Marlin: A Memory-Based Rack Area Network

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

Disaggregation of hardware resources that are traditionally embedded within individual servers into separate resource pools is an emerging architectural trend in hyperscale data center design, as exemplified by Facebook's *disaggregated rack* architecture. This paper presents the design, implementation and evaluation of a PCIe-based rack area network system called *Marlin*, which is designed to support the communications and resource sharing needs of disaggregated racks. By virtue of being based on PCIe, *Marlin* presents a memory-based addressing model for both I/O device sharing among multiple hosts and inter-host communications. That is, when a node communicates with other nodes or accesses resources in the same rack, it uses memory read and write operations. In the area of inter-node communications, *Marlin* offers *hardware-based remote direct memory access* (HRDMA) as a first-class communications primitive between servers within a rack. In addition, *Marlin* supports socket-based communications for legacy network applications and cross-machine zero memory copying for applications designed specifically to take full advantage of memory-based communications. Empirical measurements on a fully operational *Marlin* prototype based on 4-lane Gen3 PCIe technology show that the one-way kernel-to-kernel latency is 8.5μsec and the end-to-end sustainable TCP throughput is 19.6 Gbps.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; B.3.2 [**Memory Structures**]: Shared Memory

Keywords

PCIe Fabric; Rack Disaggregation; Rack-Area Network; RDMA; MR-IOV; SR-IOV; Non Transparent Bridge

1. INTRODUCTION

Hyperscale data centers increasingly use a rack rather than a machine as the basic building block. Recently, Facebook advocated a *disaggregated rack* architecture [8, 9], in which a rack consists not of a set of self-contained hosts, but of a CPU/memory pool, a disk pool, and a network interface (NIC) pool, which are connected through a high-bandwidth and low-latency rack-area network. A major deployment advantage of the disaggregated rack architecture is that it allows different system components, i.e. CPU, memory, disk and NIC, to be upgraded according to their own technology cycle. From the architectural standpoint, rack disaggregation reduces each "host" to a CPU/memory module, decouples CPU/memory from I/O devices such as disk controllers and network interfaces, and enables more efficient and flexible I/O resource allocation and utilization.

The key enabling technology for the disaggregated rack architecture is the ability for multiple CPU/memory modules to share I/O devices. *Ladon* [46] is a software-only solution that allows multiple servers or multiple virtual machines (VM) running on distinct servers to share PCIe-based I/O devices securely, efficiently and transparently. Specifically, VMs running on distinct servers of a rack are able to use the same PCIe-based NIC to communicate with VMs running on a different rack, without interfering with one another and at native speed. In *Ladon*, servers are connected with the I/O devices they share via a PCIe network.

This paper proposes to take a PCIe-based network that is originally designed for I/O device sharing and extend it into an intra-rack inter-server network system called *Marlin* for the disaggregated rack architecture, and describes two communication mechanisms that are built on top of this architecture: a socket communication mechanism that is meant to support legacy network applications, and a cross-machine memory copying mechanism that is designed to expose *Marlin*'s raw transfer capability to applications that are developed specifically to take advantage of *Marlin*. Logically, *Marlin* offers a *hybrid Ethernet/PCIe switch* to servers in a rack so that they could communicate with one another directly via the PCIe links and with machines outside the rack through Ethernet links. Physically, *Marlin* consists of a management host (MH), a PCIe switch, and a set of Ethernet NICs, and each machine in the rack is connected to a port in the PCIe switch through a PCIe expansion card and a PCIe cable. *Marlin* leverages *Ladon*'s novel multi-root I/O device sharing mechanism [46] to enable servers in a rack to share the Ethernet ports in the hybrid switch.

A distinct feature of the PCIe architecture is its memory-based addressing model. Every interaction with a PCIe device is through a memory read/write operation, including configuring a PCIe device, transferring data to and from a PCIe device via an I/O or memory access instruction, and interrupting a PCIe device. Accordingly, the fundamental communication primitive in *Marlin* is a hardware-based remote direct memory access (HRDMA) mechanism, which allows one machine to initiate a DMA transaction against another machine's memory in the same rack area network *without any software intervention*. To support existing socket-based

ANCS'14, October 20–21, 2014, Los Angeles, CA, USA.

http://dx.doi.org/10.1145/2658260.2658262.

applications, *Marlin* supports an Ethernet-over-PCIe layer that transfers Ethernet frames using HRDMA. Because of *Marlin*'s memory-based addressing model, this layer features a *sender-based buffer management* scheme [19] that allows a sender machine to manage the receive buffer space that a receiver machine reserves for the sender. To take full advantage of *Marlin*'s underlying HRDMA mechanism, *Marlin* provides a cross-machine memory copying operation (CMMC) that allows an application process running on one machine to copy data from its address space to the address space of another application process running on another machine in the same rack, *without any software in between*. In other words, CMMC is a zero-copy application-level memory copying mechanism, from the sender application's address space to the receiver application's address space.

