Seamless Fail-over for PCIe Switched Networks

William Cheng-Chun Tu
VMware Inc.
u9012063@gmail.com

Tzi-cker Chiueh
Industrial Technology Research Institute
tcc@itri.org.tw

ABSTRACT

PCI Express (PCIe) was originally designed as a *local bus* interconnect technology for connecting CPUs, GPUs and I/O devices inside a machine, and has since been enhanced to be a full-blown switched network that features point-to-point links, hop-by-hop flow control, end-to-end retransmission, etc. Recently, researchers have further extended PCIe into an intra-rack interconnect designed to connect multiple hosts within the same rack. To viably apply PCIe to such use cases , additional fail-over mechanisms are needed to ensure continued network operation in the presence of control plane and data plane failures. This paper presents the design, implementation and preliminary evaluation of a fault-tolerant PCIe-based rack area network architecture called *Ladon*, which incorporates a fail-over mechanism that takes effective advantage of PCIe architectural features to significantly reduce the service disruption time due to a control plane failure of a PCIe switch. Empirical tests on an operational *Ladon* prototype show that the proposed mechanism ensures that a PCIe root complex failure has *zero* impact on the data plane and incurs only a modest disruption time (less than 40 sec) for the control plane services.

1 INTRODUCTION

Peripheral Component Interconnect Express (PCIe) [9] is a computer expansion standard developed by the PCI Special Interest Group (PCI-SIG) and is designed to serve as a motherboard-level interconnect for on-board devices, a passive backplane interconnect among boards, and an expansion interface for connecting a machine with external devices such as storage boxes. A PCIe network is a switched network with serial point-to-point full duplex *lanes*, where each attached PCIe device is connected to the network through a *link* consisting of one or multiple lanes. The set of devices connected to a PCIe network form a PCIe domain, with one of them serving as the *root complex*, which is responsible for enumerating and configuring all other devices in the domain and is usually connected to the CPU.

Recently, researchers have proposed to replace a top-of-rack (TOR) Ethernet switch with a PCIe switch, which allows all servers in a rack to share all I/O devices in the rack [1, 4, 5, 7, 27] *and* to communicate with one another directly over PCIe links [10, 16, 28]. To convert PCIe into a viable system interconnect for inter-host communications within a rack, one must first enable multiple hosts to sit on the same PCIe network and directly communicate with each other. Although the Multi-Root IO Virtualization (MRIOV) standard [5], published in 2008, was proposed to support multiple root complexes in a single PCIe domain, no commercial implementations of true MRIOV switches or devices exist and there is no sign that there will ever be in the near future. Another approach [27, 28] to establishing direct PCIe connectivity among multiple hosts is to leverage a special type of PCIe device called *non-transparent bridge* (NTB) [21, 24], which is designed to enable one PCIe domain to directly access resources in another PCIe domain without the involvement of the latter's root complex.

Although PCIe is designed to be lossless at the transaction layer with a flow-control and retransmission mechanism [9, 19], it does not support any high-level fault tolerance mechanisms that guarantee a PCIe network's service availability to the extent which rivals that of traditional system interconnect technologies such as Ethernet, Infiniband and Fiber Channel. *Ladon* [27, 28] is a PCIe-based intra-rack interconnect technology that supports multi-host connectivity using NTB and provides a seamless fail-over mechanism that effectively addresses the following challenges imposed by the PCIe architecture:

- Because a PCIe domain's root complex is solely responsible for the configuration of all the devices in the domain, it represents a single point of failure in the domain's control plane.
- Because all devices in a PCIe domain form a hierarchical tree whose root is the root complex, it is inherently difficult to provision redundancies and support multiple paths between any two devices in the same PCIe domain.

SYSTOR '18, June 4–7, 2018, HAIFA, Israel

ACM ISBN 978-1-4503-5849-1/18/06…$15.00
https://doi.org/10.1145/3211890.3211895

- Since PCIe packet routing is based on target device addresses and PCIe device addresses cannot be modified dynamically, it is thus difficult to dynamically re-route PCIe packets when link/port failures occur.

This paper addresses the above fault tolerance challenges associated with a rack-scale PCIe switch-based network that assumes a SDN-like architecture, i.e., a central controller (root complex) controlling the operations of a number of packet-forwarding PCIe switch chips. To handle a root complex failure, *Ladon* enforces a clean separation of control plane from data plane, and augments the *virtual switch* mechanism with a device driver state saving and restore mechanism so that a slave root complex could seamlessly take over after the master root complex dies. To enable continued operation of a PCIe network in the event of a single PCIe link/switch failure, *Ladon* partitions a PCIe domain into two physically disjoint subdomains, connects every host in a rack to two PCIe end points, each of which belongs to a separate subdomain, assigns two distinct address ranges to each host, and thus makes each host accessible through either one of these two address ranges.

2 RELATED WORK

Multi-server interconnection technologies such as Fiber Channel, InfiniBand, and Ethernet play a critical role in today's HPC design. Traditionally, out of motherboard-level application of PCIe is used as an I/O expansion switch, which allows a system to support a larger number of PCIe I/O devices where a single motherboard cannot accommodate. In the blade server design, multiple servers can connect to one PCIe expansion switch, but requires creating an isolated PCIe domain for each server by re-partitioning a set of PCIe ports to each domain. As a result, each PCIe domain is an independent network without direct communication through the expansion switch. Recently, several efforts have been proposed to extend the use of PCIe as a high performance and low power system interconnect solutions [2, 10, 16, 17, 26].

John Byrne et al. [10] shows a PCIe-based network based on NTB to demonstrate performance gains and 60-124% better energy efficiency. PEARL [26] demonstrates a power-aware PCIe network and addresses the reliability of the root complex failure by designing their proprietary communication chip. Memory Channel [12, 13] relies on the reflective memory technology to build a cluster-wide shared memory network. A cluster-wide memory address space is introduced and for each node, a portion of its memory is mapped to the global address space. RONNIEE Express Fabric [7] is an in memory PCIe network for storage systems, which enables remote memory access between hosts through PCIe and delivers high performance and scalable file system implementations. As a comparison, *Ladon* focuses the reliability issues resulted from extending PCIe into an intra-rack interconnect in a multi-host PCIe system.

