgbe.txt from Linux kernel specified that it's safe to use GRO with configurations that are problematic with LRO, namely bridging and iSCSI.

GRO can be disabled at runtime using ethtool –K ethX gro off.

When not using bridging, RSC uses less CPU to handle 10Gb TCP stream than LRO, and LRO uses less CPU than GRO. GRO can still reduce CPU usage by ~30% compared to not using an offload mechanism.

8.3 Latency Considerations

Latency is an important consideration packet processing. Below we show results from tests forwarding application running in a VM, forwarding application running on the host, forwarding by the host Linux kernel, and forwarding using Intel DPDK (user-space) application. The VM and host application are identical, one running in a virtual system and one running on the host. This application is written using the Linux socket API and runs in user space. The VM uses vhost-net. Host kernel data refers to L3 forwarding done by the Linux kernel (/sys/net/ipv4/ip_forward).

The test results below are with 32 UDP streams (of 64 bytes frames). The SUT forwards packets using different configurations and latency is measured.

When using Intel DPDK, packets are transmitted by bursts, either when an internal buffer is full, or when a timer (around 140 microseconds at 1.4GHz) has elapsed. Intel DPDK latency is higher below 100 Mbps (around 80 microseconds). This can be reduced by decreasing the BURST_TX_DRAIN parameter to approximately 15 microseconds. This however has a negative effect on the maximum achievable throughput which reduces approximately 10 percent to 15 percent.

Applications running in a virtual machine have much higher latency (around three times higher) that compared to running on the host (or hypervisor). Using SR-IOV and Intel DPDK in the VM should be investigated as a way of reducing this VM latency. Not surprisingly, forwarding by the kernel achieves lower latency than from user space.

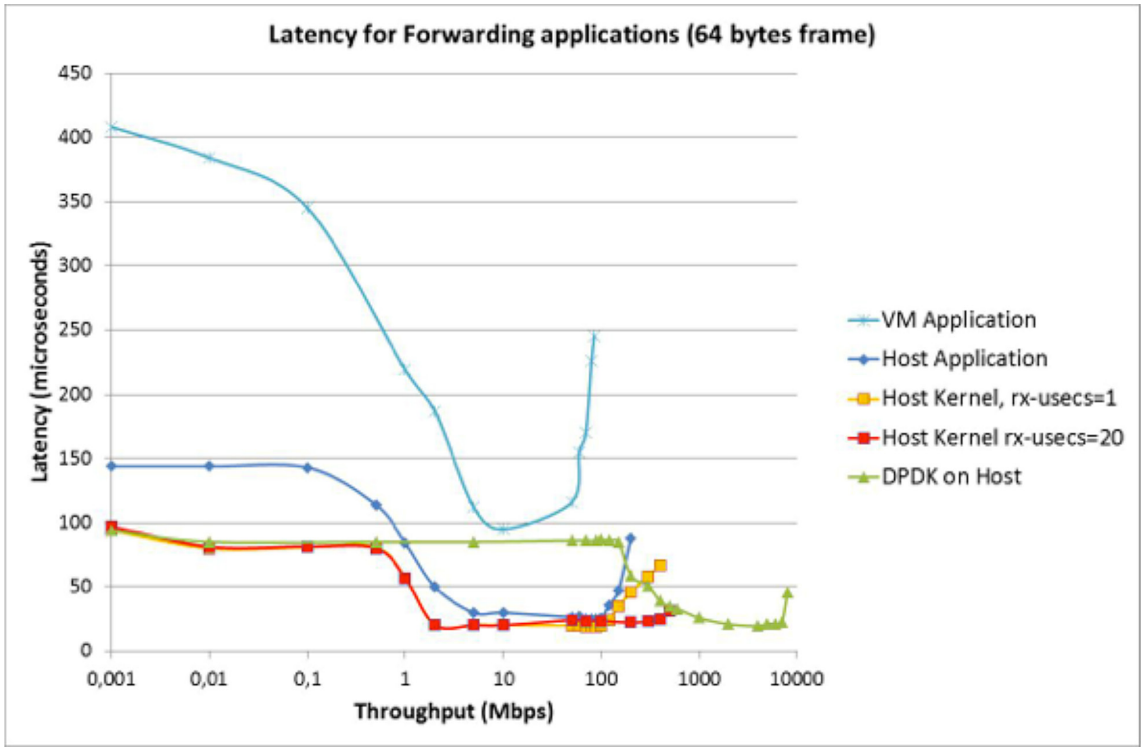


Figure 19: Comparing latency of different forwarding configurations

8.4 VM to VM Communication

Consider a model where each VM represents a network service and each service is handled in a pipeline. In this experiment we measure throughput of a user space application running in a virtual machine sending data to a user space application running in another VM. Three different network virtualization schemes are compared: virtio, vhost-net and SR-IOV. When using vhost-net or virtio, the packets are sent from one VM to the other VM through a Linux bridge. When using SR-IOV, the packets are sent and received through the virtual functions of the NIC. Those packets are then transferred between the VFs using the internal switch (L2 loopback) on the physical NIC.

Only one stream of UDP traffic is generated by netperf in one VM. Netserver is running in the other VM. Each VM is configured with two virtual CPUs.

The performance of VM to VM communication remains poor (400 kpps represent around 50 percent of line rate with 1518 bytes frames, and 3 percent with 64 bytes frames).

None of the three techniques measured (virtio, vhost, SR-IOV) are really scalable today. Vhost and virtio are limited by one single threaded task (vhost-net kernel task in the vhost test). SR-IOV is not compatible with RSS so a virtual NIC in a VM using SR-IOV can only use one queue. So, the only way to be scalable today would be to use multiple VFs and hence multiple IP addresses in each VM.

