Secure I/O Device Sharing among Virtual Machines on Multiple Hosts

Cheng-Chun Tu
Stony Brook University
1500 Stony Brook Road
New York, U.S.A
u9012063@gmail.com

Chao-tang Lee
Industrial Technology
Research Institute
195 Chung Hsing Road
Hsinchu, Taiwan
marklee@itri.org.tw

Tzi-cker Chiueh
Industrial Technology
Research Institute
195 Chung Hsing Road
Hsinchu, Taiwan
tcc@itri.org.tw

ABSTRACT

Virtualization allows flexible mappings between physical resources and virtual entities, and improves allocation efficiency and agility. Unfortunately, most existing virtualization technologies are limited to resources in a single host. This paper presents the design, implementation and evaluation of a multi-host I/O device virtualization system called *Ladon*, which enables I/O devices to be shared among virtual machines running on multiple hosts in a secure and efficient way. Specifically, *Ladon* uses a PCIe network to connect multiple servers with PCIe devices and allows VMs running on these servers to directly interact with these PCIe devices without interfering with one another. Through an evaluation of a fully operational *Ladon* prototype, we show that there is no throughput and latency penalty of the multi-host I/O virtualization enabled by *Ladon* compared to those of the existing single-host I/O virtualization technology.

1. INTRODUCTION

The ultimate goal of I/O virtualization is to allow each VM on a physical machine to directly interact with I/O devices assigned to it. Accordingly, the hardware support for I/O virtualization consists of two parts. First, IOMMU [13] is a memory management unit that performs translations between I/O device addresses and physical memory addresses, and thus offers protection of I/O devices and their memory from one another. Second, virtualization of I/O device allows multiple VMs to share a single physical I/O device as if each of them owns a dedicated instance. IOMMU is already supported in the chipset of most commercial servers, but currently only newer network interface cards support device virtualization.

Almost all I/O devices on modern servers are connected to the CPU and memory via the PCI Express (PCIe) bus [16]. The PCIe bus architecture requires that devices connected to a PCIe bus form a domain and be logically organized into a tree, with the head of the tree being the tree's *root complex*. Typically the PCIe devices in a server belong to a single domain, and the root complex is the CPU(s). Accordingly, the ability to allow VMs running in a server to interact directly with PCIe devices in the server's PCIe domain is called *single-root I/O virtualization* or SRIOV [5, 9, 8].

The advent of cloud computing calls for a re-thinking of the optimal building block for cloud-scale data centers. One promising proposal is a modular server cluster that shares I/O devices over a PCIe network. That is, servers in such a cluster could directly interact with a common set of network interface cards and/or disk controllers that belong to a single PCIe domain. I/O device sharing decouples a server's CPU/memory from its I/O devices, and thus offers several crucial advantages. First, compared with a traditional cluster architecture in which each node has its own disks, the new cluster architecture allows the shared disks to remain accessible to nodes outside the cluster even when most of the nodes in the cluster are turned off for power saving. Second, the new architecture makes it possible to deploy a smaller number of high-performance and thus expensive network interface cards, e.g. 10GE or 40GE NIC, than the number of nodes in the cluster, saving both cost and cabling complexity. With proper software, it is even possible for a cluster node to leverage multiple NICs simultaneously. Third, the PCIe network connecting the cluster nodes and the I/O devices they share could substantially reduce the power consumption of intra-cluster communications because the PCIe transceiver is meant for short-distance transmission and thus is much simpler and less power-hungry than 10GE or Infiniband [7]. Coupling I/O device sharing with I/O virtualization enables multiple VMs on multiple servers to interact directly with PCIe devices shared among the servers. This capability is called *multi-root I/O virtualization* or MRIOV [4]. Unfortunately no commercially available PCIe devices support MRIOV, and there is no sign that this situation would change soon.

This paper describes the design, implementation and evaluation of a novel PCIe-based system interconnect architecture called *Ladon* that connects multiple servers with PCIe devices supporting SRIOV and allows the PCIe devices to be seamlessly shared by VMs running on these servers. *Ladon* leverages the non-transparent bridge feature [23, 18] of modern PCIe network switches, IOMMU, and hardware support for memory virtualization [17], to enable VMs to access shared PCIe devices using unmodified native device drivers without suffering from any noticeable performance degra-

ISCA'13 Tel-Aviv, Israel

dation. *Ladon* is the first to demonstrate the feasibility of an all-software approach to support the MRIOV functionality using only SRIOV PCIe devices. Possible applications of *Ladon* include sharing of PCIe-based flash memory cards and graphics processing units (GPU).

2. PCI EXPRESS ARCHITECTURE

2.1 Basic Model

Because most I/O devices installed on modern servers are PCIe devices, extending the PCIe technology from an intra-server backplane for I/O device connection to an inter-server backplane for I/O device sharing enables existing PCIe devices to be immediately shareable without any modifications.

Each PCIe device is uniquely identified by a combination of a 8-bit bus number, a 5-bit device number and a 3-bit function number, and is given a *configuration space*, which is 4096 bytes wide and contains a standardized part (such as device ID, vendor ID, command, status, base address register or BAR, etc.) and a device-specific part. At start-up time, a PCIe device is discovered and configured by the system. At run time, the system sets up the PCIe device for DMA transactions, and the device's DMA engine carries out these DMA transactions accordingly, emitting an interrupt to the system when the DMA transactions are completed. Therefore, the four types of operations in which a PCIe device involves are discovery, configuration, DMA, and interrupt, and they are all executed in the form of read and write operations.

A PCIe card occupying a PCIe slot could function as one or multiple physical functions, each acting like a logical PCIe device with its own Bus/Device/Function ID and configuration space. The PCI-SIG standard on SRIOV virtualizes a PCIe device into multiple virtual PCIe devices each of which could be assigned to a distinct VM. SR-IOV separates a physical function (PF) from a virtual function (VF). A PF is a full-function PCI device with its IDs and complete configuration space, whereas a VF is a lighter-weight PCIe device with its IDs but without a complete configuration space. Instead, each VF only has its own copy of the data plane-related portion of the configuration space, such as transmission/reception queue heads and BAR0-BAR5, but the control plane-related portion is borrowed from its underlying PF's configuration space. As a result, SRIOV PCIe devices require special support from BIOS for memory-mapped I/O address range allocation, and from the OS for control-plane configuration. The number of physical functions on a non-SRIOV PCIe devices is typically much smaller than the number of virtual functions on an SRIOV PCIe device. *Ladon* could support both multi-PF and multi-VF PCIe devices.

2.2 Transparent Bridging

Although PCIe is designed to be a system backplane, it has now developed sufficiently to scale to a large number of end points in a PCIe domain through one or multiple PCIe transparent bridges (TB) or switches. Conceptually, every operation on a PCIe domain is specified in terms of a memory read or write operation with a target address, which is then translated into a PCIe network packet routed to the corresponding PCIe end point. More concretely, every end point in a PCIe domain is responsible for a specific range of the physical memory address space associated with the PCIe domain, which consists of the end point's configuration space and the memory address areas specified in its BARs. As the physical memory address range of a PCIe device attached to a PCIe network is initialized or re-configured, the routing tables of the network's PCIe switches are set up accordingly to route memory read and write operations whose target address falls within the PCIe device's assigned memory address range. Modern PCIe switches also support multicasting, where a memory write operation with a multicast target address is sent to all PCIe end points that belong to the corresponding multicast group.

The end points of a PCIe domain form a tree, whose root is the root complex, which connects to the CPU/memory. Accordingly, each PCIe switch has an upstream port and multiple downstream ports, where upstream and downstream connectivities mirror those in the tree.