Without MR-aware PCIe switch, using a PCIe domain isolation and address translation device such as NTB becomes a de facto solution to interconnecting multiple PCIe domains. D. Riley [22] proposes using NTB to redirect CSR for shared I/O devices. The redirected CSR requests are handled by the management host on behalf of the compute host in order to control the I/O devices. Similarly, K. Malwankar. [18] 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. As a result, NTB devices are becoming prevalent as an external devices [21, 23] or embedded in the CPU feature [24].

A server's particular PCIe bridge connecting to another server can be pre-configured as either in EP (End Point) mode or RC mode (Root Complex), where the RC-mode server own the PCIe switch's domain and the rest of servers run in EP mode. Several patents have proposed adding failure detection and cable redundancy for multi-host PCIe network. [20] describes solving the master RC failure problem by enabling all the other RCs to periodically detect the liveliness of master RC, and elect the new RC using system-wise timer. [11] presents a switch fail-over control mechanism by maintaining the primary/secondary device table entry (DTE) for the RC and marks the secondary DTE as passive. Upon a fail-over condition, updating the secondary DTE in the device table as an active entry and forming a fail-over path to enable traffic rerouting. *Ladon* is a software-based solution that leverages the existing commodity, off-the-shelf components to ensure continued network operation in the presence of any single control plane and data plane failure.

3 PCI EXPRESS ARCHITECTURE

3.1 Overview

PCIe is a layered protocol consisting of a physical layer, a data link layer and a transaction layer. The transaction layer is responsible for converting high-level PCIe transactions issued by the OS or firmware, i.e., *memory, I/O, configuration* and *message*, into PCIe transaction layer packets (TLP). An X86 machine hosts a PCIe domain, with the *root complex* residing in the north bridge, which connects CPU, memory and the PCIe network, and implements the transaction layer protocol. Each PCIe device in a PCIe domain is uniquely identified by a bus/device/function ID, and is given a set of *configuration space registers* (CSR), which are partitioned into a standardized part (such as device/vendor ID, command, *base address register* or BAR, etc.) and a device-specific part. A PCIe device occupying a PCIe slot functions as one or multiple physical functions (PF), each acting like a logical

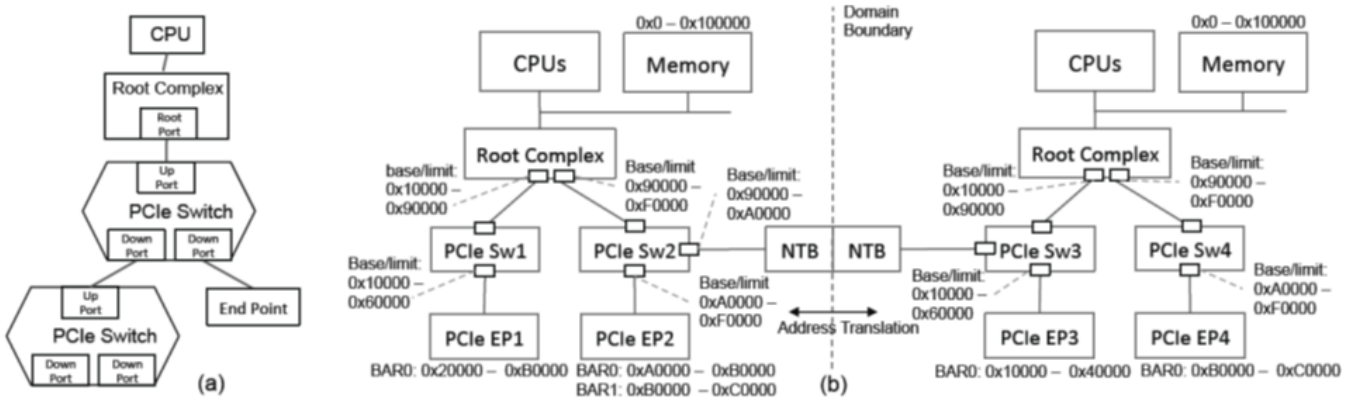

Figure 1: *(a) A PCIe domain consists of multiple PCIe switches, each with one* Up *(upstream) port and multiple* Down *(downstream) ports, and end-points in a PCIe domain form a tree, with its root being the domain's root complex. (b) A non-transparent bridge (NTB) connects two PCIe domains, where each domain has its own independent address space and is able to access the other domain's resources through the NTB's address translation facility.*

PCIe device with its own device ID and CSR. The PCI-SIG standard on single-root I/O virtualization (SRIOV) [6] virtualizes a PCIe device into multiple virtual PCIe devices. SRIOV distinguishes between a physical function (PF) and a virtual function (VF). A PF is a full-function PCI device with its ID and complete CSR, whereas a VF is a lighter-weight PCIe device that comes with its ID but without a complete CSR.

Although PCIe was originally designed to be a system backplane, it has now evolved to be able to scale to a large number of endpoints and PCIe switches, as shown in Figure 1(a). Two types of devices exist in a PCIe domain's tree: the Type 0 endpoint and Type 1 transparent bridge (TB). When a system boots up, the BIOS or the OS's PCIe driver constructs a PCIe domain's tree by relying on the root complex to recursively scan and enumerate each PCIe device in a level-by-level fashion.

3.2 Address-Based Routing

Every operation in a PCIe domain is expressed in terms of a memory read or write operation with a target address, which is translated into a PCIe network packet (TLP) that is then routed to the PCIe endpoint whose BAR covers the target address. More concretely, every end-point in a PCIe domain is responsible for a specific range of the domain's physical memory address space, which consists of the end point's configuration space and the memory address ranges specified in its BARs, as shown in Figure 1(b).

A PCIe switch consists of a set of Type 1 PCI-to-PCI bridges, which are shown in Figure 1(a) as the Up (or upstream) port and Down (or downstream) ports. The memory address range of an upstream port should cover the union of the memory address ranges associated with its downstream ports. In turn, each downstream port of a PCIe switch consists of a base and a limit register, which specifies a memory address range that covers the union of the address ranges associated with the devices reachable via that downstream port. Packets are routed within a PCIe switch by matching their target device's address with the memory address range of each switch port. The single-tree PCIe topology guarantees there is only one path between any two end-points in the tree, and the strictly hierarchical routing makes it difficult to change the bridges' routing table entries dynamically upon any failure, because changing one routing entry at a downstream port implies possibly changing all switches and endpoints in its subtree.