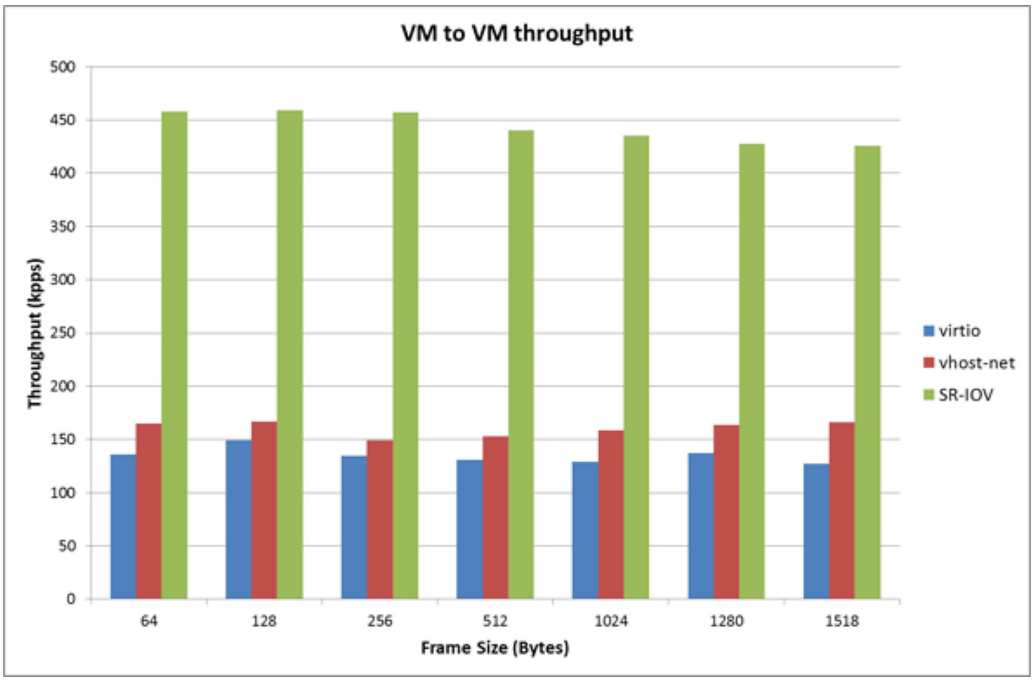


Figure 20: VM to VM Packet Throughput

9 Appendix A: VM Network Tasks

A VM running on a host system doing network communications has processes and network interrupts associated with them. Some of the processes and interrupts are directly related and directly controlled by the host operating system while a portion of the processing resources of the VM are controlled internal to the VM, which is indirectly controlled by the host processes.

The bidirectional network processing can be broken down into different device driver interrupt processing and schedulable tasks to process the network data. There are different operation modes for some of the communications components, but vhost will be discussed. Figure 21 can be broken down in two parts:

- Tasks or processing components (in yellow)
- Passive components: the processing components call these and do the work of pushing these through the passive components.

There are other processing components, not shown, that come into play to keep the stack operating under all conditions but are generally not used most of the time. For example, if the NIC TX queue gets full, data buffers will get queued on the task's socket queue, requiring a kernel task to push the buffers through at a later time. Overall this is a very simplified discussion of the components.

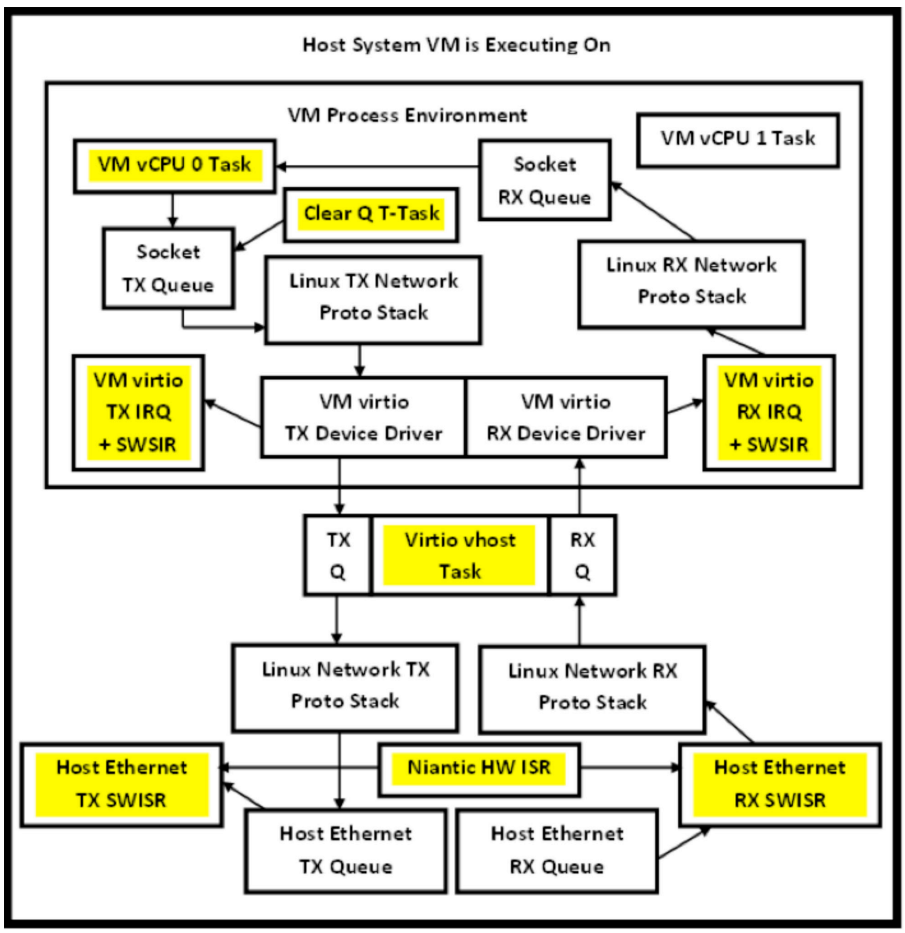


Figure 21: VM and Network Process Task Set

9.1 Host Ethernet Device Driver RX Queue

The Intel 82599 10 Gigabit Ethernet Controller is enabled and a network RX Queue for each CPU core is loaded with empty SKB network buffers. It can read and write system memory independent of the system CPUs and step through the empty buffer list in memory and query buffers status, fill with network data, and set completion status and any error codes for each buffer. When some RX buffers are completed and after period of time as set by the interrupt rate settings, the CPU will be interrupted by the Intel 82599 10 Gigabit Ethernet Controller to process the RX and/or TX queue set.

9.2 Host Ethernet RX IRQ and SWISR

The CPU core with the RX hardware interrupt affinitized to the RX queue will execute the hardware ISR. The hardware interrupt processing will set the SWISR associated to do NAPI operation for the queue.

On exit of the hardware interrupt, if the CPU core is not already executing the SWISR for the core, the CPU enters the SWISR processing and if no other SWISR processes with a higher priority are pending, processes the NAPI SWISR for the Intel 82599 10 Gigabit Ethernet Controller TX and RX queue set.

The NAPI interrupt process first clears any empty buffers from the associated TX queue to make any empty buffers available. A filled RX SKB buffer is removed from the receive queue and the buffer is pushed into the protocol stack by NAPI. After returning from the protocol stack receive operation, the device driver allocates an empty SKB buffer and puts on the receive queue to keep new buffers available to the Intel 82599 10 Gigabit Ethernet Controller to fill.