IOMMU is originally designed to prevent a PCIe end point from accessing physical memory areas outside its assigned range in the root complex, and is typically implemented in the chipset of the host. Modern PCIe switches leverage the IOMMU to prevent unauthorized accesses in peer-to-peer PCIe transactions by routing these transactions first to the root complex, where the IOMMU resides, and then routed down to the target port.

2.3 Non-Transparent Bridging

At any point in time, every PCIe domain has exactly one active root complex. In a fail-over configuration, there could be two or multiple root complexes in a PCIe domain, but only one of them is active. Non-transparent bridge (NTB) is a part of the PCI-SIG standard, was designed to enable two or more PCIe domains to inter-operate together as if they are in the same domain, and thus makes a key building block for the proposed *Ladon* system. Intuitively, an NTB is like a layer-3 router in data networking, and isolates the PCIe domains it connects so that the invariant that each PCIe domain has exactly one root complex always holds.

A two-port NTB represents two PCIe end points, each of which belongs to a separate PCIe domain. These two end points expose a type 0 header type in their CSR and thus are discovered and enumerated by their respective PCIe domains in the same way as ordinary PCIe end points. However, an NTB provides additional memory address translations, device ID translations and messaging facilities that allow the two PCIe domains to work together as a whole while keeping them logically isolate. More generally, a PCIe switch with X NTB ports and Y TB ports allows the PCIe domain in which the Y TB ports participates, called the *master* domain, to work with X other PCIe domains, each of which is a *slave* domain and supported by an NTB port. The side of an NTB port that connects to a slave domain is the *virtual* side, whereas the other side is called the *link* side.

The "magic" of an NTB lies its ability to deliver a memory read/write operation initiated from one PCIe domain to be delivered to another PCIe domain, with some translations, and then executed. From the initiating domain's standpoint, the memory read/write operation is logically executed locally within its domain, although physically it is executed in a remote domain. As mentioned earlier, every PCIe end point is equipped with a set of BARs (BAR0/1 are used for CSR, so only BAR2 to BAR5 are available), which

specify the portions of the physical memory address space of the end point's associated PCIe domain for which it is responsible. That is, all memory read and write operations in a PCIe domain that target at the memory address ranges associated with a PCIe end point's BARs are delivered to that end point. An NTB port associates with every BAR on the link (virtual) side a *translation* register, which converts a received memory address at the link (virtual) side into another memory address at the virtual (link) side. The side of an NTB port providing the BAR is the *primary* side whereas the other side is the *secondary* side. For example, one could translate the address 0xF8800000 in BAR2 of the link side to the address 0xF8A00000 of the virtual side by writing 0xF8A00000 to BAR2's translation register. In this case, the link side is the primary side and the virtual side is the secondary side. Address translation in NTB is *unidirectional*: only a memory read/write operation that hits a BAR at the primary side is translated and relayed to the secondary side, but *not vice versa*. Using the same example, a write operation against the address 0xF8800000 at the link side is translated and relayed as a write operation against the address 0xF8A00000 at the virtual side, but a write operation against the address 0xF8A00000 at the virtual side does not necessarily get translated and relayed to the link side.

When a memory read/write operation originating in one PCIe domain is translated and executed in another PCIe domain through an NTB, the source device ID used in the originating domain is different from that used in the executing domain. Suppose the source device ID of a memory read/write operation in the originating domain is <B1, D1, F1>, then the source device ID for the translated memory read/write operation in the target domain across an NTB becomes <B2, I1, F1>, where B2 is the bus number of the target domain, and I1 is an index into a device ID translation table (or LUT) in the NTB that records the original bus and device number, B1 and D1 in this case. When a response from this translated memory read/write operation is returned, the response is delivered to <B2, I1, F1> and then across the NTB to <B1, D1, F1> in the originating domain. In summary, the source device ID is translated from the link side to the virtual side by searching the device ID translation table with the bus and device number to obtain the corresponding index number, and is translated from the virtual side to the link side by using the device number as an index into the device ID translation table to obtain the corresponding bus and device number. If the search for the device ID translation table initiated by a memory read/write operation produces no match, the memory read/write operation is aborted.

To allow the two PCIe domains across an NTB port to work more closely, the NTB provides *doorbell* registers for one domain to interrupt the other, and *scratchpad* registers for one domain to pass some information to the other. These two messaging mechanisms provides an out-of-band control path for the two PCIe domains across an NTB port to better communicate with each other.

3. SYSTEM ARCHITECTURE OF *Ladon*

Figure 1 depicts that hardware architecture assumed by *Ladon*, where a set of SRIOV PCIe devices are installed in the management host and shared by multiple compute hosts, some of which are virtualized and some of which are non-virtualized. The management host is in charge of the management domain and is thus responsible for the allocation of all resources on the shared PCIe devices, whereas each of the compute hosts owns a separate sharing domain and is given virtual PCIe devices by the management host. *Ladon* enables SRIOV devices to be shared by VMs running on the compute hosts in a safe and efficient way.

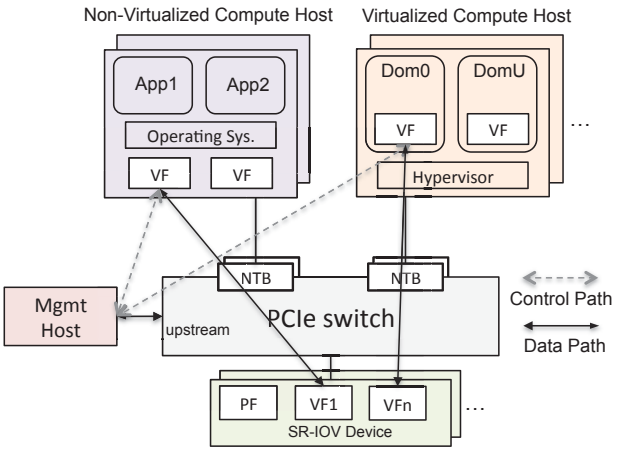

Figure 1: *The hardware system architecture that* Ladon *assumes and consists of a management host, where shared I/O devices reside, and a set of compute hosts sharing the I/O devices.*

3.1 Control Plane

When a PCIe-based server starts up, its BIOS probes the PCIe slots, configures and initializes discovered PCIe devices, and allocates to them system resources such as interrupt numbers and physical memory ranges. The result of this PCIe probing operation in BIOS is a set of data structures defined by a platform-independent interface [12] and passed to the operating system after the latter is loaded. The initialization logic of an operating system parses these BIOS data structures and obtains the detailed list of PCIe resources under its management. Because an NTB port effectively isolates the two PCIe domains it conjoins, the BIOS in each domain could not probe beyond the NTB. Therefore, after the PCIe probing operation, the BIOS in the management host in Figure 1 discovers one SRIOV PCIe device and N PCIe devices, whereas the BIOS in each compute host discovers only one PCIe device.

When an operating system boots up on a physical machine, it enumerates the PCIe devices by querying the BIOS. When a virtual machine boots up on top of a hypervisor, the VM's OS performs the same enumeration procedure. However, the hypervisor of the VM's compute host intercepts these enumeration queries, and allocates to the VM a list of virtual PCIe devices according to its configuration specification, each of which appears exactly like a physical PCIe device except it is not. To back up these virtual PCIe devices, the hypervisor requests the platform manager on the management host to assign actual PCIe devices in the management domain, and then binds the assigned actual PCIe devices to the virtual PCIe devices allocated to the VM. After the binding, the VM could directly interact with these actual PCIe devices in the management domain using their

associated native device drivers without modifications.

When a compute host starts up in Figure 2, its BIOS only sees an on-board PCIe device corresponding to the virtual side of an NTB port (NTB Virtual). As more VMs boot up on this compute host, additional virtual PCIe devices are installed on the compute host and bound to their corresponding actual PCIe devices through the NTB port. Therefore, *Ladon* provides more resource allocation flexibility, where the platform manager can dynamically allocate and de-allocate PCIe resources in the management host to compute hosts that share its I/O devices.