Marlin is based on the emerging software-defined network (SDN) architecture as exemplified by OpenFlow [35], in that it decouples the network's control plane from data plane, and it centralizes the control plane functionalities into a dedicated controller, specifically, the management host. *Marlin* takes advantage of this central control plane architecture to configure the transparent and non-transparent PCIe switches of a rack's backplane, establish a global address space that is visible to a rack's hosts, and control the accessibility of this address space to specific hosts according to user-specified policies.

2. PCI EXPRESS ARCHITECTURE

2.1 PCIe Architecture

PCIe is a layered protocol consisting of a physical layer, a data link layer and a transaction layer. The transaction layer is responsible for encapsulating high-level PCIe transactions issued by the OS or firmware, i.e., *memory, I/O, configuration* and *message*, into PCIe transaction layer packets (TLP). The data link layer is responsible for sequencing transaction layer packets (TLP) using a sequence numbering mechanism, and ensuring their reliable delivery using CRC and ACK/NACK. In a server, the *root complex* residing in the Northbridge part of the chipset connects CPU and memory with the PCIe network and implements the transaction layer. Each PCIe device in the server is uniquely identified by a bus/device/function ID, and is given a set of *configuration space registers* (CSR) and contains a standardized part (such as device vendor ID, command, *base address register* or BAR, etc.) and a device-specific part. A PCIe card occupying a PCIe slot could function as one or multiple physical functions (PF), each acting like a logical PCIe device with its own device ID and CSR. The PCI-SIG standard on single-root I/O virtualization (SRIOV) [15] virtualizes a PCIe device into multiple virtual PCIe devices, where 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 CSR, whereas a VF is a lighter-weight PCIe device with its IDs but without a complete CSR. Instead, each VF only has its own copy of the data plane-related portion of the CSR, but the control plane-related portion is borrowed from its underlying PF's CSR. 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.

Although PCIe was originally designed to be a system backplane, it is now capable of scaling to a large number of PCIe end points through one or multiple PCIe transparent bridges (TB) or switches. Topologically the end points of a PCIe domain form a tree, whose root is the root complex. To the OS, every operation in 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 by the transaction layer and delivered to the target PCIe end point. Routing is based on the address carried in the packet. Unlike Ethernet, PCIe network is lossless at the transport layer; its robust flow-control mechanism prevents packet from being dropped.

2.2 Non-Transparent PCIe Bridging

In the PCIe architecture, at any point in time every PCIe domain has exactly one active root complex. Therefore, in theory, two servers are not allowed to co-exist in the same PCIe domain. Non-transparent bridge (NTB) is a part of the PCI-SIG standard 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 *Marlin* system.

A two-port NTB represents two PCIe end points, each of which belongs to a separate PCIe domain. These two end points each expose a type 0 header type in 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 translation*, *device ID translation* and *messaging* facilities that allow the two PCIe domains to work together as a whole 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 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 in its ability to deliver a memory read/write operation initiated from one PCIe domain 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 that 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 *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 *primary* side whereas the other side is the *secondary* side. For example, an NTB could translate the physical address 0xF8800000 on its link side to the physical address 0xF8A00000 on its virtual side by writing 0xF8800000 to a link-side BAR and 0xF8A00000 to this BAR's memory translation register. In this case, the link side is the primary side and the virtual side is the secondary side.

NTB also contains an ID translation table, which converts the source ID of a PCIe operation as it travels from one domain to another. 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 provide an out-of-band control path for these two PCIe domains to better communicate with each other.

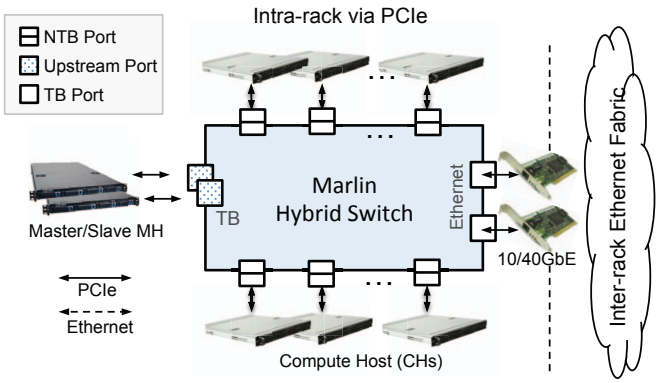

Figure 1: *The key building block of the proposed rack area networking architecture is a hybrid top-of-rack switch that consists of PCIe ports and Ethernet ports.*

3. RACK AREA NETWORKING

3.1 System Architecture

As shown in Figure 1, the key building block of the proposed rack area networking architecture is a hybrid top of rack (TOR) switch consisting of PCIe ports and Ethernet ports. Every machine or compute host in a rack is connected to a port of this *Marlin* switch through a PCIe extender card and a PCIe cable, and communicates with other machines in the rack directly over PCIe and with machines outside the rack through the Ethernet ports. As a rack's server density increases, multiple such TOR switches are needed to connect the machines in a rack. Physically, the *Marlin* switch consists of a management host (MH), a standard PCIe switch with TB and NTB ports, and multiple Ethernet NICs each connected to a TB port of the PCIe switch. The MH is connected to the *Marlin* switch through a TB port, and serves as the root complex of the PCIe domain to which the PCIe switch and Ethernet NICs belong. Other machines or compute hosts in the rack are connected to the *Marlin* switch through the NTB ports.