9.3 Host Linux Network RX Protocol Stack

The NAPI SWISR protocol stack processing does some packet checks and pushes the buffer into a queue at the bottom of the protocol stack if the buffer can be associated to a series of network packets already on the queue. If the series of packets becomes complete or there is another reason, the series of packets are pushed through the protocol stack. If the new packet is not part of the current pending linked series, the pending series is pushed through the protocol stack and then the new packet is pushed through. Often there is no pending series or the packet has no association to a possible packet series, in which case the single packet is then immediately pushed through the protocol stack. For this case, the packet is pushed to the virtio vhost receive queue (educated guess of queue existence) where it is queued and the vhost kernel task is woken up if not already running. This is a very simplified explanation, but the protocol stack code handles all operations and combinations of possible operations and this need to adapt to different combinations adds overhead to every packet or set of packets going through the protocol stack.

For the non-vhost operation, the SWISR and/or virtio device driver SWISR in the VM completes the host to VM translation, doing the host side of moving the data packet into the VM memory.

9.4 Host virtio vhost RX Queue and RX Task

The virtio vhost is a kernel task scheduled by the task scheduler. When network packets are queued by the host's protocol stack, vhost is woken up to process the packet queue to move data to the VM. The whole function of the vhost RX queue processing is to translate SKB packets in memory owned by the host into SKB packets in memory owned by the VM virtio network device driver and placed on a queue, similar to the network packet queue written to by the Intel 82599 10 Gigabit Ethernet Controller. There are several ways for this translation to occur, however a vhost task is being used to do this translation and currently the vhost task appears to uses a moderate amount of processing cycles to do the host to VM packet translation operation.

The full cost of the packet translation may not be completely done by the vhost task since there is the protocol stack side that is writing the input queue on the host and there is the virtio network device driver that is removing packets from its receive queue. Either of these operations can be doing some form of the processing that adds to the CPU cost of packet translation from the host into the VM.

9.5 VM virtio vhost Network Device Driver RX Queue

The vhost task adds the newly translated SKB packets to the VM's virtio device driver queue. The VM's hardware ISR associated to virtio queue can be posted to wake up the VM hardware interrupt processing to handle the new RX packet.

The VM's hardware interrupt processing is probably designed to only wake when transitioning from an empty queue and possibly use delayed queue startups to prevent continuous hardware interrupt processing in the VM for each packet, to prevent trashing, which happens when setting the physical network interrupt time too small of a time interval and which case network throughput rate notably drops when heavy network traffic is occurring caused from the cost of a higher number of ISR and SWISR entry and exit operations per second.

9.6 VM virtio vhost Network RX IRQ and SWISR

The Linux VM's virtio network receive device driver probably executes as a

NAPI device driver similar to the Host network device driver. In this case, vhost would cause a hardware interrupt for the network interface to be posted to the VM to start the receive processing. The VM would execute the virtio Hardware Interrupt, which would post a NAPI SWISR in the VM's operating system.

9.7 VM Linux Network RX Protocol Stack

The NAPI SWISR operation would do packet receive processing by taking SKB buffers off the receive queue and to push the packet through the protocol stack in the same manner as discussed in the host, to the socket receive queue of the task to receive the packet and wake the VM's communications task to process the new received packet if not already running.

Before the socket queue can be written to, the CPU core needs to get SMP ownership of the semaphore which is protecting the queue. The taking of the semaphore from the previous CPU core takes a time, and can only be done when the other CPU is no longer using the semaphore as in the case of reading from the queue. The ownership change stalls the new CPU core until ownership has been fully obtained, which can be a moderate number of clock cycles even when the original owner is idle and is a local core. When the previous owner is on a different physical processor, the stalling is substantially increased by the cache to cache synchronization between the physical processors.

The use of growing lists of packets for a series of packets in the bottom of the protocol stack still can be used, especially in the case of when large packets used in TCP operation.

9.8 VM Linux Network User Task

The user task wakes up and removes the data from the socket receive queue. If the CPU core which put the packet into the queue is different than the CPU core running the task that needs to remove the data, a SMP synchronization event must occur, generally stalling the CPU core until ownership is obtained from the previous CPU core.

Often several packet writes or several packet reads occur to a socket queue between reading and writing the queue from the other side, lowers the average socket SMP synchronization cost. However since a moderate amount of processing time is needed to push a packet through the protocol stack a constant state of pacing can occur, where every packet causes a socket synchronization event and also cause a task wakeup operation then followed by another socket synchronization event by the read CPU, which all are costly operations that combined can sometimes add more clock cycles to the packet overhead cost than the CPU clock cost of the push of the packet through the protocol stack to the receiving end.

9.9 VM Linux Network TX Protocol Stack

Not all network communications have a TX operation for a receive operation, except for a transaction. A VM TX operation can be viewed as independent operation from network RX operations since the code follow independent code paths with independent queues and semaphores, having very little CPU contention except for use of the CPU core.

Data is passed from the user task to the applications network socket. The queue within the socket is a linked list of SKB buffers, and the socket buffer size is artificially maintained since the individual buffers are not always full, requiring more physical memory then the set socket buffer size. For TCP, a large data block can be written which can easily queue up multiple SKB buffers, taking advantage of reduced ownership cost of the socket semaphore and SKB memory manager semaphores since the probably of a competing CPU core goes down for very short time periods between buffers. Also all SKBs can be completely filled with TCP, which must have a SKB for each packet, except in the case of large UDP packets which is similar to TCP where a series of SKB buffers are allocated for each packet more efficiently.

Discussion of the UDP packet is easier since the TCP data flow algorithms don't apply. If the protocol stack is not stalled (or paused), once the data has be copied to the packet and completed, if the socket packet queue is empty, the packet is pushed down by the user task processing to the protocol stack into the device driver, which for the VM is the virtio network device driver.

9.10 VM Linux virtio TX Device Driver

VM network packets coming into the VM's virtio Device Driver are queued for TX operation. Some preparation processing might be done to prepare for the packet for translation from the VM's memory to the Host's memory. The virtio vhost task is started if not already running.

If the TX queue is full, the attempt to put the SKB to the virtio device driver queue causes an unsuccessful operation and the protocol stack still owns the SKB buffer. The subroutines return back to the socket queue and TX buffer are then put on the TX queue, and a timer task is started which will attempt a push of the packets later time since the user task cannot be depended on doing a network operation at a later time to restart data flow.