For example, in Figure 1 (b), the BIOS first probes possible PCIe end-points, then assigns physical address ranges to BARs of discovered endpoints (EP1 to EP4), and finally configures the routing table entries of each intermediate bridge, by ensuring that the base/limit register of each bridge's downstream port covers the physical address ranges of all the end-points reachable through that port. Specifically, the PCIe Switch2 has one upstream port and two downstream ports pointing down and right. For a memory access with a target address of `0xB0040`, the root complex first routes the associated TLP to the right port toward PCIe Sw2, since its base/limit registers are `0x90000 - 0xF0000`, which covers `0xB0040`. Because the downstream port connecting EP2 covers the address range `0xA0000` to `0xF0000`, the TLP is then routed to the EP2 and eventually to EP2's BAR1.

3.3 Non-Transparent Bridge

In the PCIe architecture, at any point in time every PCIe domain has exactly one *active root complex*. Therefore, in

theory, no two servers each with their own CPUs are allowed to co-exist in the same PCIe domain. *Non-transparent bridge* (NTB), which is a part of the PCI-SIG standard, is designed to enable two or more PCIe domains to inter-operate together as if they are in a single domain, and thus makes a key building block for *Ladon*. Conceptually, 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 one of the two PCIe domains it adjoins. These two end-points each expose a Type 0 header type in the CSR and thus are discovered and enumerated by the root complex of their respective PCIe domain in the same way as an ordinary PCIe end-point. However, an NTB provides additional *memory address translation*, *device ID translation* and *messaging* facilities that allow the two PCIe domains to work together while keeping them logically isolate from each other. 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 is reachable via an NTB port. The side of an NTB port that is connected to a slave domain is the *virtual* side, whereas the other side is called the *link* side.

The magic of an NTB lies in its ability to accept a memory read/write operation initiated from one source PCIe domain, translate its target address, and then deliver and execute it in another PCIe domain. 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 that specify the portions of the physical memory address space of the end-point's 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 *memory 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 *main* side whereas the other side is the *support* side. For example in Figure 1(b), two servers each with its own PCIe domain are connected together using a two-port NTB. Assuming the NTB's left-hand port is configured to translate the physical address `0x90000` on its link side to the physical address `0x0` on its virtual side by writing `0x90000` to a link-side BAR and `0x0` to this BAR's memory translation register. In this case, the link side is the main side and the virtual side is the support side. A memory read/write initiated from the left-hand-side server targeting address `0x90000` will arrive at the BAR of this NTB's left-hand port, get translated to physical address `0x0` in the right-hand size PCIe domain, and eventually arrive at the main memory of the right-hand side server.

Note that address translation in NTB is *uni-directional*: only a memory read/write operation that hits a BAR at the main side is translated and relayed to the support side, but *not vice versa*. Using the same example, a write operation against the address `0x90000` at the link side is translated and relayed as a write operation against the address `0x0` at the virtual side, but a write operation against the address `0x0` at the virtual side is serviced locally and does not automatically get translated and relayed to the link side.

3.4 I/O MMU

IOMMU protects the physical memory space of a physical machine or host from unauthorized DMA transactions initiated by I/O devices by creating another level of address translation: from *device virtual address space* to *physical memory address space*. An IOMMU maintains a *Translation Control Entry* (TCE) table for device virtual addresses that are used as operands of DMA operations. Each TCE entry corresponds to a 4KB physical memory page in the host and contains associated access control bits. When a DMA operation from an I/O device hits the IOMMU, IOMMU searches the TCE using the source device ID and the operation's target device virtual address, converts the target device virtual address into the corresponding physical memory address if there is a matched TCE entry and the DMA operation is compatible with the matched TCE entry's access control bits, and denies the DMA operation otherwise.

4 LADON ARCHITECTURE

4.1 Architectural Overview

As shown in Figure 2, the key building block of the proposed PCIe-based multi-host rack-area network architecture is a hybrid top-of-rack (TOR) switch that consists of PCIe ports and Ethernet ports. Every machine or compute host (CH) in a rack is connected to two ports of this *Ladon* switch through two independent PCIe extender cards and two PCIe cables, and communicates with other CHs in the rack directly over PCIe and with machines outside the rack through the Ethernet ports of the TOR switch. Physically, a *Ladon* switch consists of a management host (MH), a standard PCIe switch with TB and NTB ports, and multiple Ethernet NICs each of which is connected to a TB port of the PCIe switch. The MH is connected to the *Ladon* switch through an upstream port, and serves as the root complex of the PCIe domain to which the PCIe switch and Ethernet NICs belongs. To address the concern that MH is a single point of failure, *Ladon* sets up two management hosts, one serving as the master management

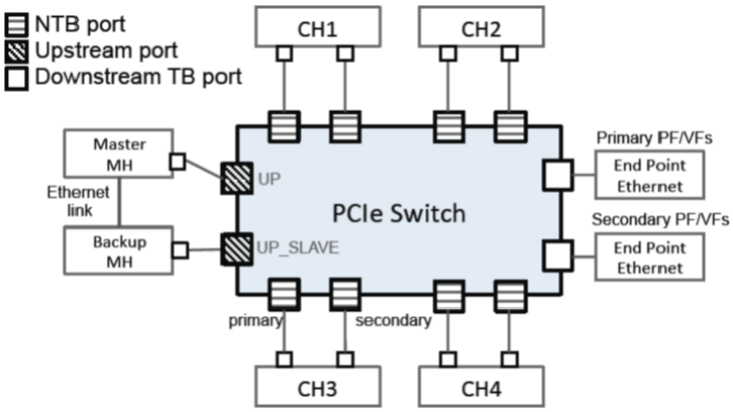

Figure 2: *The key building block of the proposed PCIe-based intra-rack network architecture is a hybrid top-of-rack switch that consists of PCIe ports and Ethernet ports. Each compute host (CH) is connected via two distinct NTB ports to the central PCIe switch, which could be built from multiple PCIe switch chips. A redundant backup management host (BMH) is passively monitoring the liveliness of the master management host (MMH). Two Ethernet NICs are attached to the PCIe switch to support inter-rack communication.*

host (MMH) and the other as the backup management host (BMH). The MMH and BMH are connected with a point-to-point Ethernet link, which carries PCIe state synchronization and heartbeats.

