Multi-PCIe socket network device

Achiad Shochat

Mellanox
Yokneam, Israel
achiad@mellanox.com

Abstract

This paper explains why connecting a single network port device to the host CPU through multiple physical PCIe sockets may be desired in some cases and suggests a software model to handle it.

Many concepts discussed here are generally applicable to any I/O device that uses DMA and for any host bus technology (not necessarily PCIe), but we will focus on the PCIe network devices case.

Keywords

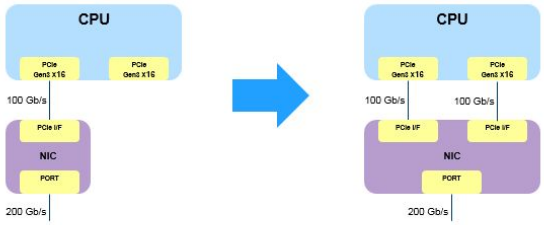
Linux, kernel, PCIe socket, network port, netdev, aRFS, NUMA, server

Introduction

There are two cases where connecting a network port device through multiple physical PCIe sockets may be desired:

1) The network port speed is higher than that of any available PCIe socket in the host system
2) NUMA (Non Uniform Memory Access) Systems

Case 1: network port faster than PCIe socket

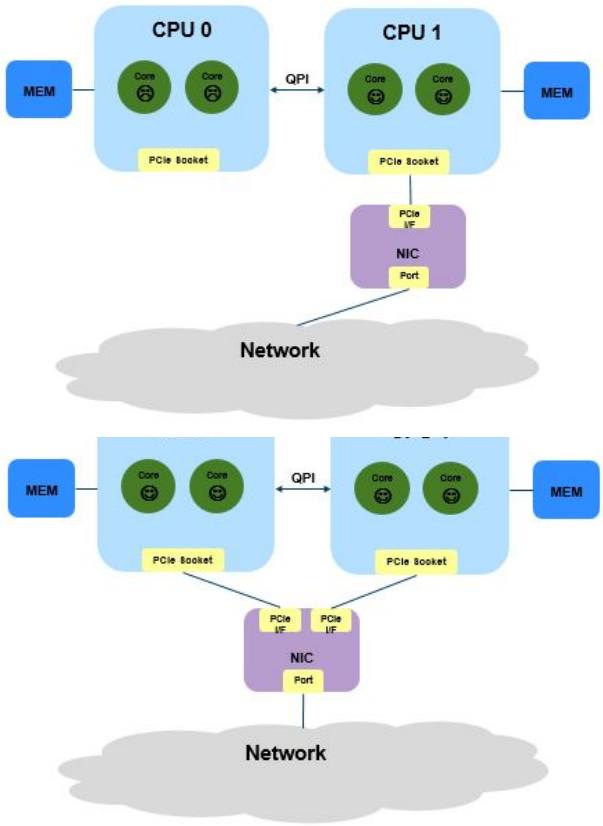


The motivation in this case is quite obvious…
This case gets more viable lately as network device ports running at 200 Gb/sec are being introduced these days while most servers do not have PCIe sockets supporting this speed.

Case 2: NUMA systems

A single PCIe socket is likely to satisfy the application network bandwidth needs, even with compute power-intensive systems such as modern NUMA systems usually are. In NUMA systems however a given PCIe socket is local to a given host memory instance only. Therefore a NIC attached to a single PCIe socket can benefit memory locality for its DMA and I/O operations only with a single memory instance and thus cannot take advantage of the NUMA architecture. A network application running on a remote CPU core will incur NIC DMA to the remote CPU memory which obviously yields inferior performance. Furthermore, those DMAs to remote memory load the memories interconnect bus (e.g QPI) which may create other bottlenecks in the system. This severely limits the system scaling.

By connecting the NIC to multiple PCIe sockets those performance scaling pitfalls are resolved.



[TODO: add local / remote access performance numbers]

Netdev per network port

The Linux kernel convention is to represent each network port by a kernel netdev (struct net_device). So far in the kernel, netdevs representing physical device ports are being created upon device PCI probe – one (or sometimes more) netdev per PCI function. This is fine as long as each network port is represented by a single PCIe function. Doing so for a multi-PCIe bus device port however will break the netdev-per-port convention and yield multiple

netdevs over a single network port. Therefore the right approach is to create a single netdev that operates the device through its multiple PCIe buses and thus hide the PCIe topology details from the network stack and the operating system.