9.11 VM Linux virtio TX IRQ, SWISR

The network TX interrupt is not to start communications, although the protocol stack has mechanisms to restart the network flow when stopped, but to clear the buffers from the device driver that has completed sending. Once the

SKB buffer has been removed from the network TX queue, if completion/release subroutine is present, the subroutine is called, and the buffer release subroutine on also the buffer returns buffer back to the memory manager. The reason the buffer has the free subroutine is to allow multiple memory pools, and the subroutine directs it to the correct memory pool. Releasing a list of empty buffers is more efficient since once ownership of the memory semaphore is obtained the next ownership costs very little if no other task gets ownership, which is more probable in short time periods. Also the last buffer link list memory will probably be in local cache, preventing the need to wait to get the link information out of memory.

There are actually two separate buffers to an SKB buffer, a management buffer and a packet buffer, which allows multiple management buffers to own and/or transfer a packet buffer, which saves on copying the packet buffer. So to buffer allocations from separate managers occur for allocation of an SKB buffer and two memory free operations occur to put the buffers back onto associated free lists. This type of design tradeoff of flexibility which improves some types of network performance but has some impacts on the basic protocol stack performance.

When the network is also receiving data, processing of the receive packets often clears the network TX buffers from the TX queue since there is a need for empty buffers to put on the receive queue. Combining operations that group semaphore and hot caches for the buffer operations gives better performance. If no network data is being received, then the TX interrupt service routine clears the buffers.

9.12 Host virtio vhost TX Queue and TX Task

The virtio vhost task is started when output buffers are placed on the VMs output queue. The vhost takes the packets off the queue and translates the buffer from the VM system SKB memory to the host's SKB memory and buffers. Once translated, vhost task pushes the packet into the hosts TX protocol stack, which pushes the packet to the Ethernet device driver output queue. If there is not room on the TX queue, a status is returned indicating that the task, vhost in this case, needs to wait some time before attempting to do another TX push operation into the network stack.

9.13 Host Linux Network TX Protocol Stack

The protocol stack does some utility operations, sets checksums if needed and routes the packet to the correct network device driver including correct queue on the device driver. The protocol stack is passive with the work is being done by the running task, which is doing the work of pushing the packet into the protocol stack. The push operation also starts the TX NIC output if needed.

In this case both the host and VM is Linux so the protocol stack is the same (or very similar for a different kernel version).

9.14 Host Ethernet TX Queue

Once on the host network TX queue the NIC transmits the packet on the physical interface according to the rules of the physical interface and marks the buffer completion status when done. Operation is similar to the VM TX queue operation in this example.

9.15 Host Ethernet TX IRQ and SWISR

The TX interrupt clears the empty buffers, similar to as described on the VM's TX interrupt service routine.

10 Appendix B: Test Setup Notes

10.1 Crystal Forest Notes

Crystal Forest has dual 8 core, 1.4 GHz CPUs, 20480KB Cache, with Hyper-threading enabled, and 16 GB ram installed.

CentOS 6-2 installed, with linux-3.2.2 kernel compiled using CentOS 6-2 configuration, with Receive Packet Steering (RPS) disabled in the compilation (requires net configuration file modification and then disable in the configuration). The firewall is disabled, several unneeded daemons shutdown, and irqbalance disabled.

The IXGBE device driver is ixgbe-3.7.21 release. In the single VM affinitization test, the network IRQ processing was affinitized to a single core in the same physical processor as the rest of the VM related tasks..

The Crystal Forest CentOS system was set to boot with a serial port as the console, and uses Runlevel 3 which is text mode for console (not X-windows which is Runlevel 5). Also the grub was set to use the serial port for selecting the OS kernel to boot.

10.2 CentOS-6.2 KVM Virtual Machine

The VM is a CentOS 6.2 with 3.2.2 kernel. The VM was started with 2 virtual CPUs, 1 GB of RAM, 2 networks interfaces, and using virtio with vhost.

11 Appendix C: Host and VM Affinitization in Linux with Examples

For the interested reader, more detailed VM affinitization notes follow with examples included on how to review and do the affinitization operations:

In order to effectively affinitize a host or VM processing, the user needs to understand how to do the affinitization. This section describes the methods used to affinitized tasks and interrupts on both the host as well as in the VM.

The CPU core affinitization determines what CPU core set can be used to execute a program and/or which core set can be used to do interrupt processing. The CPU core set is one or more CPU cores of the available processors. The relationship of the CPU cores in a system is dependent on the processor architecture and board layout.

Often the relationship of the CPU cores used for processing can give better process performance for some programs and interrupt processing then when other CPU core sets are used. The process of setting a certain CPU core set to do the processing is referred to as CPU Affinitization.

Using tests of the programs and controlling the CPU cores used for the program's processes, the relationship of the CPU core usage and performance can be determined. Ideally a random set of CPU cores would give the same performance compared to other CPU core sets, but this is not usually the case due to SMP synchronization. The synchronization delay cost is more when cores are on different physical sockets compared to cores within the same physical CPU. The lowest SMP core to core synchronization cost is between a hyper-threaded core pair. Considerable improvements in hardware and operating systems have reduced the costs of SMP synchronization delay time and number of SMP synchronization points for a multi-core and multi physical processor systems over the older operating systems and older processors, however not all SMP costs have been eliminated and still play a critical role in optimizing system performance.

11.1 Common CPU Core Organizations

The relationship of the CPU cores in a system is processor dependent (e.g. Sandy Bridge vs Westmere) and can be affected by the particular layout of the system motherboard. A system can have one or more physical socketed processors which have one or more CPU cores within each physical CPU, and is referred to by the physical socket ID.

The order of CPU cores is system dependent. However when running Linux the organization of the CPU cores and the physical CPUs is relation to the Linux operating system is available. By reviewing the data displayed by the Linux command **cat /proc/cpuinfo** for each of the processor cores, the core ordering in the system can be determined.

```
cat /proc/cpuinfo
processor       : 31
vendor_id       : GenuineIntel
cpu family      : 6
model           : 45
model name      : Intel(R)
                  Xeon(R)CPU
                  E5-2680 0 @
                  2.70GHz
stepping        : 6
microcode       : 0x60d
cpu MHz         : 1200.000
cache size      : 20480 KB
physical id     : 1
siblings        : 16
core id         : 7
cpu cores       : 8
apicid          : 47
initial apicid  : 47
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
```

This last CPU information record shows the information for the last CPU information record for a system with 2 physical processors (count starts with 0 and 1 is last physical id), each having 8 cores, and with hyper threading enabled for a total of 32 schedulable processor cores of 0 through 31. In the data, physical id refers to the physical processor starting with 0, the core id refers to the core of 0 to n, which is 7 in this case for the last core of the processor, and the apicid is even for the first hyper threaded portion of the core pair, while the odd apicid value refers to the second hyper threaded portion of the CPU core pair. By reviewing these numbers through the cpuinfo data set records, the relationship of the CPU cores to physical processors of the system can be determined.