Every host connected to a *Ladon* rack, including the MH and CHs, has a *physical memory space*, which in turn comprises a *local memory subspace* and *non-local memory subspace*. A host's main memory and MMIO devices are mapped to its *local memory subspace*. The MH of a *Ladon* switch maps the local memory subspace of every attached machine to its non-local memory subspace, and in turn every attached machine maps the mapped portion of the MH's physical memory space to its non-local memory subspace. The MH's physical memory space thus serves as the system's global memory space. For example, assume every machine in the rack including the MH has a 32GB local memory subspace. The MH first maps the i-th attached machine's local memory subspace to the range $[32GB+(i-1)*32GB, 32GB+i*32GB)$ of its non-local memory space, as shown in Figure 3(a). Then each attached machine maps the active portion of the MH's physical memory space to its non-local memory subspace, which is above 32GB, as shown in Figure 3(b). With this set-up, an attached machine could access the i-th attached machine's local memory subspace by reading or writing the $[64GB+(i-1)*32GB, 64GB+i*32GB)$ range of its physical memory space. In other words, an attached machine could access its local memory resources either directly through a range in its local memory subspace (below 32GB), or indirectly through a range in its non-local memory subspace (above 64GB). Suppose there are 50 machines attached to a *Ladon* switch, including the MH. Then every attached machine could see a 1600GB worth of physical memory, with 32GB local in its own machine (zero hop), 32GB in the MH (one hop) and 1536GB in other attached machines (two hops). Consequently, a *Ladon* switch ties together the local memory spaces of all the machines attached to it into a global memory pool. The physical memory addresses in modern 64-bit X86 servers are at least 48 bits long, which is sufficient to support a global memory pool of up to 256TB.

Because every machine connected to a *Ladon* switch could potentially access each physical memory page of every other machine attached to that switch, data security and safety becomes a critical issue. More specifically, *Ladon* must guarantee that a machine be able to access a remote physical memory page in the global memory pool only when it is explicitly allowed to. *Ladon* leverages IOMMU [8, 14, 29] to provide this security guarantee. Every machine attached to a *Ladon* switch, including the MH, is assigned a unique PCIe device ID in the MH's PCIe domain, which remains unique after device ID translation across an NTB port. Therefore, the target address of a PCIe operation from Machine A to Machine C is matched against a different IOMMU mapping table in Machine C than that used for a PCIe operation from Machine B to Machine C. To open up a physical memory page P in Machine C only to Machine B, *Ladon* places an entry for P in Machine C's IOMMU mapping table for Machine B, so that memory accesses to P from Machine B could match an entry in this IOMMU mapping table. The above design not only opens up the page P to Machine B, but also restricts P's accessibility to Machine B only, because no other IOMMU mapping tables in Machine C contain an entry for Page P. To prevent Machine B from accessing the page P, *Ladon* simply removes the entry corresponding to P from Machine C's IOMMU mapping table for Machine B.

4.2 Data Plane Fault Tolerance

The MH's physical memory space is split into two halves called PMS1 and PMS2, each of which contains a local memory subspace and a non-local memory subspace. To survive a PCIe link/port failure, *Ladon* connects each CH to the TOR switch through a primary NTB and a secondary NTB, and maps each CH's local memory subspace to two distinct memory address ranges in MH's physical memory space, one in PMS1 and the other in PMS2. The example in Figure 3 assumes that each CH needs a 32GB local memory subspace for its main memory and memory mapped I/O (MMIO) devices. *Ladon* maps the i-th CH's local memory subspace to the MH's PMS1 ($[0, 1TB)$) via the CH's primary NTB port

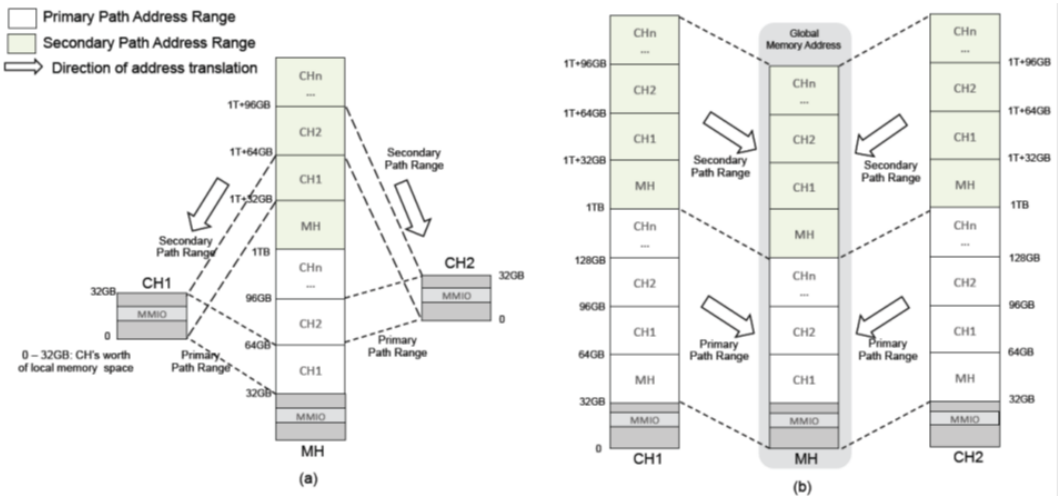

Figure 3: *Construction of* Ladon*'s global memory address space requires two address translations; one set up by the MH to map the local memory subspace of every attached machine to the MH's physical memory space, as shown in (a), and the other set up by each attached machine to map the MH's physical memory space into its own physical memory space, as shown in (b). To support fault tolerance, the MH maps each attached machine's local memory subspace to two independent regions in MH's physical memory space, thus offering two independent paths to reach each attached machine's local memory subspace.*

and to the MH's PMS2 ($[1TB, 2TB)$) via the CH's secondary NTB port, as shown in Figure 3 (a). Then the i-th CH maps the MH's PMS1 to the lower half of its non-local memory subspace via its primary NTB port, and the MH's PMS2 to the upper half of its non-local memory subspace via its secondary NTB port, as shown in Figure 3 (b). Consequently, the i-th CH's local memory subspace is mapped to and thus accessible via *two* distinct physical address ranges in the global memory pool, $[32GB + (i-1) * 32GB, 32GB + i * 32GB)$ via the CH's primary NTB port, and $[1TB + 32GB + (i-1) * 32GB, 1T + 32GB + i * 32GB)$ via the CH's secondary NTB port.