A PCIe end point is uniquely characterized by the following three address ranges:

- PCIe configuration space registers (CSR),
- Memory-mapped I/O (MMIO) region, and
- Message Signaled Interrupt (MSI) or MSI-X address.

The configuration space registers (CSR), which, among other things, include device ID, header type and 6 base address registers (BAR) that specify physical memory address ranges for which the device is responsible. The BARs are used to specify additional memory address ranges for memory-mapped I/O (MMIO) regions, i.e., target addresses used in MMIOs for device-specific commands and status reports. A PCIe device's MSI address is the target address that it uses to issue interrupts to the CPU.

In *Ladon*, when a VM on a sharing compute host H boots up and is supposed to see a virtual PCIe device, say a virtual network interface card (NIC), the host's hypervisor asks *Ladon*'s platform manager for an actual PCIe device in the management domain to back up the virtual NIC. An *actual* PCIe device in the management domain could be a PF or a VF of a *physical* PCIe device, for example an SRIOV NIC. More generally, *Ladon*'s platform manager allocates an actual PCIe device, identifies its three characterizing address ranges, sets up necessary mappings in the NTB to translate between these addresses in the management domain and their counterparts in H's domain, and informs H's hypervisor accordingly. The hypervisor on H emulates the three translated address ranges of each virtual PCIe device by including them into the memory address ranges specified in the BARs of the virtual side of the NTB port that connects H with the management host. After the set-up, the OS of a VM on H configures and commands the VM's virtual PCIe device using normal memory read/write operations, and H's NTB port translates and relays these operations to the target domain.

3.2 Data Plane

Bandwidth-intensive PCIe devices such as network interface cards or disk controllers include a DMA engine to move data to and from memory without involving the CPU. A DMA transaction between a PCIe device and memory consists of three steps. First, the device's driver in the OS sets up the DMA transaction's source or target addresses by writing to the PCIe device's command registers. Second, the PCIe device's DMA engine initiates the DMA operation over the PCIe network. Third, the PCIe device issues an interrupt when the DMA operation is completed.

In *Ladon*, a virtual PCIe device with which the OS of a VM running on a compute host sets up a DMA transaction

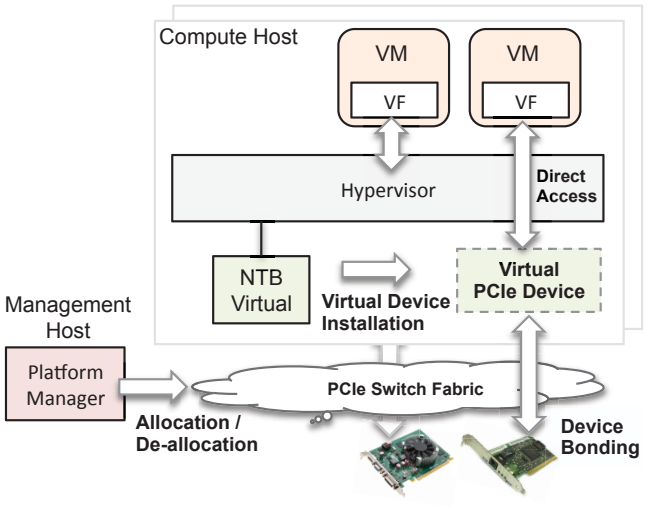

Figure 2: *Flexible resource allocation in Ladon: the platform manager running on the management host dynamically allocates and de-allocates the PCIe resources residing in the management host to sharing compute hosts and their VMs.*

must be backed up by an actual PCIe device in the management host. The source or target address the VM's OS writes to the virtual PCIe device in the first step is a *device virtual address* in the associated actual PCIe device. When the actual PCIe device performs memory copying operations using this device virtual address, it is first translated by the management domain's IOMMU to a *machine physical address* of the management host, which in turn is accepted by the virtual side of an NTB port and translated into another *device virtual address* in the PCIe end point corresponding to the NTB port's link side. This device virtual address is then translated by the IOMMU of the VM's compute host into another *machine physical address* of the VM's compute host. However, the VM's OS refers to the target address of a DMA transaction originally in terms of a *guest virtual address*, which is translated by the OS's page table to a *guest physical addresses*, which in turn is translated by the hypervisor to a *machine physical addresses*. Because translating from a guest virtual address in a VM of a compute host to a device virtual address of an actual PCIe device in the management host involves look-ups of multiple privileged translation tables in IOMMUs, NTB translation register and extended page table, a VM on a compute host of the *Ladon* architecture requires help from the underlying systems software to properly set a DMA transaction's target or source address. In addition to a DMA transaction's source/target address, the device ID of a DMA transaction's initiating device is also translated as a memory copy operation crosses domains over an NTB, as explained in the last section.

Ladon allows the propagation of an interrupt generated by an actual PCIe device in the management host across an NTB and delivered to the CPU of a compute host as if the interrupt comes from the corresponding virtual PCIe device. Modern PCIe devices no longer use the conventional pin-based interrupt mechanism because it is quite limited and inflexible. Instead, they use the message-signaled interrupt (MSI) mechanism, which allows a PCIe device to issue an interrupt by writing a special payload to a special memory

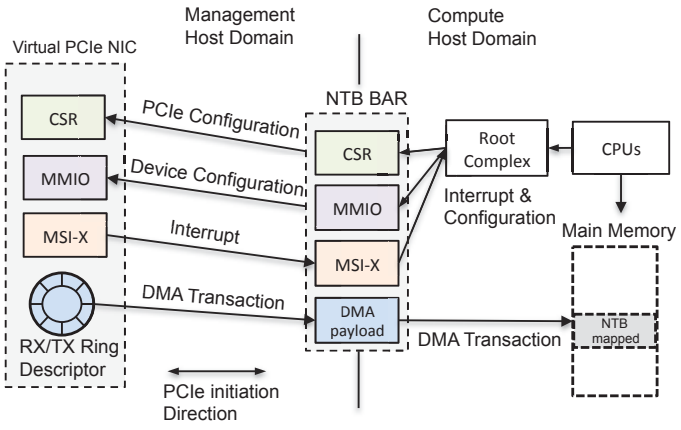

Figure 3: *To set up a virtual NIC for a VM on a compute host, which is backed by a VF of an SRIOV NIC on the management host,* Ladon *identifies the virtual NIC's CSR, MMIO, MSI-X and DMA payload areas, and installs necessary mappings in the BARs of the NTB port connecting the compute host with its associated management host. The memory addresses in the transmission/reception descriptor rings of the VF are filled by a native device driver inside the VM at run time.*

address that is in the local APIC space. Every actual PCIe device contains an MSI-X table for the systems software to specify the payload and address used to generate each possible interrupt from the device. For example, the MSI-X table of an VF of the Intel SRIOV NIC contains three addresses 0xFEE00518, 0xFEE00598, 0xFEE00618 for its reception, transmission and message interrupts, respectively. Unlike the target/source addresses of DMA transactions, addresses used in MSI-X interrupt delivery go through the translation in an NTB and the interrupt remapping table in the management domain and compute host.

When an actual PCIe device in the management host performs a memory copying operation against a physical memory area in a compute host through DMA, the compute host's physical memory area holding these DMA payloads must be made accessible to the actual PCIe device. To do so, *Ladon* creates an aperture from the management host's machine physical address space to the compute host's machine physical address space by using a BAR in the NTB port's link side and properly setting up the associated translation register.