There are some common CPU core organizations recommended by Intel, but these need to be verified for a particular system since the CPU core order is set by motherboard layout. Knowing the core organization to sockets and hyper-threaded core pair sets are important when affinitizing tasks and/or interrupts.

11.1.1 Common Intel Xeon Processor E5 Series Dual 8 Core Physical Processors Organization

An Intel Xeon Processor E5 series-based system with two 8 core physical CPUs is commonly organized with CPU cores as follows:

```
Physical CPU 0 cores:  0, 1, 2, 3, 4, 5, 6, 7.
Physical CPU 1 cores:  8, 9, 10, 11, 12, 13, 14, 15.
```

When hyper threading is enabled, the additional cores are:

```
Physical CPU 0 hyper-threaded cores of 0, 1, 2, 3, 4, 5, 6, 7: is 16, 17, 18, 19, 20, 21, 22, 23.
Physical CPU 1 hyper-threaded cores of 8, 9, 10, 11, 12, 13, 14, 15: is 24, 25, 26, 27, 28, 29, 30,
31.
```

The CPU core order is different order than earlier system core organizations since related programs have less SMP overhead when cores used are on the same physical processor. The search for a core to execute on tends to be sequential, and other factors such as power usage have influenced the change of core id order.

11.1.2 Intel® Xeon® Processor 5600 Series Dual 6 Core Physical Processors

An Intel ®Xeon® Processor 5600 Series-based system with two 6 core physical CPUs is commonly organized with CPU cores as follows:

```
Physical CPU 0 cores:  0, 2, 4, 6, 8, 10.
Physical CPU 1 cores:  1, 3, 5, 7, 9, 11.
```

When hyper threading is enabled, the additional cores are:

```
Physical CPU 0 hyper-threaded cores of 0, 2, 4, 6, 8, 10: is 12, 14, 16, 18, 20, 22.
Physical CPU 1 hyper-threaded cores of 1, 3, 5, 7, 9, 11: is 13, 15, 17, 19, 21, 23.
```

11.1.3 Intel Xeon Processor 5600 Series and Intel® Xeon® Processor 5500 Series Dual 4 Core Physical Processors

An Intel Xeon Processor 5600 Series or Intel® Xeon® Processor 5500 Series-based or system with two 4 core physical CPUs is commonly organized with CPU cores as follows:

```
Physical CPU 0 cores:  0, 2, 4, 6.
Physical CPU 1 cores:  1, 3, 5, 7.
```

When hyper threading is enabled, the additional cores are:

```
Physical CPU 0 hyper-threaded cores of 0, 2, 4, 6: is 8, 10, 12, 14.
Physical CPU 1 hyper-threaded cores of 1, 3, 5, 7: is 9, 11, 13, 15.
```

11.2 Task CPU Core Affinitization with Taskset

Task, program, or process affinitization to the CPU cores is done with the taskset utility either during task creation or after task creation using the task's process id. The utility taskset sets or changes a bit mask of the CPU cores which the task can be scheduled to run on. Without using taskset, the CPU cores to run a program on is generally set to the same CPU core set that the program that is being used to start the program, with some exceptions. When starting a program with a shell, the CPU core mask is set to all the available CPU cores in the system.

11.2.1 Displaying a Process's CPU Core Affinitization

The taskset utility can also be used to display the current CPU core set that the program can run on by using the –p command line flag and current process id. For example, netserver process is started and running, it's process id can be found using ps, and then taskset can be used to display the current CPU core mask, in hex format, that the netserver process can be scheduled run on.

The example netserver task is started where it can run on any of the system's CPU cores:

```
[~]# netserver

Starting netserver with host 'IN(6)ADDR _ ANY' port '12865' and family AF _ UNSPEC
```

Find the core mask for the task, which is netserver for this case:

```
[~]# ps -e |grep netserver | awk ' { print $1 }' xargs -n1 taskset -p
pid 6158's current affinity mask: ffffffff
```

When getting the CPU core mask, the process id directly follows the -p command line flag. In this case the program can be scheduled to run on any of the 32 cores since all 32 bits of the 32 bit mask are set.

11.2.2 Setting Task Affinitization during Task Startup

The CPU core affinitization utility, taskset, can be used in the same command line or script by placing taskset command just before the program execute command, listing the CPU cores in a list or using a bit mask.

11.2.2.1 Starting a Program with a CPU Core List

The taskset utility can be used with a comma separated list of decimal CPU core ids following a -c command line flag:

```
[~]# taskset -c 1,2,3,9 netserver
Starting netserver with host 'IN(6)ADDR _ ANY' port '12865' and family AF _ UNSPEC
```

Find the core mask for the task:

```
ps -e |grep netserver | awk ' { print $1 }' xargs -n1 taskset -p
pid 6289's current affinity mask: 20e
```

11.2.2.2 Starting a Program with a CPU Core Mask

The taskset utility can also be used with a hexadecimal bit mask of CPU cores following the taskset command:

```
[~]# taskset 0ad05 netserver
Starting netserver with host 'IN(6)ADDR _ ANY' port '12865' and family AF _ UNSPEC
```

If the mask does not start with a number character, a zero needs to be placed in front of the first hex digit of the bit mask number. For the mask, CPU 0 is represented by bit 0 of mask or 000001. Each successive CPU core is the next bit higher in the mask. The CPU core mask bit can also be represented by the number 1 shifted up by the CPU core number. For example CPU core 4 = (1 << 4) = 10h.

11.2.3 Changing Task Affinitization after Task Startup, using the Process ID

After the program has been started, the CPU core set that the process can be run on can be changed. For example starting netserver:

```
[~]# netserver
Starting netserver with host 'IN(6)ADDR _ ANY' port '12865' and family AF _ UNSPEC
```

Find the process id for the task, which is netserver in this case

```
[~]# ps -e |grep netserver
6838 ?        00:00:00 netserver
```

The CPU cores can be changed with a hexadecimal bitmask:

```
[~]# taskset -p 0ad055 6838
pid 6838's current affinity mask: ffffffff
pid 6838's new affinity mask: ad055
```

A list of cores by decimal number rather than a hexadecimal mask can be used with taskset when using the -c command line argument:

```
[~]# taskset -p -c 2,3,7,14,29 6838
pid 6838's current affinity list: 0,2,4,6,12,14,15,17,19
pid 6838's new affinity list: 2,3,7,14,29
```

11.3 Interrupt Processing Affinitization

The setting of CPU Core affinity for interrupt processing is done differently than for the scheduled tasks and processes. The interrupt core affinitization assigns a particular processor core or set of processor cores to do the interrupt processing for a given interrupt request number. The processor that does hardware interrupt processing (HWISR) usually does the associated software interrupt processing (SWISR), with the SWISR flag generally being set by the HWISR.