The MH of the *Marlin* switch maps the main memory of every attached machine to its physical memory address space, and in turn exposes its physical memory address space to every attached machine. For example, assume every machine in the rack including the MH has 32GB worth of local memory. The MH first maps the main memory of the i-th attached machine's local memory to the range from $[32GB + (i-1) * 32GB, 32GB + i * 32GB)$ of its physical memory address space, as shown in Figure 2(a). Then each attached machine maps the MH's entire physical address space to the range of its physical address space above 32GB. With this set-up, an attached machine could access the i-th attached machine's local memory by reading or writing the $[64GB + (i-1) * 32GB, 64GB + i * 32GB)$ range of its local physical memory address range, as shown in Figure 2(b). Therefore, an attached machine could access its local memory through either a range below 32GB (directly), or through a range above 64GB (indirectly through the MH's physical address space). Suppose there are 50 machines attached to the *Marlin* 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 *Marlin* switch turns the physical memories of all the machines attached to it into a global memory pool.

Modern BIOS performs physical memory address allocation for PCIe devices at the system boot-up time, but this allocation conflicts with the above design. *Marlin* overcomes this by overriding BIOS's allocation decisions after the kernel takes control. Today's 64-bit servers support at least 48 bits of physical addresses, which are enough to support a 256TB physical address space.

When the *Marlin* switch boots up, its MH enumerates all the devices connected to the PCIe switch, including the Ethernet NICs and NTB ports. Then it sets up the above memory address mappings by programming the BARs, the memory translation registers, and the device ID translation tables on the NTB ports of the PCIe switch. Finally, the MH also exposes the physical memory address range associated with the Ethernet NICs to all attached machines so that they could directly interact with these NICs. To enable attached machines to access these NICs in a way that does not interfere with one another, the MH allocates to each machine one or multiple VFs from the SRIOV-capable Ethernet NICs. This VF allocation mechanism requires a special PCIe driver to be installed in the attached machines [46].

Because every machine connected to a *Marlin* switch could address each physical memory page of every other machine attached to the same switch, data security and safety becomes a critical issue. More specifically, *Marlin* 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. *Marlin* leverages IOMMU [16, 27, 48] to provide this security guarantee. When a PCIe device on a machine accesses the machine's physical memory, IOMMU translates the addresses specified in the access operations into the machine's physical memory address space using an IOMMU mapping table. When the target address of a PCIe operation does not match any entry in the IOMMU mapping table, the operation is denied and aborted. IOMMU was originally proposed to prevent one VM in a machine from corrupting another VM in the same machine, but is re-purposed in *Marlin* to prevent one physical machine from accessing the main memory of another physical machine without the latter's permission.

Every machine attached to a *Marlin* 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, *Marlin* 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 close the page P to Machine B, *Marlin* simply removes the entry corresponding to P from Machine C's IOMMU mapping table for Machine B.

3.2 CPU-based Accesses to Remote Memory

Because *Marlin* enables every machine connected to a *Marlin* switch to directly address every other machine's memory, it offers two new ways for machines connected to the same switch to interact with one another. First, a CPU on one machine could use standard remote memory copying API functions such as memcpy to move data between machines. However, because the shared memory abstraction supported by *Marlin* is non-cache-coherent, memory regions that are meant to be accessed by multiple machines must be marked as non-cacheable. Such non-cacheable shared mem-

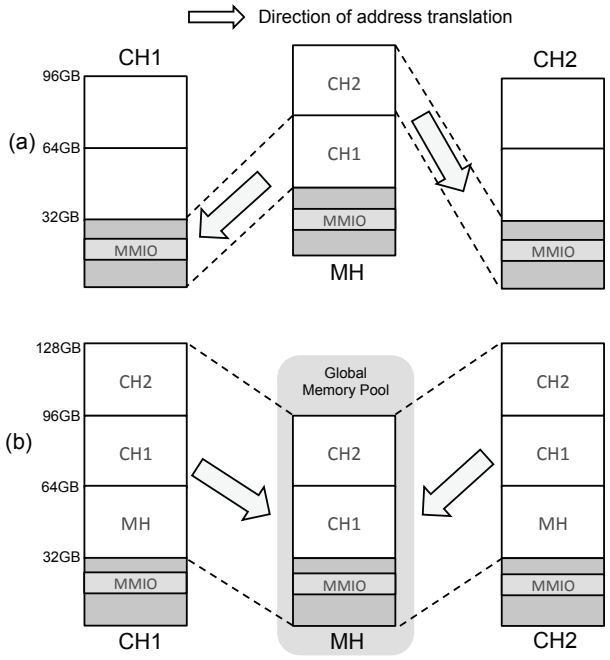

Figure 2: *Construction of* Marlin*'s global memory pool requires two address translations; one set up by the MH to map the physical memory of every attached machine, or Compute Host (CH), to the MH's physical memory address space, as shown in (a), and the other done by each CH to map the MH's physical memory address space into its own local physical address space, as shown in (b).*