Why not use Linux bond/team

One could think of an alternative approach - have a netdev per device PCIe bus (as done so far) and enslave them under a bonding netdev. Well, that approach breaks the netdev-per-port convention and as a consequence creates un-clarity when it comes to managing device port offloads such as RSS and aRFS. Also, as the single PCIe bus device driver would be un-aware of being just a part of the whole network port driver it would likely strive to create RX/TX queue per system CPU core rather than just per local CPU core, yielding RX/TX queues redundancy in the system.

In addition, setting a bond device requires manual settings – not a nice out-of-box experience.

Multi-PCIe bus device detection

In order to achieve a single netdev over a multi-PCIe bus device port, some method is required for the device driver to identify that multiple PCIe buses connect the same network port.

Ideally that method should be generic (NIC vendor agnostic) – through the device PCI configuration space. Maybe by having the PCI class code specify that the device has multiple PCIe buses and specify the common network port ID in the capabilities list.

A device specific method may be used as well.

Ordering and TX/RX queues PCIe bus affinity

Traffic sent to/from a given netdev TX/RX queue is expected to be delivered in order.

By PCI spec ordering rules, traffic sent on a given PCIe bus is delivered in order. But the PCI spec does not guaranty any ordering between traffic sent on different PCIe buses.

This implies that each TX/RX queue must be affined with a single PCIe bus.

In MUNA systems this aligns with the common practice of assigning an RX/TX queue per core.

Shared netdev resources

Unlike RX/TX queues, other device resources are not affined with a PCIe bus. These are mostly resources controlling the device flow steering – RSS hash function, RSS indirection table, aRFS rules, and any other flow rules.

Device resources management

Through which PCIe bus of the device its resources are managed is a device specific implementation policy. Possible policies for example:

Manage all resources via one of the PCIe buses

Manage affined resource (RX/TX queues) through their designated bus and un-affined resources through one of the buses

Manage any resource through any bus

RSS (Receive Side Scaling)

In Order to support RSS the device must support a single indirection table that points to RX queues affined with different PCIe buses.

Traditionally the indirection table controls the load balancing of ingress traffic over the host CPU cores. Now it also implicitly controls the load balancing of ingress traffic over the device PCIe buses.

aRFS (Accelerated Receive Flow Steering)

By having a single netdev running over multiple PCIe buses, and as RX queues are affined with a PCIe bus, aRFS will naturally steer flows through the PCIe bus assigned with the core's RX queue. In MUNA systems this also means that RX flows will be naturally steered to local memory. This way there are no remote memory NIC DMAs, which relieves the memories inter-connect (QPI), yields linear scalability with systems cores and reduces the networking latency.

PCIe hot plug

Different PCIe buses may be attached/detached (either physically or virtually by a shell command line) to/from the system dynamically and independently of each other.

A multi-PCIe bus device driver should create an RX/TX queue per bus-local CPU core. Therefore the amount of netdev RX/TX queues depends on the number of device PCIe buses that are attached in the system at a given moment.

The amount of netdev RX/TX queues however must be determined at netdev allocation (at alloc_etherdev_mqs()).

So a multi-PCIe device driver may take two approaches to handle it:

1) Static approach
    a. Register a netdev only when all PCIe buses of the device are attached
2) Dynamic approach
    a. Register a netdev upon first PCI probe of any of the device buses, dynamically enable/disable RX/TX queues upon

further PCI bus probes/removes and un-register the netdev upon removal of its last PCI bus

SR-IOV and device assignment

So far, cloud management SW, such as OpenStack, assume that a network port is represented by a single PCIe function. Thus, when it wishes to provide a virtual machine (VM) with a pass-through network port it assigns it a single PCIe virtual function (VF) who's parent physical function (PF) represents the network port.

With a multi-PCIe bus device this assumption is no longer true. Each VM should be assigned with multiple VFs - one per device PCIe bus. Virtualization management SW need to be extended to support it. This emphasizes the need for making the multi-PCIe device detection method generic.

Supporting NIC devices

All Mellanox NICs starting from ConnectX-4 support multi-PCIe buses connectivity:

ConnectX-4
ConnectX-4Lx
ConnectX-5

All future Mellanox NICs are expected to keep supporting it.