Many interfaces which require a moderate amount of processing once interrupted as in the case of a network interface has a layered interrupt processing. First the hardware interrupt generally does very fast acknowledgement of an interrupt and then sets the associated software interrupt flag and starts to exit. If a SWISR flag is pending and is not currently running, the SWISR starts executing and the hardware interrupt is re-enabled. Otherwise the hardware interrupt fully exits back into the currently running software executing before the HWISR, which can be the SWISR or other task. Only one software interrupt can running for the CPU core and does the long processing periods while the hardware interrupts are enabled, which can generate more work for the Software Interrupt before it exits.

The Niantic NIC supports multiple receive and transmit queue pairs which are assigned a unique interrupt request number for each NIC queue. For each interrupt number, the CPU core affinity for interrupt processing can be set. Reviewing the interrupt default affinity can be done by looking in the interrupt related directory, /proc/irq/(irq number):

```
[~]# cat /proc/interrupts | grep eth0
 125: … eth0-TxRx-0
 126: … eth0-TxRx-1

[~]# cat /proc/irq/125/smp _ affinity
00000000,ffffffff
```

The goal is to assign a different CPU core to different queues to distribute the NIC processing load across the CPU core set.

For example, the set_irq_affinity.sh script for the Niantic IXGBE device driver sets a unique CPU core for each of the unique RX and TX queue interrupt pairs:

```
[~]# ./set _ irq _ affinity.sh eth0
no rx vectors found on eth0
no tx vectors found on eth0
eth0 mask=1 for /proc/irq/113/smp _ affinity
eth0 mask=2 for /proc/irq/114/smp _ affinity
eth0 mask=4 for /proc/irq/115/smp _ affinity
eth0 mask=8 for /proc/irq/116/smp _ affinity
eth0 mask=10 for /proc/irq/117/smp _ affinity

eth0 mask=10000000 for /proc/irq/141/smp _ affinity
eth0 mask=20000000 for /proc/irq/142/smp _ affinity
eth0 mask=40000000 for /proc/irq/143/smp _ affinity
eth0 mask=80000000 for /proc/irq/144/smp _ affinity
```

In this case a bit mask for each of 32 NIC queue pairs, initialized with 32 queue pairs since there are 32 schedulable CPU cores, is sequentially assigned to a CPU core on a system that has 32 CPU cores. In the latest IXGBE device driver design, the RX and TX queue pairs are handled by one interrupt, but they could be separated into 2 interrupts, as in earlier IXGBE device drivers. For Network TX operations, the output packet is placed in the transmit queue numbered the same as the CPU core that is doing the transmit operation. The above mask assigns the interrupt processing to the same core as the TX output queue giving one queue TX and RX set per CPU core. However a different CPU core can be assigned to do the TX and RX interrupt processing for each queue pair.

The affinity can be set by a command line, setting the affinity mask by using an echo operation as follows:

```
[~]# echo 4 > /proc/irq/125/smp _ affinity
```

Reviewing the affinity mask, we see the CPU core 2 which is (1 << 2)= 4 mask, is set:

```
[~]# cat /proc/irq/125/smp _ affinity
00000000,00000004

[~]# cat /proc/irq/125/smp _ affinity _ list

2
```

11.3.1 Interrupt Balancer Server Daemon: irqbalance

The interrupt balance server daemon irqbalance, might disrupt the interrupt affinitization in some cases. There are different versions and have different characteristics. Experts for the Niantic have indicated that with the later IXGBE device driver and with the recent irqbalance, the affinitization by the IRQ balancer will result in target affinitization, but I lack actual details of which version and how this is done. However the IRQ balancer can sometimes change CPU core affinitization of the interrupts from what was manually set. The newer IRQ balancer has some algorithms built in to change interrupt handing to one physical CPU rather then spread across 2 physical CPUs when system loads are low to reduce system power usage.

To see if irqbalance is running

```
[~]# ps –e |grep irqb
4784 ?       00:00:00 irqbalance
```

Often turning off the interrupt balancer before setting affiliation is needed to prevent interrupts from being dynamically changed for the target affinitization. The irqbalance daemon can be disabled at boot up, but can also be terminated with the following command:

```
[~]# killall irqbalance
```

Leaving the interrupt balancer on when strict CPU interrupt affinitization is required, needs to be carefully studied for the particular case. During most tests requiring affinitization or repeatability, the IRQ balancer should be turned off before doing the affinitization.

To restart the IRQ balancer daemon (need to include the "on" to enable):

```
[~]#irqbalance on
```

11.4 Linux KVM QEMU Virtual Machine CPU Core Affinitization

Setting the CPU core affinitization involves affinitization of the VM tasks to the physical CPUs on the host and/or the affinitization of VM's interrupt processing of the virtual processors within the VM's guest operating system.

11.4.1 Affinitization of the QEMU Virtual Processors to the Host's Physical CPU Cores

When QEMU is started using KVM, the base QEMU program creates a processing thread on the host for each virtual processor of the guest operating system. These threads are independently scheduled on the host as separate tasks.

If vhost is enabled in the QEMU arguments during startup, for the more recent kernels, a vhost task is created for each sub-network to the VM and is unique to the particular QEMU instance and terminated when QEMU exits. For older versions of vhost, only one global vhost task is created to service all VMs on the host system, but is inadequate for system with more than one VM or a VM with more than one network if network data rates are moderate or high.

The base QEMU program does some minor work and sometimes creates some other minor tasks, but these are not major CPU consumers. The processes ids can be used to set the affinitization of the QEMU and virtual processors to the system's physical CPU cores. The following using ps –eLF shows the process ids of the QEMU virtual machine with two virtual CPUs threads:

```
[vm _ ctl]# ps -eLF |grep qemu

root      4138     1  4138  0 … qemu-system-x86 _ 64 -enable-kvm -m 1024 -smp 2 …
root      4138     1  4142  6 … qemu-system-x86 _ 64 -enable-kvm -m 1024 -smp 2 …
root      4138     1  4143  3 … qemu-system-x86 _ 64 -enable-kvm -m 1024 -smp 2 …
```

The primary QEMU task is listed first and has process 4138 in this case. The first VM virtual CPU core in this case has process id 4142, which internally to the VM is CPU 0 in the guest operating system. The second VM virtual CPU has process id 4143 and is CPU 1 in the VM guest operating system. Both of these threads list the owning QEMU process id 4138 before its own process id in the listing. Often another process appears in the list between QEMU and the virtual CPU tasks, however generally disappears after a minute or two, requiring some care to be used in selecting the correct process ids of the virtual processor tasks.