Figure 3 shows how *Ladon* sets up a virtual NIC for a VM on a compute host, which is backed by a VF of an SRIOV NIC on the management host. It identifies the virtual NIC's CSR, MMIO, MSI-X and DMA payload areas, and installs necessary mappings in the BARs of the NTB port connecting the compute host with its associated management host. The memory addresses in the transmission/reception descriptor rings of the VF are filled by a native device driver inside the VM at run time. Two BARs of the NTB are used. One BAR of the virtual side of the NTB port is used to support CSR and MMIO areas, for which the virtual side of the NTB port is the primary side. One BAR of the link side of the NTB port is used to support MSI-X and DMA payload areas, for which the link side of the NTB port is the primary side.

4. SECURE SHARING

The trusted computing basis (TCB) of the *Ladon* architecture consists of the management software running on the management host and the hypervisor and administration domain on each compute host. In an SRIOV environment, the translation table between guest physical addresses and machine physical addresses, e.g. Intel's extended page table (EPT), prevents a malicious VM from corrupting physical memory areas and I/O devices that it is not authorized to modify; the IOMMU prevents a malicious VM from setting up write DMA transactions that corrupt physical memory areas and I/O devices that it is not authorized to modify. In an MRIOV environment, a malicious VM could directly write or set up write DMA transactions to corrupt the following resources:

- The physical memory portion of the management host's machine physical address space,
- The non-physical memory portion of the management host's machine physical address space,
- The machine physical address space of any other compute host than the VM's host, and
- The portion of the machine physical address space of the VM's host that the VM is not authorized to access.

The ability to prevent these attacks is a key to the viability of the *Ladon* architecture, which exploits multiple translation mechanisms provided by PCIe and server virtualization to ensure the proposed I/O device sharing is indeed immune to these attacks.

4.1 Address Spaces and Translation Tables

Let's use the hardware structure shown in Figure 3 to illustrate the address spaces and the associated translation tables in the *Ladon* architecture. When the DMA engine on a VF of an SRIOV PCIe device in the management host (MH) initiates a DMA operation against a memory area of a compute host (CH), it uses a device virtual address DVA_{vf} specific to that VF. DVA_{vf} is then translated via the MH's IOMMU into a machine physical address MPA_{mh} of the management host. MPA_{mh} is accepted by a BAR of the link side of the NTB port connecting MH and CH and translated by its associated translation register to another device virtual address DVA_{ntbvs} specific to the virtual side of the NTB port. Eventually DVA_{ntbvs} is translated by the CH's IOMMU into a machine physical address MPA_{ch} of the compute host. At the same time, the source device ID that the VF uses to initiate a DMA transaction in the management host is translated by the NTB port's device ID translation table (or LUT) into another source device ID in the compute host.

When a VM running on a compute host (CH) configures a VF in the management host (MH), the address it starts with is a guest virtual address GVA_{vm}, which is translated by the VM's guest page table into a guest physical address GPA_{vm}. Then the hypervisor translates GPA_{vm} into a machine physical address MPA_{ch} in the compute host. MPA_{ch} is accepted by a BAR of the virtual side of the NTB port connecting MH and CH and translated by its associated translation register to another device virtual address DVA_{ntbls} specific to the link side of the NTB port. Eventually DVA_{ntbls} is

translated by the MH's IOMMU into a machine physical address MPA_{mh} of the management host, which is eventually picked by the target VF.

Each VM has a set of per-process guest virtual address spaces and exactly one guest physical address space. Every compute and management host has one machine physical address space, and one PCIe device ID address space. Every PCI end point has its own device virtual address space. Guest page table maps guest virtual addresses to guest physical addresses. Extended page table (EPT) maps guest physical addresses to machine physical addresses. IOMMU translation table maps device virtual addresses to machine physical addresses. BAR translation registers map machine physical addresses to device virtual addresses. NTB device ID translation table or LUT maps a PCIe device ID in one domain into another PCIe device ID in another domain.

4.2 Access Control

A VM running on a compute host CH is only allowed to write to the VM's guest physical address space and CH's physical memory area that is allocated to the virtual PCIe devices assigned to the VM. As in the case of SRIOV, *Ladon* accomplishes this by properly programming CH's EPT to restrict the VM's accesses to CH's machine physical address space to the portion that it is authorized to access. With this EPT set-up, it is impossible for the VM to modify any machine physical address ranges of the management host (MH) or other compute hosts that are not mapped to CH's machine physical address space.

To prevent a write DMA transaction set up by a VM running on CH from corrupting MH's physical memory, *Ladon* leverages MH's IOMMU to control which parts of MH's physical memory each of its PFs and VFs could access. In theory, the same IOMMU could be used to control which parts of MH's machine physical address space each of its PFs and VFs could access. Modern PCIe switches support an ACS (Access Control Service) feature, which, when enabled, forces all peer-to-peer traffic to go through the root complex's IOMMU. That is, when receiving a peer-to-peer packet, the PCIe switch redirects it to the root complex port, which contains the IOMMU, and then injects it back to the switch after translation. In practice, however, some servers do not always correctly inject packets that are translated by the IOMMU back to the PCIe switch. As a result, the IOMMU mechanism does not always work for peer-to-peer transactions between PCIe switch ports.

To ensure a write DMA transaction set up by a VM running on CH is allowed to access only CH's physical memory, *Ladon* exploits the NTB's device ID translation table and the compute host's IOMMU. When a VF in MH with the device ID [a:b.c] is assigned to CH, *Ladon* inserts an entry <a:b> into the device ID translation table or LUT in the NTB port connecting MH and CH, which allows the VF's original device ID [a:b.c] to be translated to another device ID in CH. If the device ID translation table of the NTB port associated with any other compute host does not contains any entry <a:b>, it is not possible for the VF to access other compute hosts' memory. However, it is possible that the device ID translation table of the NTB port associated with some compute host other than CH contains an entry <a:b>; in that case the VF's device ID <a:b.c> could be successfully translated to another device ID in this compute host. Fortunately, because the VF is not originally assigned to this compute host, this compute host's IOMMU will not have a translation table for the translated device ID, and the DMA operation is eventually dropped. In summary, *Ladon* rejects an unauthorized DMA operation from a VF in MH to any other compute host than the compute host to which it is assigned, because of a miss in the device ID translation table or a lack of the corresponding IOMMU translation table.

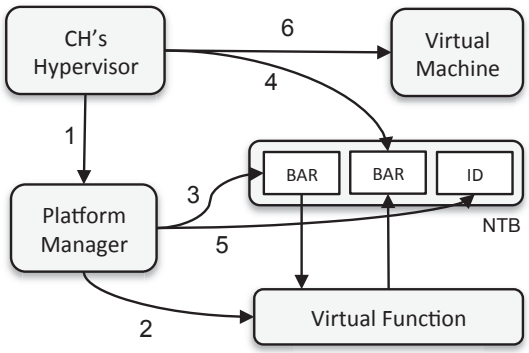

Figure 4: Ladon *requires six steps to set up a virtual PCIe device for a newly started VM.*

5. PROTOTYPE IMPLEMENTATION

The hardware testbed for the *Ladon* prototype consists of four Intel X86 servers, two 10GE NICs, two PLX PEX8619 NTBs, and one PLX PEX8696 PCIe switch. One server serves as the management host, which is a Supermicro E3 tower server, has an 8-core Intel Xeon 3.4GHz CPU and 8GB of memory. Two other servers serve as compute hosts, both of which are ASUS machines, each equipped with an 8-core Intel i7 3.4GHz CPU and 8 GB of memory. These two compute hosts run KVM with the Intel VT-d support enabled so that multiple virtual machines could run on them. The last server is the same as the compute host, equipped with an additional Intel 10GE NIC, and serves as a remote test host for network traffic load generation and reception. The management host is connected to the upstream port of the PEX8696 PCIe switch through a PCIe adaptor plugged into the host's x16 PCIe slot. Each of the two compute hosts is connected to one port of a PEX8619 NTB through a PCIe adaptor plugged into its x8 PCIe slot, with the other port of the NTB connected to a downstream port of the PCIe switch. In addition, an Intel 82599 SRIOV 10GE NIC is plugged into a downstream port of the PCIe switch and a duplex fiber optic cable connects this SRIOV NIC with another Intel 82599 10GE NIC directly, i.e., without a switch or hub in between.