ory in *Marlin* could be used to implement a memory-based locking mechanism across machines. Second, *Marlin* allows a CPU in one machine to directly send an MSI-X based interrupt to another machine using a single memory write instruction, without relying on any additional I/O device or doorbell mechanism. Once the memory write PCIe packet from the interrupting host gets translated by NTB to the other side, the memory write becomes a legitimate MSI-based write and eventually hits another host's local APIC memory area. For example in Figure 2, CH1 interrupts CH2 by writing to CH2's local APIC memory address space with destination address 96G + 0xFEE00000. The NTB translates ths address from 96G + 0xFEE00000 in CH1 to 0xFEE00000 in CH2, which becomes a legitimate MSI address.

3.3 Hardware-based Remote Direct Memory Access

On each NTB port of the *Marlin* switch is a DMA engine that could initiate a DMA transaction from one local physical address range to another local physical address range. Because the physical memories of the MH and all other attached machines are all mapped to each attached machine's local physical address range, this DMA engine could perform both *local memory copying*, where both the source and target reside in the local machine, and *inter-machine memory copying*, where either the source or target but not both resides in a remote machine. This inter-machine memory copying operation is a form of remote DMA (RDMA) that is completely hardware-based and does not involve any software. It differs significantly from InfiniBand (IB) RDMA, in which the payload at the source is DMAed into an IB interface, fragmented and encapsulated into IB packets, which are sent over IB links, get de-

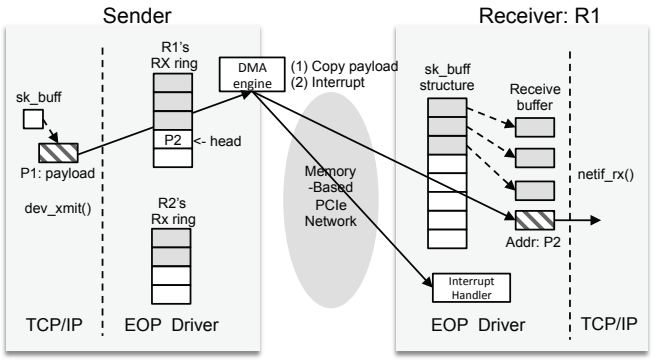

Figure 3: *Transmission and reception of an Ethernet over a PCIe network. The DMA engine copies the payload referenced in an sk_buff structure at the sender side directly to the receiver side, in this case, R1. The sender-side EOP driver on one node keeps track of the head and tail pointer of every receive ring buffer that other nodes allocate for it. Transmitting a packet requires at least two HRDMA operations, one to copy the actual packet and the other to interrupt the receiver to inform it of the transmission completion.*

capsulated and re-assembled, and DMAed to the receive buffers. In contrast, payloads are packaged into PCIe packets which flow from source to destination, and as a result, payloads do not need to experience additional format translation (e.g. between PCIe and Infiniband) and could be moved in a cut-through rather than store-and-forward fashion. This HRDMA mechanism is the fundamental building block of the *Marlin* architecture.

The DMA engine on the PCIe switch supports scatter-and-gather DMA. Therefore, the current HRDMA implementation exploits this capability to aggregate multiple high-level operations issued from the same source node, each destined to a different destination node, into a single DMA transaction. In addition, the DMA engine supports multiple channels, each of which could carry out an independent DMA transaction on its own. The current HRDMA implementation further exploits this fact to issue multiple concurrent DMA transactions at a time. These two aggregation mechanisms significantly improve *Marlin*'s HRDMA throughput.

3.4 Ethernet over PCIe

The Ethernet over PCIe (EOP) interface is positioned as a pseudo device interface similar to an Ethernet NIC. In the IP routing table of every attached machine, all the attached machines that are directly connected to the *Marlin* switch are assumed to be in the same IP subnet and are explicitly routed to the EOP interface, whereas all other machines are routed through the Ethernet interfaces. This way, packets destined to machines attached to the same *Marlin* switch as the home machine are sent out through the EOP interface. In addition, to improve the transport efficiency, the MTU of the EOP interface is set to 64KB, the same as the maximum segment size for IP datagram.