Once process ids of the QEMU tasks are known they can be affinitized to a particular physical CPU core or set of CPU cores. The following example affinitizes the QEMU task and the two associated virtual processor tasks to different CPU cores:

```
[vm _ ctl]# taskset -p 100 4142
pid 4142's current affinity mask: ffffffff
pid 4142's new affinity mask: 100

[vm _ ctl]# taskset -p 200 4143
pid 4143's current affinity mask: ffffffff
pid 4143's new affinity mask: 200

[vm _ ctl]# taskset -p 2000 4138
pid 4138's current affinity mask: ffffffff
pid 4138's new affinity mask: 2000
```

This affinitizes physical host CPU core 8 (100h) to Virtual CPU 0, physical host CPU core 9 (200h) to Virtual CPU 1, and physical host CPU core 13 (2000h) to QEMU primary task. Beforehand, the original masks were set to ffffffff indicating that these processes could be scheduled on any CPU core indicating no affinity to begin with. Affinitization does not need to be restricted to a single CPU core, a mask with multiple CPU mask bits can used.

11.4.2 Affinitizing the vhost Tasks to the Host's Physical CPU Cores

A listing of the created processes, excluding the task threads just after startup shows:

```
[vm _ ctl]# ps -e
  PID TTY          TIME CMD
 ...
 4138 pts/0    00:00:13 qemu-system-x86
 4139 ?        00:00:00 kvm-pit-wq
 4140 ?        00:00:00 vhost-4138
 4141 ?        00:00:00 vhost-4138
```

There are three process listed after the QEMU process which have process ids which are close to the QEMU process id and are also lower process ids than the associated virtual processor ids. The kvm-pit-wq task rarely shows up as a load on the top display, so affinitization probably has very little value. Of interest is the vhost task(s) that have the associated QEMU process id concatenated on the end. Earlier versions of vhost only have one system wide vhost process and do not have an associated QEMU process id in its name in the process listing. The vhost task uses considerable processing power when high throughput communications is being used, so is important for affinitization.

```
[~]# taskset -p 4000 4140
pid 4140's current affinity mask: ffffffff
pid 4140's new affinity mask: 4000

[~]# taskset -p 8000 4141
pid 4141's current affinity mask: ffffffff
pid 4141's new affinity mask: 8000
```

The first vhost task, process id 4140 is for the first network listed in VM startup script and is usually the network assigned to eth0 in the VM. In this case the vhost task was affinitized to physical host CPU 14 (4000h). The second vhost task, process id 4141 is for the second network listed in VM startup script and is usually the network is assigned to eth1 in the VM and is affinitized to physical host CPU 15 (8000h). Inside the VM, the eth0 and eth1 names can be swapped since they can be associated to the virtual MAC address, which needs to be unique within the system. In the host, the name associated to the bridge is the subnet, tap 10 for first listed network and tap11 for the second network are the bridged subnets that these interfaces are on. The associated VM network interface needs to be on the same IP subnet as the associated bridge IP subnet for Ethernet communications to the VM to work on a bridged network.

11.4.3 Affinitizing Tasks Within a VM

Affinitizing the VMs tasks in Linux type guest operating system is done in the same manner as on the host using taskset. As in the case of host, taskset sets CPU virtual processors that the guest's scheduler can run a task on in the same manner as on the host, see earlier documentation on taskset. However the VM Affinitization is only to the virtual CPU of the VM and not directly to a physical CPU of the system. In order to have a specific VM task affinitize to a physical system CPU, the VM virtual CPU QEMU thread must also be affinitized to a physical CPU on a system as discussed earlier. For example, the network test program netserver can be affinitized to the VMs virtual CPU 0 with the following:

```
[~]# taskset -c 0 netserver
```

See earlier documentation on taskset.

Affinitizing to a virtual CPU within the VM without affinitizing to a physical processor core sometimes has some advantages. For example, the best network performance of a VM with 2 virtual CPUs also running 2 maximum performance network tasks would be to have the tasks and associated VM interrupts unaffinitized since Linux kernel default operation is to execute a task on the same processor as network traffic is received on. However, if only one network task is running within a VM with 2 virtual CPUs, having the second CPU do network receive interrupt and software interrupt processing can have better network throughput since two CPUs are used to do the processing, providing both the virtual CPUs are running on the same physical CPU. This can be done by affinitizing the network task to one virtual CPU and the network interrupt processing to another virtual CPU.

11.4.4 Affinitizing Interrupt Processing Within a VM

For Linux base VM guests, Interrupt Processing is affinitization to virtual CPUs in the same manner as on the host, including irqbalance considerations. As in the case of tasks, the affinitization is to the virtual CPU of the VM. In order to be affinitized to a host physical CPU core, the VM virtual CPU must also be affinitized to a host physical CPU core.

One advantage of the VM is a much fewer interrupts and usually with fewer virtual CPUs to consider. The host system often can have hundreds of interrupts registered while the following VM interrupt loading is for a two virtual CPU VM which has two network interfaces:

```
[~]# cat /proc/interrupts
      CPU0     CPU1
...
40:          0          0  PCI-MSI-edge      virtio0-config
41:    2463968     142758   PCI-MSI-edge      virtio0-input
42:        194        467   PCI-MSI-edge      virtio0-output
43:          0          0  PCI-MSI-edge      virtio1-config
44:       8534       1824   PCI-MSI-edge      virtio1-input
45:          0          1  PCI-MSI-edge      virtio1-output
```

Having two networks connected to the VM results in two virtio subsystems, virtio0 and virtio1 being loaded. The network association to the virtio interface can be determined by the network order and MAC address in the QEMU command line and the MAC address listed by ifconfig output. The following can be used to review the network Interrupt Affinity to the Virtual CPUs:

```
[~]# cat /proc/irq/41/smp _ affinity
1
[~]# cat /proc/irq/42/smp _ affinity
1
[~]# cat /proc/irq/44/smp _ affinity
1
[~]# cat /proc/irq/45/smp _ affinity
3
```

We see that most of the default interrupts are affinitized to virtual CPU 0 (1). To affinitize all the virtio input (receive) and output (transmit) interrupts to virtual CPU 1 (2), the following can be used:

```
[~]# killall irqbalance
[~]# echo 2 > /proc/irq/41/smp _ affinity
[~]# echo 2 > /proc/irq/42/smp _ affinity
[~]# echo 2 > /proc/irq/44/smp _ affinity
[~]# echo 2 > /proc/irq/45/smp _ affinity
```

The IRQ balancer should be disabled or evaluated as on the host before IRQ affinitization is done. The configuration interrupts of virtio are not constantly used, so the default setting is probably OK. Affinitizing the network tasks to virtual CPU 0 is the two virtual CPUs VM configuration for network processing.