The PEX8696 PCIe switch is a 96-lane, 24-port PCIe Gen2 switch, where each lane provides 4Gbits/s raw transfer bandwidth. We assigned 16 lanes of this PCIe switch to the management host, 8 lanes to each of the two compute hosts, and another 8 lanes to the Intel 82599 NIC. Because the management host is connected to the PCIe switch's upstream ports, the VFs and PF on the Intel NIC are configured and under the control of the management host's OS.

In the prototype, we ran Fedora 15 with the Linux kernel version 2.6.38 on each of the four servers and the virtual machines running on top of the two compute hosts that are virtualized. The platform manager runs as a kernel thread

in the management host, which receives requests from the hypervisors in the compute hosts, performs allocation and de-allocation of virtual PCIe devices, and associates virtual PCIe devices with actual PCIe devices under its control.

We modify the Linux kernel hosting the KVM hypervisor and running on each compute host to intercept the PCIe device enumeration operations from the VMs, and send virtual PCIe device allocation requests to the platform manager through the NTB's doorbell and scratchpad mechanism as memory write operations against the NTB. The NIC device driver running on the management host is an Intel ixgbe version 3.7.14 PF driver, whereas the NIC device driver running in each VM is an Intel ixgbevf version 2.4.0 VF driver.

The detailed steps required to set up a virtual PCIe device for a newly started VM from a compute host are shown in Figure 4 and explained as follows:

1. The hypervisor of the requesting compute host informs the platform manager the CSR, MMIO, MSI-X and DMA payload memory address ranges of the virtual PCIe device, which are in the compute host's machine physical address space.

2. The platform manager allocates a VF from an SRIOV physical PCIe device, in this case the Intel NIC, and identifies this VF's CSR, MMIO, MSI-X, and DMA payload memory address ranges, which are in the management host's machine physical address space.

3. Using the NTB's translation register and the management host's IOMMU, the platform manager sets up translations that map from the virtual PCIe device's CSR and IOMMU address ranges to the CSR and MMIO address ranges of the corresponding VF.

4. Using the NTB's translation register and the compute host's IOMMU, the compute host's hypervisor sets up translations that map from the VF's MSI-X and DMA payload address ranges to the MSI-X and DMA payload address ranges of the corresponding virtual PCIe device.

5. The platform manager sets up the NTB's device ID translation tables so that the virtual PCIe device's ID could be mapped to the a legitimate device ID in the management host's domain, and the VF's ID could be mapped to a legitimate device ID in the compute host's domain.

6. The requesting compute host's hypervisor programs the extended page table properly and informs the new VM of the guest physical addresses for the CSR, MMIO, MSI-X and DMA payload areas associated with the requested virtual PCIe device.

The first version of *Ladon* prototype assumes that each VM allocates a dedicated DMA payload area for each virtual PCIe device installed, because modern BIOS allocates from the 3-4GB portion of the machine physical address space memory areas assigned to PCIe devices[1]. In the context of a virtual NIC, this dedicated DMA payload area corresponds to the virtual NIC's transmission and reception buffer. Even though the socket buffers of a VM's OS could reside in any parts of the OS's kernel heap, an outgoing packet's payload has to be copied from its original socket buffer to this dedicated DMA payload area before it can be DMAed to the actual NIC backing up the virtual NIC; similarly, an incoming packet's payload is first DMAed into the dedicated DMA payload area and has to be copied to a socket buffer before it can be processed by the VM's protocol stack. Currently, each DMA payload area is 4MB. When a compute host requests multiple virtual PCIe devices, the DMA payload areas for these devices are assigned contiguous address ranges in the compute host's machine physical address space so that only one BAR is needed in the NTB to map them into the management host's machine physical address space. This is essential because each NTB only has 4 available BARs.

This additional data copying step in packet transmission and reception incurs an additional performance overhead and requires a modification to the NIC driver running inside the VMs. In the next section, we present the optimization design of *Ladon*, which eliminates the additional data copying restriction due to the BIOS and allows native device driver to run in VMs.

[1]The BIOS restricts the MMIO space to be 3-4GB so that a VF can only DMA to/from this 3-4GB region, which incurs an extra packet copying operation.

6. ZERO-COPY OPTIMIZATION

A solution to the data copying overhead between the socket buffer and the dedicated DMA area is to map the *entire* physical memory address space of a CH to the MH's physical memory address space. This requires setting up a pair of BARs of the link side of the NTB connecting MH with the CH, and make the coverage of the BAR pair larger than the CH's physical memory address space, usually larger than 8GB. However, modern BIOS allocates the MMIO addresses from the 3GB-4GB range, which makes it impossible to set up BARs with large address range. *Ladon* solves this problem by disabling the pair of BARs to be used to map a CH's physical address space at BIOS probing time and enables them after the kernel takes over from BIOS. Once the MH's kernel is up, *Ladon* finds an unused physical address range in its physical address space, assigns it the BAR pair, and sets up a map that associates this address range in MH's physical address space and the CH's physical address space. Given that today's 64-bit x86 systems have at least 36 bit usable physical address space or 64GB, there are plenty of unused physical address ranges available.

Ladon further intercepts the DMA API in compute host's kernel so that when a VF driver in a VM running on a CH requests for the DMA target address corresponding to a given socket buffer address, the DMA API function returns the proper host physical address in MH's physical adddress space by taking into account the mappings associated with MH's BAR pair covering the CH's physical address space and with the IOMMU at the virtual side of the NTB connecting MH with the CH. For example, a network driver in a VM running on a CH intends to use a host physical address in CH's physical address space, 0x17A2D000, as the target address of a receive buffer. However, it needs the help of the hypervisor to translate this guest physical address into a host physical address in MH's physical address space, say 0x40BD476000, and sends it to the VF assigned to the VM. When a network packet arrives at the assigned VF, it issues a DMA write operation to copy the packet to the MH host physical address 0x40BD476000. The link-side BAR pair of the NTB covering the CH's physical address space

takes this and translates it to a virtual device address, say, 0xBD476000, and the IOMMU of the NTB's virtual side further translates the virtual device address to the CH host physical address 0x17A2D000. With this design, no modification to the VF driver is required and no extra data copying is needed because the DMA engine on the NIC could access any portion of a VM's memory directly.

This optimization does not introduce additional security risk as long as the management host supports IOMMU. Because the management host's IOMMU is inaccessible to any VM running on any CH, no VM could access any portion of MH's physical address space without authorization, and by deduction, any CH's physical address space without authorization.

7. PERFORMANCE EVALUATION

The key value of *Ladon* is its ability to enable a VM running on one host to directly access an SRIOV PCIe device controlled by a remote host securely and efficiently. To demonstrate the efficiency of *Ladon*, we measure the network packet latency and throughput of a network connection that is set up in the following three configurations and compare them:

- SRIOV: The network connection is between a VM running on the management host and the remote test host.
- MRIOV: The network connection is between a user VM running on a compute host and the remote test host,
- MRIOV+: The network connection is between a user VM running on a compute host and the remote test host, using the zero-copy optimization.

To the MH, the Intel NIC is a physical function or PF. To a CH and a VM on a CH, the Intel NIC is a virtual function or VF. Because the Intel NIC does not allows VFs to use jumbo frames [10], we ran all our experiments without jumbo frames. For the SRIOV driver, we enabled the Generic Receive Offload (GRO) mechanism, transmit and receive checksum offload, TSO (TCP Segmentation offload) and New API (NAPI). In these experiments, we ran the KVM hypervisor on the compute host, and each VM on the compute host is equipped with 2GB RAM and a 3.4GHz CPU and running native Linux kernel 2.6.38 with a virtual PCIe device assigned to it. To minimize the VM-level context switching overhead, each VM is pinned down to a dedicated CPU core.