The EOP driver exposes the same interface to the high-level IP protocol stack and uses the same transmission and reception buffer ring as a standard Ethernet driver. A transmission or reception buffer ring is an array of pointers to buffers that is used in a cyclic fashion. In addition to an array of buffer pointers, a receive buffer ring holds two index pointers to the array, *head* and *tail*. Receiver buffer ring entries between *head* and *tail* point to packets that are received and not yet forwarded up to the IP protocol stack. However, there are two key differences between the EOP driver and the

standard Ethernet driver. First, each attached machine pre-allocates a receive buffer ring specifically for *every other* attached machine in the rack. Second, the free space in a receive buffer ring reserved on a machine M1 for a remote machine M2 is supplied by M1 but directly consumed by M2.

As shown in Figure 3, when receiving a socket buffer structure, e.g., sk_buff in Linux, that contains a pointer ($P1$) to an Ethernet packet to be transmitted, the sending-side logic of the EOP driver takes control and performs the following steps:

1. Derives from the packet's destination MAC address the corresponding receive buffer ring and then the ring's current *head* pointer ($P2$),
2. Issues an HRDMA transaction using $P1$ as the source address, $P2$ as the target address, and the Ethernet packet's length as the HRDMA transaction size, and
3. Handles the *completion* interrupt when the HRDMA transaction is done by issuing a *packet arrival* interrupt to the destination machine, incrementing (modulo) the *head* pointer of the associated receive buffer ring, and calling on the upper layer to send down the next packet to be transmitted.

In the design above, the sending-side logic of the EOP driver may be blocked at Step 1 because there is no free buffer at the remote receive buffer ring, or at Step 2 because all descriptors in the DMA engine are in use. When the EOP driver on the destination machine receives a packet arrival interrupt, its receiving-side logic comes into play and performs the following:

1. Locates the local receive buffer ring corresponding to the machine issuing the interrupt, then its *tail* pointer and finally the pointer to the buffer actually holding the received packet,
2. Constructs a sk_buff structure that contains the actual receive buffer's pointer among other things, and forwards the sk_buff up to the IP protocol stack, and
3. Allocates a new buffer, puts its pointer into the receive buffer ring entry pointed to by the *tail* pointer, increments (modulo) the *tail* pointer, and informs the associated sender of the new *tail* pointer.

At Step 2, the packet arrival interrupt handler examines the referred buffer to ensure it contains a valid packet before forming a receive sk_buff out of it. When a sender copies an Ethernet packet to a receiver, it encodes the updated *head* pointer, the packet length information and a valid bit flag in the packet's destination MAC address, so that the receiver could determine if a buffer holds a valid packet, derive the size of the transmitted Ethernet packet, and update the *head* pointer of the sender's associated receive buffer ring.

At Step 3, a remote memory access is required to inform the sender of the updated *tail* pointer, which the sender compares with the *head* pointer to detect the situation in which the associated receive buffer ring is filled up. To eliminate this remote memory access in every packet transfer, every machine caches the *tail* pointer of each of its receive buffer rings on remote machines. These cached *tail* pointers may be stale because, to decrease the code path length in the packet arrival interrupt handler, the EOP driver asynchronously allocates new buffers, updates the *tail* pointer and informs the sender of this update in the background. However, the EOP driver ensures that a *tail* pointer and its associated *head* pointer are always sufficiently far apart that such staleness is harmless.

With sender-side caching of the head and tail pointers of receive buffer rings and lazy *tail* pointer updates, every Ethernet packet transmission incurs only two remote memory accesses, one for the packet payload including the embedded EOP header information, and the other for the packet arrival interrupt. In addition, every once in a while, a machine needs to use a remote memory access to update the *tail* pointer cached at every other machine that previously sent packets to it. As a further optimization, when a sender sends a sequence of packets to a receiver, the sender could choose to issue only one packet arrival interrupt for multiple sent packets, essentially implementing *interrupt coalescing*. This requires the packet arrival interrupt handler to process as many received packets as possible when it is invoked.

3.5 Cross-Machine Memory Copying

Application-level cross-machine memory copying (CMMC) allows an application running on one machine to copy data to another application running on another machine connected to the same *Marlin* switch. The design pattern using CMMC works as follows.

1. The sender application first asks the receiver application for an application-level receive buffer of a certain size.
2. The receiver application allocates a page-aligned receive memory area locally, pins it down in memory, derives the physical page numbers of the physical pages backing this receive memory area, and returns to the sender application the pointer to this receive memory area, as well as the physical page number list.
3. When the sender application receives the response returned by the receiver application, it registers the receiver application's IP address, the receiver memory pointer and the physical page number list with its underlying OS.
4. The sender application makes a remote write system call to write to anywhere in this receive memory area in the receive application.

Three new system calls are created to support CMMC. An *address translation* system call is used at Step 2 for a receiver application to derive the physical page number list behind a page-aligned contiguous memory area. A *registration* system call is used at Step 3 for a sender application to register with the underlying OS a remote memory area and its corresponding physical page list, and receive a handle. A *remote write* system call is used at Step 4 for a sender application to write a local buffer to a remote memory area.