11.4.5 QEMU and vhost Task Affinitization Using Taskset at QEMU Startup

The VM can be started using taskset when starting QEMU to affinitize the VM to the host CPU core set. However, not all elements of the VM operation are affinitized.

```
taskset -c 8,9,10,11,12,13,14,15 qemu-system-x86 _ 64 -enable-kvm -m 4096 -smp 8 ...
```

Checking the Affinitization of the QEMU and Virtual Processor tasks:

ps -eLF |grep qemu | awk ' { print $4 }' | xargs –n1 taskset -p

```
pid 19848's current affinity mask: ff00
pid 19852's current affinity mask: ff00
pid 19853's current affinity mask: ff00
pid 19854's current affinity mask: ff00
pid 19855's current affinity mask: ff00
pid 19856's current affinity mask: ff00
pid 19857's current affinity mask: ff00
pid 19858's current affinity mask: ff00
pid 19859's current affinity mask: ff00
```

We see that the QEMU main task and the Virtual Processor tasks have the correct affinitization. However since this uses vhost, we check the associated vhost tasks:

```
[vm _ ctl]# ps -e |grep vhost | awk ' { print $1 }' | xargs –n1 taskset -p

pid 19850's current affinity mask: ffffffff
pid 19851's current affinity mask: ffffffff
```

The vhost tasks created by QMEU do not have the set QEMU affinitization. Looking at the CPU loads of the host, in this case we see that the VM was using CPU core 15 which is on physical CPU 1 for the communications process, but vhost is on CPU core 7 (doesn't

have a user task) which is located on the physical CPU 0.

```
[~]# top
top - 03:56:47 up 21 days,  1:02,  3 users,  load average: 0.93, 0.30, 0.14
Tasks: 248 total,   1 running, 247 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.0%us,  0.0%sy,  0.0%ni, 97.7%id,  2.3%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu4  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu5  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  :  0.0%us, 60.8%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi, 39.2%si,  0.0%st
Cpu8  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu9  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu10 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu11 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu12 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu13 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu14 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu15 : 68.2%us, 13.4%sy,  0.0%ni, 18.4%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
...
 PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
19850 root      20   0     0    0    0 S 100.0  0.0   1:05.41 vhost-19848
19848 root      20   0 4859m 745m 3656 S 81.4  4.7   1:57.41 qemu-system-x86
20250 root      20   0 15268 1260  856 R  0.3  0.0   0:00.18 top
...
```

This split of QEMU tasks and vhost on different physical CPUs seems to be consistent and happens for most executions. This is poor since vhost and QEMU tasks have large number of network packets transferred between them and so does constant SMP synchronization across the physical processors. To correct this, we also affinitize the associated vhost tasks to cores within the same physical processor as the QEMU tasks.

```
[~]# ps -e |grep vhost
19850 ?        00:02:52 vhost-19848
19851 ?        00:00:00 vhost-19848

[~]# taskset -p 0ff00 19850
pid 19850's current affinity mask: ffffffff
pid 19850's new affinity mask: ff00

[~]# taskset -p 0ff00 19851
pid 19851's current affinity mask: ffffffff
pid 19851's new affinity mask: ff00
```

This notably improves network performance since SMP synchronization only needs to occur between CPU core other CPU core within the same physical processor rather than doing SMP synchronization across the physical processors, which stalls the CPU moderately more time for each synchronization operation.

12 Glossary

ATR	Application Targeted Routing
COTS	Commercial Off-The-Shelf
DPI	Deep Packet Inspection
FCS	Frame Check Sequence
GRO	Generic Receive Offload
IOMMU	Input/Output Memory Management Unit
KVM	Kernel-based Virtual Machine
LRO	Large Receive Offload
MSI	Message Signaling Interrupt
RSC	Receive Side Coalescing
RSS	Receive Side Scalingv
SP	Service Provider
TCO	Total Cost of Ownership
TSO	TCP Segmentation Offload

13 References

DOCUMENT NAME	SOURCE
RFC 1242 (Benchmarking Terminology for Network Interconnection Devices)	http://www.ietf.org/rfc/rfc1242.txt
RFC 2544 (Benchmarking Methodology for Network Interconnect Devices)	http://www.ietf.org/rfc/rfc2544.txt
RFC 6349 (Framework for TCP Throughput Testing)	http://www.faqs.org/rfc/rfc6349.txt
Advanced Telecommunications Compute Architecture Reference Design	471851_rev05_ATCA_RefDesignUG_review.pdf
Crystal Forest Software for Linux Getting Started Guide	440005_GSGLx_v0_71_review.pdf
Crystal Forest Server – Embedded Form Factor Platform	Order Number: 451210, Revision 0.6
Intel® 82599 10 Gigabit Ethernet Controller Datasheet	http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html
Bandwidth Sharing Fairness	http://www.intel.com/content/www/us/en/network-adapters/10-gigabit-network-adapters/10-gbe-ethernet-flexible-port-partitioning-brief.html
OpenFlow with Intel 82599	http://ftp.sunet.se/pub/Linux/distributions/bifrost/seminars/workshop-2011-03-31/Openflow_1103031.pdf
Wu, W., DeMar,P. & Crawford,M (2012). A Transport-Friendly NIC for Multicore / Multiprocessor Systems.	IEEE transactions on parallel and distributed systems, vol 23, no 4, April 2012. http://lss.fnal.gov/archive/2010/pub/fermilab-pub-10-327-cd.pdf
Why does Flow Director Cause Placket Reordering?	http://arxiv.org/ftp/arxiv/papers/1106/1106.0443.pdf
Linux man pages	Man taskset
Linux kernel Documentation	Documentation/IRQ-affinity.txt
Netperf	http://www.netperf.org/netperf/training/Netperf.html
Crystal Forest	http://www.intel.com/content/www/us/en/communications/global-communications-network.html
IA packet processing	http://www.intel.com/p/en_US/embedded/hwsw/technology/packet-processing_
Intel® DPDK	http://www.intel.com/go/dpdk http://www.intel.com/p/en_US/embedded/hwsw/technology/packet-processing

Disclaimers

Δ Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See www.intel.com/products/processor_number for details.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order. Copies of documents which

have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web site at www.intel.com.

*Other names and brands may be claimed as the property of others.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Visit: http://www.intel.com/products/processor_number for more information.