With this set-up, the MH or a CH could access the i-th CH's local memory subspace using *two* independent paths, one through the i-th CH's primary NTB port, and the other through the i-th CH's secondary NTB port. For example, a memory request from CH1 reading or writing the $[96GB, 128GB)$ range of its physical memory space goes through CH2's primary NTB port to reach CH2's local memory subspace, whereas a memory request from CH1 reading or writing the $[1T + 96GB, 1T + 128GB)$ range of its physical memory space reaches the same local memory subspace of CH2 through CH2's secondary NTB port.

Even though every CH occupies two physical address ranges in the global memory pool managed by the MH, at any point in time, only one of the two physical address ranges is active and therefore one of the two NTBs is used. With this set-up, whenever a link/port failure occurs that affects an CH, say X, the PCIe Advanced Error Reporting (AER) driver [30] at X is invoked, and it triggers the following fail-over procedure:

(1) Reports the detected error to the MH,
(2) Asks the MH to tell all other CHs to use X's secondary address range to reach X from now on, and
(3) Asks the MH to modify its IOMMU so as to ensure that the MH's local DMA operations destined to X also switch to X's secondary address range.

Each CH maintains a list of physical memory address ranges it uses to access a remote CH's resources, including main memory and MMIO devices. When a CH receives a PCIe fail-over notification about an affected CH, the CH modifies its list to indicate that it should reach the affected CH through the affected CH's secondary address range. For example, upon receiving a PCIe fail-over notification about CH1, CH2 changes its view of CH1 from $[64G, 96G)$ to $[1T + 64G, 1T + 96GB)$. A memory access to $1T + 64G$ in CH2 is translated to a memory access to $1T + 32G$ in MH and eventually hits the secondary NTB port of CH1 in the MH's domain.

Similarly, the MH maintains a similar list of physical memory addresses for target CHs of DMA operations initiated by I/O devices residing in the MH's PCIe domain. When a CH is in error, the MH modifies its IOMMU entries to redirect its DMA operations previously destined to the primary address range of the in-error CH to its secondary address range.

The above data plane fault tolerance design only handles link/port failures that affect individual attached CHs, but

cannot deal with entire PCIe switch failures or inter-switch link failures, which are left for future work.

4.3 Control Plane Fault Tolerance

4.3.1 Control Migration between Management Hosts

The management host (MH) of a *Ladon* switch is responsible for the mapping of the attached CHs' physical memory spaces into its physical memory space, and for exposing its physical memory space, including the address range associated with its PCIe devices, e.g. NICs, to the attached CHs in a secure way. After address allocation of each PCIe device, the MH configures the routing table of each PCIe bridge in the hierarchical routing tree, so that PCIe packets could be forwarded accordingly. Once the MH completes this set-up, its role becomes visible only when there is a change in its PCIe domain, i.e., addition or deletion of PCIe end-points. Most notably, the MH is not involved at all in the peer-to-peer data transfers among the PCIe end-points. In fact, when the MH dies, as long as the routing states in P2P bridges remain, the PCIe end-points could continue to exchange data with one another without any pauses. So when the MH fails, it is neither necessary nor desirable to recover from such a failure immediately, especially because the current recovery procedure for a MH failure requires a system-wide restart of all PCIe end points, which in turn may trigger at least a device driver reset and possibly a system reboot on the attached CHs. This subsection describes a seamless control plane fail-over mechanism that enables the data plane to continue operating without any disruption when the MH fails and recovers.

To achieve seamless fail-over across a MH failure, *Ladon* sets up two management hosts, one serving as the master management host (MMH) and the other as the backup management host (BMH). The MMH and BMH are connected with a point-to-point Ethernet link, which carries PCIe state synchronization and heartbeats. In addition, the MMH synchronously replicates modifications to the following states to the BMH over the dedicated Ethernet link:

- Results of initial PCIe device scanning and enumeration, including assignments of physical address ranges, interrupt numbers, etc.,
- Contents of the BARs, translation registers, device ID translation tables, IOMMUs of the NTB ports,
- Allocation of virtual functions (VFs) of PCIe devices supporting single-root IO Virtualization (SRIOV) to attached machines, and
- Internal states of the PCIe device drivers in its domain.

Modern PCIe switch chips, including the PCIe switch used in this project, support a capability to partition a physical switch into multiple *virtual switches* [21] that work independently. *Ladon* takes advantage of this flexibility by partitioning the root PCIe switch to which MMH and BMH are connected into two virtual switches, VS1 and VS2, with the MMH connected to the upstream port of VS1 and the BMH connected to the upstream port of VS2. VS1 connects MMH with all other PCIe devices, whereas VS2 has only BMH connected to it. When the MMH dies, the BMH detects it via the heartbeat mechanism and takes over by re-assigning all PCIe devices except the MMH from VS1 to VS2. The re-assignment allows VS2 to form a new PCIe hierarchy, with the BMH as the root and all other PCIe devices in the downstream. The key property of this port re-assignment operation is that *the routing states of the PCIe bridges do not change when the PCIe devices are switched from VS1 to VS2.* In fact, only the highest level bridge needs to change its upstream port to connect to the BMH. Because PCIe bridge's routing state only contains a contiguous address range that covers its downstream ports, and any packet with a target address falling within a PCIe port's address range will be forwarded to that port, changing a PCIe switch's upstream port has no impact on the peer-to-peer communication between PCIe end points attached to the switch. Moreover, because the BMH has exactly the same PCIe-related states as the MMH, it could immediately resume the control plane service, which processes events such as when PCIe devices are added or deleted or when PCIe resources are allocated or freed.