The input arguments to a remote write system call are a *handle*, which represents a remote memory area, a *target offset*, which represents the start position in the remote memory area for the write operation, a *source pointer*, which represents the start address of the local source buffer, and *length*. Given a remote write system call, *Marlin*'s CMMC implementation first converts the handle to the corresponding remote memory area, then identifies the sequence of physical page numbers behind the area to be written, and finally sets up a sequence of HRDMA operations using the source pointer and the target physical page numbers. The number of HRDMA operations required to implement a remote write system call depends on the degree of fragmentation, or the number of contiguous physical memory areas behind the source area and the target area. For example, if the source area and the target area of a CMMC operation are backed by 4 and 6 disjoint physical memory regions, respectively, this CMMC operation requires 6 HRDMA operations. Optionally, the remote write system call could include an option that sends a copy completion signal to the receiver application when all of the underlying HRDMA operations are completed.

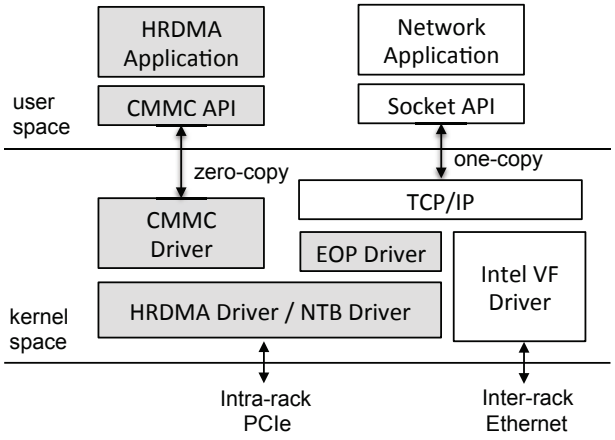

Figure 4: *The software architecture of* Marlin*'s compute host component, where the shaded parts are software introduced by* Marlin

4. PROTOTYPE IMPLEMENTATION

The hardware test-bed used in the *Marlin* prototype consists of five Intel X86 servers, one 10GE NIC, eight PLX PEX8732, two PLX PEX8717 switches with 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 not connected to the PCIe switch. All these servers are Supermicro 1U server equipped 8-core Intel Xeon 3.4GHz CPU and 8GB of memory.

Because of signal deterioration concerns [12], each of the two compute hosts is connected to a pair of PEX8732 devices to boost the signals, then a PEX8717 switch, then again connected to a pair of PEX8732 devices, and finally the PEX8748 switch. [1] 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.

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 *Marlin* prototype implementation is based on Fedora 15 with the Linux kernel version 2.6.38, and consists of a management host component and two compute hosts. The management host is responsible for enumerating the PCIe devices, including non-transparent PCIe switches and Ethernet NICs, sets up the BARs and translation registers on NTBs, and allocates and de-allocates virtual PCIe devices to compute hosts for I/O device sharing.

Figure 4 shows the software architecture of *Marlin*'s implementation on compute hosts, which offers two inter-host communications mechanisms: the Ethernet-over-PCIe (EOP) driver supports a socket-based communication mechanism that is meant to support legacy network applications, and the cross-machine memory copying (CMMC) driver is designed to expose *Marlin*'s zero-copy data transfer capability to user applications via a specially designed API.

[1]Using additional signal repeaters is a temporary system artifact in *Marlin*'s current test-bed. The production-grade switching such as PLX's Argo series does not require extra switches as signal repeaters.

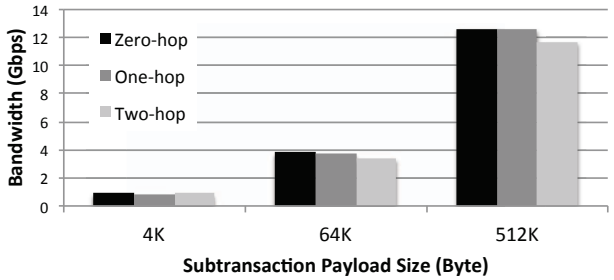

Figure 5: *The throughput comparison among zero-hop, one-hop and two-hop DMA operations with a single source-destination pair when the transfer size is 4KB, 64KB and 512KB*

The NTB driver is invoked at the system start-up time to set up the BARs and translation registers to map the management host's physical address space into a compute host's physical address space. The HRDMA driver sits below the CMMC and EOP driver and exposes a set of remote memory read/write semantics for the CMMC and EOP driver, and performs several optimizations to fully utilize the PCIe link's raw bandwidth.