In addition, we analyze the fairness associated with network resource sharing when multiple network connections that are in the same or different configurations use the same physical NIC simultaneously.

7.1 Raw NTB Bandwidth

The key building block in *Ladon* is the NTB, whose raw bandwidth strongly affects the performance of the virtual PCIe devices exposed to a VM. To obtain an accurate picture of the NTB's raw transfer capability, we measured the throughput of a memory copy operation from the management host to a compute host. When we ran a program on the management host to read and write a 4MB area on a compute host, the average measured read throughput and write throughput over 100 runs are 11.99MB/sec and 11.4MB/sec, respectively. However, when we used the DMA engine inside the NTB to read and write the same 4MB area, the average read throughput and write throughput jump to 3.69GB/sec and 3.44GB/sec, respectively. The dramatic difference between the two originates from the fact that each iteration of the memory copying loop copies 4 bytes of data at a time and worse yet CPU caching is turned off because the remote host's memory is treated as a PCIe device. In contrast, the average PCIe packet payload size in the DMA case is 128 bytes. In any case, the measured throughput of DMA-based memory read/write operations shows that the NTB's raw transfer bandwidth is at par with the front-side memory bus bandwidth.

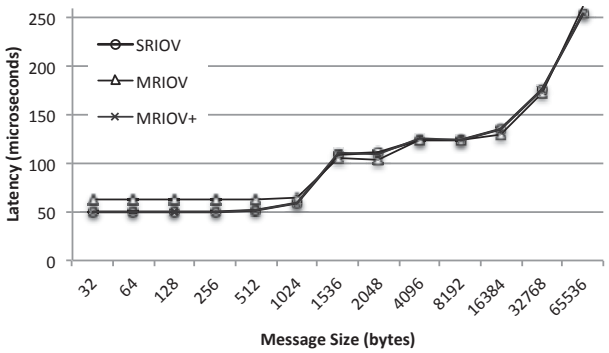

Figure 5: *One-way packet latency between a VM on a compute host and the test host, between a VM on the management host and the test host, when the packet size is varied from 32 bytes to 64 Kbytes.*

7.2 Latency Results

To measure the packet latency, we used the NetPIPE benchmarking tool [22], which employs ping-pong tests to measure the half round-trip time between two servers. Figure 5 shows the latency measurements reported by NetPIPE when the message size is varied from 32 bytes to 64 Kbytes.

When the packet size is smaller than or equal to 512 bytes, the packet latency of the MRIOV configuration, MRIOV+ configuration, and that of the SRIOV configuration remain largely the same, because the per-packet overhead dominates the packet latency, and therefore the packet latency is largely independent of the packet size. However, as the packet size is increased to 1024 bytes and beyond, the per-byte overhead starts to dominate the packet latency and accordingly the packet latency increases with the packet size.

When the packet size is smaller than or equal to 512 bytes, the packet latency of the MRIOV configuration is noticeably higher than that of the SRIOV configuration because, due to BIOS limitation, the MRIOV configuration requires an additional data copying step to move the packet payload to the DMA area. However, this latency difference essentially disappears when the packet size is large than or equal to 1024 bytes because it is over-shadowed by other components in the end-to-end packet latency. Of all the packet sizes, the packet latency of the MRIOV+ configuration shows no noticeable differences than that of the SRIOV configuration.

7.3 Throughput Results

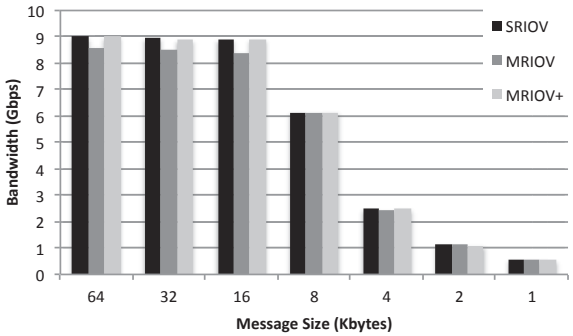

Figure 6: *Measured throughput (Gbps) of a network connection whose receiver is a VM on the management host and a VM on a compute host.*

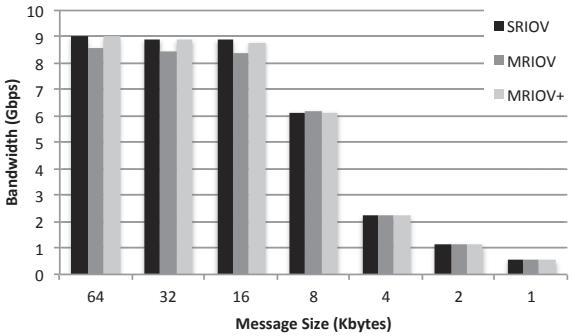

Figure 7: *Measured throughput (Gbps) of a network connection whose sender is a VM on the management host and a VM on a compute host.*

We measured the throughput of a TCP connection in SRIOV, MRIOV, and MRIOV+ configuration using Iperf [25]. For each measurement, we ran Iperf for 300 seconds and take the average of the highest 90% measurements reported by Iperf.

For MRIOV, when a packet destined to a VM on a compute host arrives at its corresponding actual NIC, the NIC DMAs the packet via an NTB to a reception buffer, generates an interrupt, whose handler copies the packet out of the reception buffer into a socket buffer, registers a software interrupt, and finally runs the software interrupt handler to pass the packet to the upper layer of the network protocol stack for further processing. Exactly the same sequence happens when the destination VM is on the management host, except the DMA operation does not cross an NTB and there is no data copying step. The receive throughput is affected by the DMA operation's efficiency, the interrupt overhead, the per-packet higher-layer protocol processing cost, and optionally the data copying cost. NAPI cuts down the interrupt overhead by using polling to aggregate multiple hardware interrupts when the incoming interrupt rate is too high. Generic receive offload (GRO) reduces the protocol processing overhead by aggregating multiple incoming Ethernet packets into a larger IP packet before passing it to the TCP/IP protocol stack. Because the jumbo frame support is turned off, the DMA operation's efficiency is largely unaffected by the packet size.

Figure 6 shows the receive bandwidth of a VM running on the management host, and a VM running on a compute host, when the remote test host serves as the sender, and the packet size is varied from 1KB to 64KB. When the packet size is equal to or smaller than 8KB, there is no difference between the receive bandwidth of the SRIOV configuration and that of the MRIOV configuration because smaller packet size limits the extent of GRO's aggregation and makes the protocol processing step the bottleneck, so much so that the overhead associated with the extra data copying step in the MRIOV configuration is masked. However, when the packet size is equal to or larger than 16KB, the protocol processing step's efficiency is significantly improved and no longer a bottleneck, and the overhead associated with the extra data copying step in the MRIOV configuration, which is largely unaffected by the packet size, now matters. For example, when the packet size is 64KB, the SRIOV configuration takes 26 μs to complete the protocol processing of 64 packets, while the MRIOV configuration takes 45μs to do the same. As a result, the receive bandwidth of the MRIOV configuration when the packet size is 16KB, 32KB and 64KB are 4.51%, 5.13%, 5.63% smaller than those of the SRIOV configurations, respectively. The receive bandwidth difference between the SRIOV and MRIOV configuration increases with the packet size, as the protocol processing step becomes more efficient with the increasing message size. With the optimized design, the received bandwidth between SRIOV and MRIOV+ configuration shows almost no difference, as the socket buffer copying overhead has been removed.