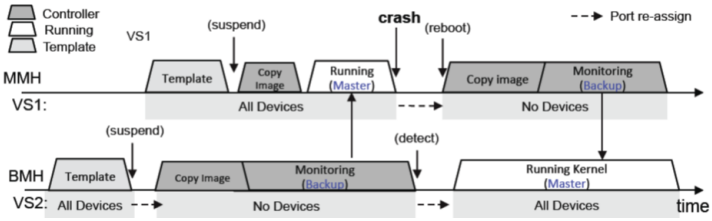

Figure 4: *Timeline of* Ladon*'s control plane fail-over mechanism, including the kernel images created and used by the MMH and BMH, and association of PCIe devices with VS1 and VS2 (shaded parts). The MMH is originally the master, then, upon detecting the MMH crashes, the BBH becomes the new master by rebooting itself with a kernel image that captures the initial PCIe device driver states and then augmenting it with subsequent PCIe device changes transferred from the MMH.*

4.3.2 Capturing of PCIe Device Driver States

Among the states maintained by a management host, the internal states of PCIe device drivers, for example, NIC or RAID drivers, are the most difficult if not impossible to capture, especially for closed-source drivers [25]. To overcome

this implementation challenge, *Ladon* leverages Linux's suspend/resume [3, 15] facility, in which when the suspend operation is invoked, the entire system state including the drivers' states, are snapshotted and saved to the disk.

Specifically, both the MMH and BMH have three disk partitions that hold three different kernel images: the *controller kernel image*, the *template kernel image*, and the *running kernel image*. The controller kernel is used to boot up a management host so as to manage the other two kernel images. The template kernel is used to hold a golden kernel image to be reused after a failure, and is created from a suspend operation of the system after all PCIe device drivers are initialized. The running kernel is the kernel MMH and BMH runs to perform management host functions.

As shown in Figure 4, initially all PCIe devices except the MMH are connected to VS2, and the BMH boots up from the controller kernel image until all of its PCIe device drivers are initialized. Then the BMH suspends itself to form the template kernel image, reboots itself from the controller image, and then copies the template image to form the running image. Finally the BMH is again rebooted from the controller image, and, from this point on, monitors the health of MMH using heartbeat messages.

Next, all PCIe devices except the BMH are re-assigned to VS1, and the MMH boots up from the controller image and continues to run until all of its PCIe device drivers are initialized. Then the MMH suspends itself to form the template image, reboots itself again from the controller image, and then copies the template image to form the running image. Finally the MMH reboots itself once more, this time from the running image, and gets the whole system going.

When the BMH detects that the MMH encounters an irrecoverable error, the BMH instructs all other members of VS1 except the MMH to join VS2, and reboots itself using the running image. Because the BMH's running image contains all the necessary PCIe device driver states, the BMH is able to manage the PCIe devices and NTBs in VS2's PCIe domain *without resetting them*. Note that the BMH must reassign members of VS1 to VS2 *before* rebooting itself, because otherwise the reboot may fail as the device drivers in the running kernel image could not find their corresponding devices. The control plane's service is disrupted during the time only when the BMH is rebooted.

The suspend/resume-based approach is simple to implement because it does not require understanding of or modifications to PCIe device drivers. However, it is limited in that it cannot accommodate any run-time changes to PCIe devices or PCIe resource allocation after the initial snapshots were taken. To account for these changes, the MMH must log and transfer them via the dedicated Ethernet link to BMH at run time, and the BMH replays them after rebooting itself using the running kernel image during the take-over process.

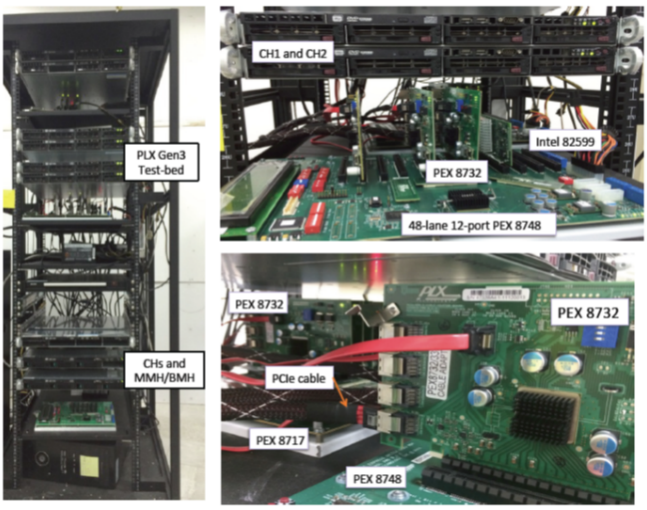

Figure 5: *The* Ladon *testbed is a rack (left) consisting of five 1U X86 servers, one dual-port SRIOV 10GE NIC, and multiple PCIe switches. CHs are connected through external PCIe cables to the NTB ports of the TOR PCIe switch (left top and bottom).*

5 EVALUATION

5.1 Prototype Implementation

The hardware test-bed in the *Ladon* prototype consists of five Intel X86 servers, one 10GE NIC, eight PLX PEX8732 switches, two PLX PEX8717 switches supporting non-transparent ports, and one PLX PEX8748 PCIe switch. Two servers serve as the master and backup management host, the other two servers serve as compute hosts, and the fifth server is a remote server that is connected with the TOR switch through a 10GE link. Each of these servers is a Supermicro 1U server equipped with an 8-core Intel Xeon 3.4GHz CPU and 8GB 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 PEX8748 switch is partitioned into two virtual switches, VS1 and VS2. The master management host (MMH) is connected to the VS1's upstream port of the PEX8748 PCIe switch through a PCIe adaptor plugged into the host's x4 PCIe slot, while the backup management host (BMH) is connected to the VS2's upstream port using the same set-up.

The PEX8717 device is a 16-lane 10-port switch that supports up to 2 NTB ports and four DMA channels for peer-to-peer data transfers. In addition, an Intel 82599 SRIOV 10GE NIC serves as the out-of-rack Ethernet adapter, is plugged into a downstream TB port of the PCIe switch, and is connected to another Intel 82599 SRIOV 10GE NIC on the fifth server using a duplex fiber optic cable.