5. PERFORMANCE EVALUATION

5.1 Hardware-based Remote DMA

The fundamental building block for *Marlin* is a DMA transaction, which could be *local*, where the source and destination memory areas belong to the same compute host, or *remote*, where the source and destination memory areas belong to different machines that are 1 hop (compute host and management host) and 2 hops (compute host to management host to compute host) away. Figure 5 shows the throughput comparison of a single DMA transaction consisting of a *single* source-destination pair of different transfer size when the source and destination memory areas are zero hop, one hop and two hops away. Each reported number is the average of 10 measurements. Except for small transfer size, the throughput of two-hop HRDMA is lower than that of one-hop HRDMA because the former incurs longer latency, and the throughput of one-hop HRDMA is comparable to zero-hop HRDMA because they incur approximately the same latency.

Marlin's HRDMA driver performs the following optimizations to maximize the HRDMA performance: batching, combining and exploitation of multiple DMA channels. Suppose there is a sequence of 10,000 4KB HRDMA requests from one attached compute host to another. In the *baseline* configuration, the HRDMA driver initiates a separate DMA transaction for each of these 10,000 HRDMA requests. In the *multi-channeled* configuration, the HRDMA driver initiates a separate DMA transaction for each of these 10,000 HRDMA requests using multiple DMA channels. In the *batched* configuration, the HRDMA driver exploits the scatter and gather capability of the DMA engine and initiates 40 DMA transactions, each of which consists of 256 subtransactions each with a transfer size of 4KB. The DMA engine pipelines the processing of the subtransactions inside a DMA transaction. In the *combined* configuration, the HRDMA driver combines 32 consecutive memory pages because they are contiguous, and initiates 40 DMA transactions, each of which consists of 8 subtransactions each with a transfer size of 128KB. The measured throughputs of these four configurations are shown in Table 1.

The drastic throughput difference between *baseline* and *batched* shows that batching is very effective because it significantly cuts

Baseline	Multi-Channel	Batched	Combined
1.54 Gbps	2.88 Gbps	20.17Gbps	20.31Gbps

Table 1: *The effectiveness of various optimizations supported by* Marlin*'s HRDMA*

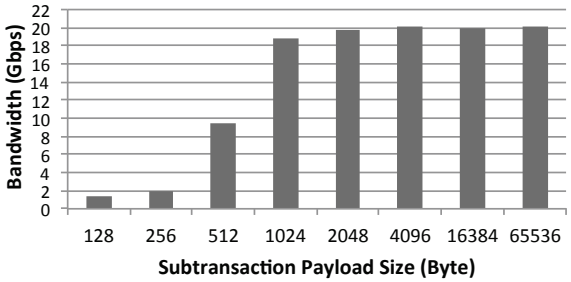

Figure 6: *The throughput comparison among DMA transaction runs that contain the same number of subtransactions but each subtransaction's payload size is varied*

down the number of DMA transactions and thus the total per-DMA transaction overhead. The tiny throughput difference between *batched* and *combined* suggests that the number of source-destination pairs in each DMA transaction does not matter much when the total transfer size per DMA transaction is the same. When consecutive DMA payload pages are physically contiguous, the HRDMA driver tries to combine such pages as much as possible. However, combining helps improve the throughput if it reduces the total number of DMA transactions but does not matter much if it just reduces the number of source-destination pairs in a DMA transaction. Finally, the substantial difference between *baseline* and *multi-channel* means that multiple DMA channels does help to mask part of the per-DMA transaction overhead when the DMA transaction size is small.

Considering that data transfer requests from EOP and CMMC may not have large and physically contiguous payloads, we further investigate the impact of memory fragmentation to the HRDMA throughput. We set up a DMA transaction that consists of 2,000 subtransactions, each corresponding to a payload of the same size, varied the per-subtransaction payload size from 128 bytes to 64 Kbytes and measured the total elapsed time of each DMA transaction run. Figure 6 shows the average throughput of 10 runs under different payload size. The HRDMA's throughput saturates at around 20.3 Gbps when the per-subtransaction payload size reaches 4KB. Because the PCIe switch used in our test-bed supports a maximum payload size of 128 bytes and up to 32 outstanding PCIe packets, the maximum amount of data outstanding in each DMA subtransaction is 4KB. Consequently, increasing per-subtransaction payload size beyond 4KB does not lead to any noticeable additional throughput improvement. [2]

5.2 Cross Machine Memory Copying

To measure the latency of CMMC, we created a sender program running on one compute host that writes 4 bytes using CMMC to a pre-defined memory location (say M_r) in a receiver program, which runs on another compute host, polls M_r and writes 4 bytes back using CMMC to a pre-defined memory location (say M_s) in

[2]The theoretical bandwidth of a 4-lane Gen3 PCIe link is 32Gbps, whereas *Marlin*'s HRDMA achieves 63.4% link utilization (20.3/32). Generally, closed to full link utilization could be achieved by increasing the maximum payload size [25, 38]. However, our current test-bed machines support up to 128 bytes.