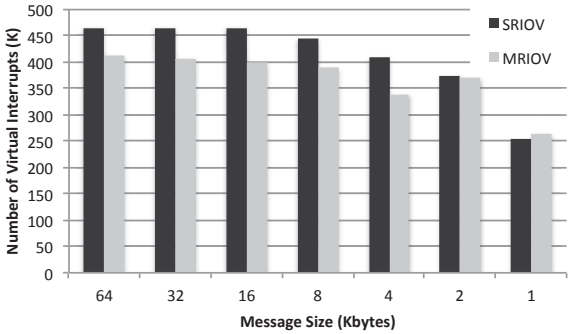

Figure 8: *Number of virtual interrupts in a 60-second TCP connection delivered from the hypervisor to the target VM.*

The transmit bandwidths of the SRIOV and MRIOV configuration exhibit a similar trend with the varying packet size, as shown in Figure 7, while the optimized MRIOV+ configuration also exhibits no overhead compared to the SRIOV configuration.

7.4 Interrupt Count

When a packet arrives at an NIC, it generates a hardware interrupt to the management host's hypervisor in the SRIOV configuration and the compute host's hypervisor in the MRIOV configuration, and then the hypervisor delivers a virtual interrupt to the packet's target VM. We measured the number of hardware interrupts received by the hypervisor and the number of virtual interrupts received by the

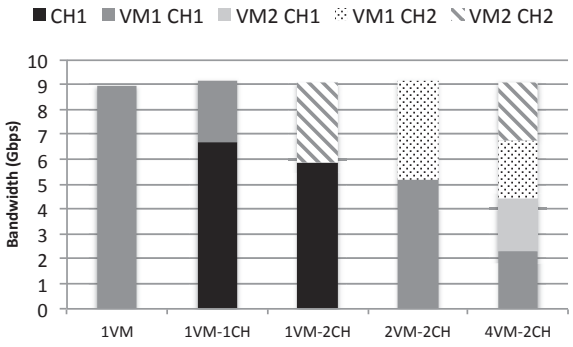

Figure 9: *Receive bandwidth comparison among different types of target hosts or VMs.*

target VM in a 60-second TCP connection under different packet size. In each of two test configurations, the number of hardware interrupts is almost the same as the number of virtual interrupts, suggesting that the hypervisor does not perform further interrupt coalescing. However, the number of virtual interrupts in the MRIOV configuration is smaller than that in the SRIOV configuration, as shown in Figure 8 when the packet size is larger than 2KB.

This discrepancy is due to the NAPI mechanism in the Linux-based target VM. NAPI allows a network device driver to run in either polling mode or interrupt mode. When a hardware interrupt handler is invoked and it processes K incoming packets, NAPI disables the NIC interrupt and uses periodic polling to process incoming packets. While in the polling mode, if the number of packets processed in one invocation falls below K, NAPI enables the NIC interrupt and processes incoming packets in an interrupt-driven fashion. The default value of K is 64 in our experiments. Because the hardware interrupt handler of the MRIOV configuration incurs an additional data copying overhead, its packet processing rate is lower and therefore it is more likely to operate in the polling mode. Consequently, the number of virtual interrupts the target VM receives is smaller in the MRIOV configuration than in the SRIOV configuration.

7.5 Fairness

In this subsection, we assess the degree of fairness when multiple TCP connections, each with the same sender but different types of receivers, share the same physical NIC under the MRIOV+ configuration. We consider the following five cases:

- 1VM: one receiver, which is a VM running on a compute host,
- 1VM-1CH: two receivers, one of which is a compute host and the other of which is a VM running on the same compute host,
- 1VM-2CH: two receivers, one of which is a compute host and the other of which is a VM running on a different compute host,
- 2VM-2CH: two receivers, one of which is a VM running on one compute host and the other of which is a VM running in a different compute host, and
- 4VM-2CH: four receivers, two of which are two VMs running on one compute host and the other two of which are two VMs running on a different compute host.

Table 1: Power Conumption Comparison

Technology	Switch (W)	Adapter (W)	Total (W)
Ethernet[1]	176	5.5 × 16	264
Infiniband[2]	34	15.5 × 16	282
Ladon[3]	22.73	5.5 + 2 × 16	60.23

[1] 176W: Brocade TurboIron switch, 5.5W: Intel 82599 NIC.
[2] 34W: Mellanox InfiniScale III switch, 15.5W: Mellanox ConnectX MHQH19.
[3] 22.73W: PLX 8696 switch, 2W: PLX 8619 NTB adpater.

For each case, we measured the receive bandwidth of a TCP connection that runs for 300 seconds and the result is shown in Figure 9. The aggregated receive bandwidth when multiple VFs are used is slightly higher than that when only one VF is used, which could reach around 9 Gbps. The results corresponding to the 1VM-1CH and 1VM-2CH case show that a compute host commands a higher share of the NIC's raw bandwidth than a VM running on the same or a different host, presumably because a VM incurs an additional virtual interrupt overhead compared with a host. The results corresponding to the 2VM-2CH and 4VM-2CH case show that the VMs on the same or different compute hosts are granted an approximately equal share of the NIC's raw bandwidth.

7.6 Power Consumption

We compare *Ladon* with existing blade servers in terms of network component expenses and power consumption. Consider a cluster design with 16 servers, each of which is equipped with a 10GE NIC and connected to a top-of-rack 10GE switch. Using *Ladon*'s I/O sharing architecture immediately saves the cost of 15 NICs by allowing the 16 servers to share the 10GE NIC via the PCIe switch fabric. The top-of-rack 10GE switch is replaced by the standard PCIe switch, which connects the 16 servers and the built-in layer-2 switch on the shared 10GE NIC, and forwards packets among these 16 servers.

The benefits from reducing the number of I/O devices include not only reduced cost but also reduced electricity usage or power, cooling, as well as space requirement. Table 1 lists the power consumption of *Ladon*, compared with Ethernet-based and InfiniBand-based clustering technology. We compare the maximum power consumption of a 24-port Ethernet, InfiniBand, and PCIe switch, and their corresponding adapters. While the switch dominates the power consumption in Ethernet-based solutions and the adapters dominates in InfiniBand-based solutions, *Ladon* proves to be a more competitive solution because its I/O sharing and consolidation design affords at least 4 times reduction in power consumption.

8. RELATED WORK

PCI-SIG and Patents: PCI-SIG MRIOV introduces the concept of virtual PCIe hierarchy, which extends host's PCIe domain through the fabric to each PCIe endpoint. An MRIOV-capable switch with virtual fan-out routes PCIe packets to each host port without modifying the original

packets. A fabric manager, called MR-PCIM (Multi Root PCI Manager), is also required, which coordinates sharings of I/O devices. However, the adoption of MRIOV requires modification both in hardware and software and MR-aware endpoint is not available now and may not be available in a near feature. As a result, lots of attempts have been made to achieve multi-root sharing using SRIOV compliant I/O devices. Without MRIOV-capable switch, many research efforts focus on using NTB as solution to separate two PCIe domains and leverage NTB's address translation mechanism to access the shared I/O devices.

D. Riley [20] proposes using NTB to redirect a shared device's CSR from the management host to the requested compute host. K. Malwankar [15] provides multiple proxy devices on a shared PCIe fabric. The proxy device can be associated with a real I/O device by copying the configuration space of the real device to the proxy device. J. Regula [19] allocates pseudo physical functions to the compute host from virtual functions of SRIOV devices enumerated by the management domain. Although modern device supports MSI-X, most of the existing proposals deliver the shared device's interrupt using doorbell registers [11]. Unlike *Ladon* which maps the MSI-X addresses to NTB's BAR and delivers as a memory write, mapping MSI-X interrupts to the limited number of doorbell registers could incur higher latency and scale poorly. While the limitations and performances are unknown from these patent descriptions, *Ladon* provides detailed design of the concept and achieves native performance as existing SRIOV system without modifying the device driver.