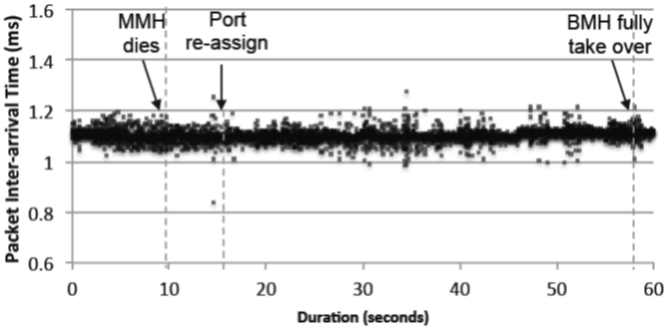

Figure 6: *The observed inter-packet arrival time of a UDP connection between a test sender and a test receiver, which are connected through a PCIe switch and a 10GE link. Between the time when the MMH fails and the time when the BMH takes over, there is no packet loss and the variations in the inter-packet arrival times observed at the receiver are negligible.*

The PEX8748 PCIe switch is a 48-lane, 12-port PCIe Gen3 switch, where each lane provides 8Gbits/s raw transfer bandwidth. We assigned 4 lanes of this PCIe switch to the management host, 4 lanes to each of the two compute hosts, and another 4 lanes to the Intel 82599 NIC. The NIC and the two compute hosts are connected to ports that are assigned to VS1. Because the master management host is connected to VS1's upstream port, the VFs and PF on the Intel NIC are under the control of the master management host

The *Ladon* prototype implementation is based on Fedora 15 with the Linux kernel version 2.6.38, and consists of a management host component and a compute host component. The management host enumerates all the PCIe devices in its domain, including non-transparent PCIe switches and Ethernet NICs, sets up the BARs and translation registers on NTBs, maps compute hosts' memory and I/O devices to the management host's physical memory space, and allocates and de-allocates virtual PCIe devices to compute hosts for I/O device sharing. Each CH runs the Intel's VF Ethernet driver and an NTB driver, while the MH runs the Intel PF Ethernet driver and also the NTB driver, controlling all the NTB device at side of the PCIe switch. *Ladon* implements its failover mechanism as a registered handler function of both the NTB driver and Ethernet driver at each CH and the MH.

5.2 Fail-over Latency for a Management Host Failure

To assess the performance impact of a management host failure on network connections that traverse the *Ladon* hybrid switch, we used one of the compute hosts in the test-bed as the test sender and a server outside the rack as the test receiver, and programmed the test sender to send a sequence of 64-byte UDP packets continuously once every msec for a period of time. During this period, we manually crashed the master MH and triggered the control plane fail-over mechanism in the backup MH, which re-assigned attachment compute hosts and the 10GE NIC from VS1 to VS2 and initiated a reboot of itself before eventually taking over the control plane, and measured the packet loss rate at the test receiver. We ran the above experiment five times, and found no packet loss during the entire experiment period.

Step	Time (sec)
MMH Failure Detection	0.25
Switch Port Reassignment	0.001
BMH Reboot	38.1
Total	38.35

Table 1: *Detailed delay breakdown of the fail-over process for a management host failure, in which the BMH takes over the control from the MMH*

Suspecting that a PCIe switch may buffer PCIe packets to survive the management host failure, we also measured the inter-packet arrival time at the receiver. Figure 6 shows that the observed inter-packet arrival time remains flat throughout the entire experiment period for each of the five runs. That is, not only the transition from the master MH to the backup MH loses *zero* packet, it does not even have any impact on the test network connection's performance. We have repeated the same tests for UDP connections between servers that are both attached to the PCIe switch, and obtained identical results. These results convincingly demonstrate the clean isolation between *Ladon*'s control plane and data plane, to the extent that the data plane's functionality and performance are completely insulated from such major control plane breakdowns as a management host failure.

Even though the above result demonstrates that a management host failure and fail-over has zero impact on the *data plane*, a management host failure does disrupt the services provided by *Ladon*'s *control plane*, such as addition or deletion of PCIe devices, or allocation of PCIe device resources to compute hosts. As shown in Table 1, after detecting the failure of the master management host using the heartbeat mechanism, which takes on average 0.25 second when the heartbeat timer is set to 0.1 second, the BMH takes over the PCIe domain by first reassigning all ports except the port assigned to the MMH from VS1 to VS2, and then rebooting itself from the running kernel image. The BMH re-assigns ports by programming a set of special registers on the root PCIe switch chip, which takes less than 1 millisecond. Rebooting the BMH to the point of fully resuming control plane service takes on average 38.1 seconds. Therefore, the total service disruption time to *Ladon*'s control plane due to a management host failure is around 38.35 seconds.

Step	Time (μsec)
Link Failure Detection	2.4
Failure Determination	8.5
Route Change Notification and Processing	76
Total	86.9

Table 2: *Detailed delay breakdown of the fail-over process for a link/port failure associated with a compute host*

In contrast, without *Ladon*'s control plane fail-over scheme, after detecting the MMH fails, the BMH must first initiate a system-wide shut-down and restart to all CHs, and then reboots itself to resume the services on the data and control plane. The entire process takes about 85 seconds. During this period, the data plane service is unavailable and no packets can travel on the PCIe network.

5.3 Fail-over Latency for a Link Failure

The current *Ladon* prototype only handles failures of links/ports associated with an CH. To efficiently handle PCIe link/switch failures, *Ladon* detects occurrences of failures by leveraging PCIe's advanced error reporting (AER) [30] capability, which allows a PCIe component to send an error reporting message to the root complex. Such error signaling can occur because of a failure of a PCIe link or of a transaction initiated on a PCIe link. Table 2 shows the delay breakdown of the fail-over process of such a failure, which is broken down into the following three components.

Failure Detection: This is the time between when a PCIe device detects an uncorrectable fatal error and sends an error message to the root complex and triggers an interrupt, and when the PCIe AER driver responds to the interrupt. In our prototype, the link failure detection time is around 2.4 μsec.

Failure Determination: When a CH's AER driver receives an AER interrupt, it notifies the MH, which makes a final verdict on whether it is indeed a data plane failure that deserves a fail-over action. In the current *Ladon* prototype, a CH's AER driver notifies the MH by writing to a pre-defined 4-byte memory location, which a thread on the MH polls constantly and, upon detecting a change, reads it and immediately writes a response into a pre-defined 4-byte location in the CH. The round-trip delay as measured by a CH's AER driver is 17 μsec, which suggests that the one-way notification delay is 8.5 μsec.