Products: Proprietary hardware-assisted solutions includes Dolphin's DX interconnect [14]. Dolphin's system provides both the MRIOV capable switch and software stacks for I/O device sharing as well as clustering. Other than leveraging PCIe, other approaches such as ExpEther [24] connects multiple I/O devices using standard Ethernet and its ExpEther virtual PCIe switch enables each compute host to extend its PCIe bus over Ethernet. Besides, there are also numerous I/O consolidation appliances [3, 2] in the market proposing I/O device sharing on top of InfiniBand or FCoE. Comparatively, *Ladon* demonstrates an all-software approach built on top of standard PCIe switch to support the MRIOV functionality.

Security: In a single host virtualized environment, the problem of direct device assignment for untrusted virtual machines and its protection strategies have been well discussed [27, 6]. Willmann et al. [26] present four policies for creating IOMMU mapping and showed their corresponding performance numbers. *Ladon* implements direct mapping, which is transparent to the guest and incurs less performance overhead. [21] proposes a scalable and secure I/O system that virtualizes the entire I/O subsystem and separates the I/O execution from the untrusted VM. An I/O device is attached to a device controller in an IP or InfiniBand network. The I/O becomes a service in a network and I/O consumers or VMs get access by using its library at higher level of abstraction. However, with this secure but indirect I/O access, the overhead of applying another layer may not be able to meet the high I/O performance requirement of today's data-centric workload.

Although current *Ladon* prototype is based on Intel x86 architecture, most of the required components could also be found in AMD's virtualization technology, i.e., AMD-V's IOMMU and NPT. Depending on PLX's NTB devices may be a limitation of *Ladon*, however, given the fact that some of today's Intel Xeon processors [23] support NTB and its driver is planning to be included into mainline Linux kernel [1], we expect that NTB will become more popular in the near future.

9. CONCLUSION

Traditional I/O device virtualization allows VMs on a server to share physical I/O devices in a secure and efficient way. The *Ladon* system described in this paper takes one step further to enable VMs running on multiple servers to share in a secure and efficient way a common set of I/O devices that are under the control of a management host. This multi-host I/O device sharing capability enables cluster-wide I/O device consolidation and greatly simplifies the system interconnect complexity. This architecture decouples a server's process/memory from its I/O devices, and opens up many new design possibilities for modular compute clusters. More concretely, the research contributions of this work include:

- A PCIe-based system interconnect architecture that connects together multiple servers and a set of PCIe devices and allows the servers to share and directly access these devices in a way that is consistent with the original single-root PCIe architecture,
- A secure I/O device sharing scheme that leverages the address/ID translation schemes, such as EPT, BAR translation registers, IOMMU, LUT, etc. to prevent unauthorized accesses to shared I/O devices, and
- A fully operational prototype that successfully demonstrates the feasibility and efficiency of the MRIOV capability of *Ladon*and its optimized design, with a throughput penalty of less than 6% for MRIOV and virtually no throughput/latency penalty for MRIOV+, when compared with SRIOV.

We are currently working on adding fail-over support for the management host (MH) of a *Ladon* cluster, which is not supported in the current *Ladon* prototype. Because MH is not involved in the interactions between VMs and their assigned virtual I/O interfaces, in theory the interactions should not be disrupted by a failure of MH. In practice, however, the PCIe architecture dictates a reset of all the PCIe devices when their root complex dies. Therefore, how to minimize the fail-over latency associated with an MH failure is an interesting technical challenge.

10. REFERENCES

[1] Intel Provides Linux PCI Express NTB Support. `http://www.phoronix.com/scan.php?page=news_item&px=MTE0MDE`. accessed Nov 4, 2012.

[2] I/O Consolidation White Paper, NextIO, Inc. `http://www.nextio.com/resources/files/wp-nextio-consolidation.pdf`. accessed June 4, 2012.

[3] Micron I/O Virtualization White Paper, Micron Technology, Inc. `http://www.micron.com/~/media/Documents/Products/White%20Paper/micron_io_virtualization_wp.pdf`. accessed May 4, 2012.

[4] Multi-Root I/O Virtualization and Sharing 1.0 Specification, PCI-SIG, 2008.

[5] Single-Root I/O Virtualization and Sharing Specification, Revision 1.0, PCI-SIG, 2008.

[6] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. Van Doorn, J. Nakajima, A. Mallick, and E. Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. In *OLS06: The 2006 Ottawa Linux Symposium*, pages 71–86, 2006.

[7] J. Byrne, J. Chang, K. Lim, L. Ramirez, and P. Ranganathan. Power-Efficient Networking for Balanced System Designs: Early Experiences with PCIe. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, page 3. ACM, 2011.

[8] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. High Performance Network Virtualization with SR-IOV. *Journal of Parallel and Distributed Computing*, 2012.

[9] Y. Dong, Z. Yu, and G. Rose. SR-IOV Networking in Xen: Architecture, Design and Implementation. In *USENIX Workshop on I/O Virtualization (WIOV)*, pages 10–10, 2008.

[10] P. Dykstra. Gigabit Ethernet Jumbo Frames. *Relation*, 10(1.19):2630, 1999.

[11] A. Griggs. Method and System to Improve the Operations of an Integrated Non-Transparent Bridge Device, June 2 2010. US Patent App. 12/792,490.

[12] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Advanced Configuration and Power Interface Specification, 2004.

[13] R. Hiremane. Intel Virtualization Technology for Directed I/O (Intel VT-d). *Technology@ Intel Magazine*, 4(10), 2007.

[14] V. Krishnan. Towards an Integrated IO and Clustering Solution Using PCI Express. In *Cluster Computing, 2007 IEEE International Conference on*, pages 259–266. IEEE, 2007.

[15] K. Malwankar, D. Talayco, and A. Ekici. PCI-Express Function Proxy, Oct. 1 2009. WO Patent WO/2009/120,798.

[16] D. Mayhew and V. Krishnan. PCI Express and Advanced Switching: Evolutionary Path to Building Next Generation Interconnects. In *High Performance Interconnects, 2003. Proceedings. 11th Symposium on*, pages 21–29. IEEE, 2003.

[17] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, 2006.

[18] J. Regula. Using Non-Transparent Bridging in PCI Express Systems. *PLX Technology, Inc*, 2004.

[19] J. Regula. Multi-Root Sharing of Single-Root Input/Output Virtualization, Dec. 28 2010. US Patent App. 12/979,904.

[20] D. Riley. System and Method for Multi-Host Sharing of a Single-Host Device, May 8 2012. US Patent 8,176,204.

[21] J. Satran, L. Shalev, M. Ben-Yehuda, and Z. Machulsky. Scalable I/O-a Well-Architected Way to Do Scalable, Secure and Virtualized I/O. In *Proceedings of Workshop on I/O Virtualization*, 2008.

[22] Q. Snell, A. Mikler, and J. Gustafson. Netpipe: A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, volume 6, 1996.

[23] M. J. Sullivan. Intel Xeon Processor C5500/C3500 Series Non-Transparent Bridge. *Technology@ Intel Magazine*, 2010.

[24] J. Suzuki, Y. Hidaka, J. Higuchi, T. Baba, N. Kami, and T. Yoshikawa. Multi-Root Share of Single-Root I/O Virtualization (SR-IOV) Compliant PCI Express Device. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 25–31. IEEE, 2010.

[25] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The TCP/UDP Bandwidth Measurement Tool. *http://dast.nlanr.net/Projects*, 2005.

[26] P. Willmann, S. Rixner, and A. Cox. Protection Strategies for Direct Access to Virtualized I/O Devices. In *USENIX Annual Technical Conference*, pages 15–28, 2008.

[27] B. Yassour, M. Ben-Yehuda, and O. Wasserman. Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines. Technical report, Technical Report H-0263, IBM Research, 2008.