Route Change Notification and Processing: After determining a fail-over action is justified, the MH notifies all the CHs about the route change one by one. As a result, the total notification latency from the MH to all CHs becomes the number of CHs times 8.5 μsec. As mentioned in Section 4.2, each CH maintains a list of physical memory addresses it uses to access a remote CH's resources, and, upon receiving a fail-over notification about an in-error CH, modifies this list to ensure that all subsequent accesses to the in-error CH use its secondary address range. Modification to this list takes less than 1 μsec. Because the current prototype could hold up to 8 CHs, the total notification delay is around 8*(1+8.5) = 76 μsec.

In summary, the total fail-over latency for a link failure in the current prototype (up to 8 CHs) is about 86.9 μsec, the bulk of which is attributed to route change notification. This route change notification delay could be significantly reduced if *Ladon* exploits the multicast capability of modern PCIe switches, where a memory write operation with a multicast target address is sent to all PCIe end points that belong to the corresponding multicast group.

6 CONCLUSION

Because PCI Express's transceiver is designed for short distance (less than 10 meters), it is simpler, cheaper, and more power-efficient, and makes a more cost- and power-efficient system area network for intra-rack inter-machine communications than comparable technologies such as 10Gbps Ethernet or Infiniband. Its memory-based addressing model is especially compelling because it opens up the possibility for one machine to directly read or write any memory location of another machine without involving the latter's CPU. *Ladon* exploits this direct remote memory access capability to set up its fail-over mechanisms that ensure continued network operation in the presence of a control/data plane failure. This paper presents the design, implementation and evaluation of a fault-tolerant PCIe-based rack area network architecture that takes effective advantage of PCIe architectural features to significantly reduce the service disruption time due to a failure of a PCIe root complex and a PCIe link/port. The specific research contributions of this work thus include:

- A fault-tolerant PCIe-based rack-area interconnect architecture that connects together a set of compute hosts and PCIe devices, and allows each compute host's resources to be mapped into two distinct memory address ranges in a rack-wide global memory space and thus be accessible via two independent paths, and
- Design of a seamless data plane fail-over mechanism that detects a PCIe link/port failure and the affected compute host, notifies the management host, and gets all other compute hosts to communicate with the affected compute host using its secondary address range from this point on, and
- Design, implementation and evaluation of a seamless control plane fail-over mechanism that successfully prevents a control plane failure from disrupting the data plane and incurs only a modest service disruption time on the control plane service.

REFERENCES

[1] [n. d.]. I/O Consolidation White Paper, NextIO, Inc. http://www.nextio.com/resources/files/wp-nextio-consolidation.pdf. ([n. d.]). accessed June 4, 2012.

[2] [n. d.]. PCI Express System Interconnect Software Architecture for x86-based Systems. http://www.idt.com. ([n. d.]). K. Kong, September 2008.

[3] [n. d.]. TuxOnIce. Revolutionize the way you start your computer. http://tuxonice.net/. ([n. d.]).

[4] [n. d.]. Xsigo Systems, Inc. http://en.wikipedia.org/wiki/XsigoSystems. ([n. d.]). accessed in July 2017.

[5] 2008. Multi-Root I/O Virtualization and Sharing 1.0 Specification, PCI-SIG Standard. (2008).

[6] 2008. Single-Root I/O Virtualization and Sharing Specification, Revision 1.0, PCI-SIG Standard. (2008).

[7] A3Cube Inc. [n. d.]. RONNIEE Express Fabric - The In Memory Network . http://www.a3cube-inc.com/ronniee-express.html. ([n. d.]).

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

[9] R. Budruk, D. Anderson, and T. Shanley. 2004. *PCI Express System Architecture*. Addison-Wesley Professional.

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

[11] G. K. Bayer and D. F. Craddock and T. A. Gregg and M. Jung and A. Kohler and E. G. Nass and O. G. Schlag and P. K. Szwed . [n. d.]. Switch failover control in a multiprocessor computer system. ([n. d.]). Mar. 18 2014, US Patent 8,677,180.

[12] R. Gillett. 1996. Memory channel network for pci. *IEEE Micro, vol.16, no.1* (1996), 12–18.

[13] R. Gillett and R. Kaufmann. 1997. Using the memory channel channel network. *IEEE Micro, vol.17, no.1* (1997), 19–25.

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

[15] Hui-Hao Chiang and Maohua Lu and Tzi-cker Chiueh. 2011. Physical Machine State Migration. In *Proceedings of IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*. 25–32.

[16] V. Krishnan. 2007. Towards an Integrated IO and Clustering Solution Using PCI Express. In *Proceedings of 2007 IEEE International Conference onCluster Computing*. IEEE, 259–266.

[17] V. Krishnan. 2008. Evaluation of an integrated pci express io expansion and clustering fabric. In *Proceedings of 16th IEEE Symposium on High Performance Interconnects (HOTI âĂŹ08)*. 93–100.

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

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

[20] R. L. Billau and J. D. Folkerts and R. L. Franke and J. S. Harveland and B. G. Holthaus. [n. d.]. Autonomic pci express hardware detection and failover mechanism. ([n. d.]). Aug. 29 2007, US Patent App. 11/846,783.

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

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

[23] STMicroelectronics. [n. d.]. iATU Operations, SPEAr1340 architecture and functionality reference manual. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/referencemanual/DM00024168.pdf. ([n. d.]).

[24] Mark J. Sullivan. 2010. Intel Xeon Processor C5500/C3500 Series Non-Transparent Bridge. *Technology@ Intel Magazine* (2010).

[25] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. 2006. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS), vol. 24, no.4* (2006), 333âĂŞ–360.

[26] T. Hanawa and T. Boku and S. Miura and M. Sato and K. Arimoto. 2010. Pearl: Poweraware, dependable, and high-performance communication link using pci express. In *Proceedings of IEEE/ACM International Conference on Green Computing and Communications (GreenCom) and on Cyber, Physical and Social Computing (CPSCom)*. IEEE, 284–291.

[27] William Tu and Mark Lee and Tzi-cker Chiueh. 2013. Secure i/o device sharing among virtual machines on multiple hosts. In *Proceedings of ACM International Symposium on Computer Architecture*. ACM, 108–119.

[28] William Tu and Tzi-cker Chiueh and Mark Lee. 2014. Marlin: a memory-based rack area network. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for Networking and Communications Systems*. ACM, 125–136.

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

[30] Y. Zhang and T. L. Nguyen. 2007. Enable pci express advanced error reporting in the kernel. In *2007 Ottawa Linux Symposium*. 297–304.