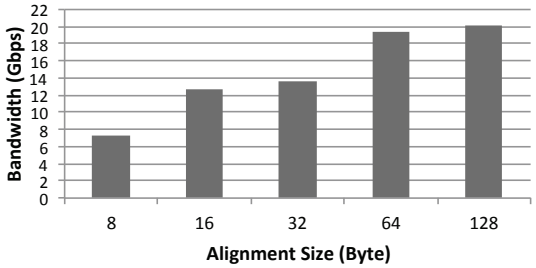

Figure 7: *The throughput comparison among different address alignments.*

the sender program immediately after detecting a change in M_r. The sender program takes a timestamp before writing the first 4 bytes, polls M_s after writing the first 4 bytes, takes another time stamp after detecting a change in M_s, and finally calculate the difference between these two timestamps. The resulting average round-trip delay for CMMC is 26.9 μsec, among which 11.4 μsec is spent on the two HRDMA operations. The latency penalty of CMMC beyond HRDMA is mainly attributed to virtual-to-physical address translation overhead and system call invocation delay.

To measure the throughput penalty of CMMC when compared with HRDMA, we wrote a sender program running on one compute host that allocates a 4MB contiguous physical memory area, and uses CMMC to copy this memory area to a 4MB contiguous physical memory area allocated by a receiver program, which runs on another compute host. The resulting average throughput for CMMC is 20.11 Gbps, which is 0.98% lower than the throughput of the underlying HRDMA operations (20.31 Gbps). This throughput penalty is again due to address translation and system call overhead.

5.3 Ethernet Over PCIe

To evaluate the performance of EOP, we first measured TCP throughputs using iperf [45]. Initially the measured TCP throughput is below 10 Gbps, which is much less than the underlying HRDMA throughput of 20Gbps. After further investigations, we found that the DMA engine's performance is very sensitive to the alignment of the source and destination addresses used in the DMA transactions. Therefore, we varied the degree of alignment of a DMA transaction's source and destination address but kept the payload size the same, and measured the HRDMA throughput. The results, shown in Figure 7, show that only when source/destination addresses are multiples of at least 64 bytes, the HRDMA performance is able to reach its maximum.

However, assuming the source buffer of an EOP transaction is always 64-byte aligned is impractical. The starting address of the Ethernet packet itself may not be 64-byte aligned because the protocol headers may be of variable length, e.g. both TCP and IP have optional fields. Fortunately, in Linux the gap between an Ethernet packet buffer's starting address and the largest 64-byte multiple that is smaller than the starting address, or *skb headroom*, still belongs to the allocated packet buffer and does not contain any meaningful data. An optimized version of the EOP driver exploits this invariant by using this largest and closest 64-byte multiple associated with a transmitted Ethernet packet as the source address of the corresponding DMA transaction, and embedding the alignment offset information in the headroom, if the transmitted Ethernet packet is not 64-byte aligned. When a user application transmits a sequence of source buffers that are not physically contiguous, modern OS such as Linux explicitly copies them into a contiguous kernel buffer

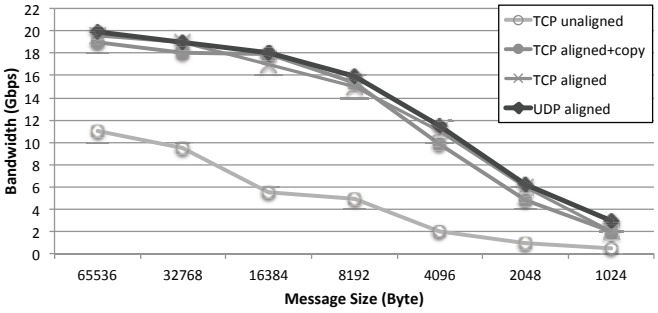

Figure 8: *The TCP/UDP throughput comparison between the vanilla and optimized versions of the EOP driver under MTU sizes.*

	Protocol	Bridges	NTB translation		Total
CPU	0.88 μs	5.2 μs	14.1μs		20.18μs
	Pre-processing	**Pre-DMA**	**DMA**	**Post-DMA**	**Total**
EOP	2.8 μs	11.4μs	11.5μs	3μs	28.7μs
CMMC	1 μs				26.9μs

Table 2: *The round-trip latency breakdown of sending a 4-byte remote memory read operation using CPU, an 64-byte ping packet using EOP, and a 4-byte remote memory write using CMMC.*

if the underlying NIC does not support scatter/gather DMA, or simply sets up a DMA subtransaction for each source buffer without data copying if the underlying NIC supports scatter/gather DMA. Although the DMA engine in our test-bed supports scatter/gather, the 64-byte alignment requirement suggests that it is more efficient to make a copy of these source buffers than to deal with their alignment issues one by one, especially when the size of these source buffers is small.

Figure 8 shows that when the source and destination address are 64 bytes aligned, the TCP and UDP throughputs of the vanilla EOP driver are effectively the same under different MTU sizes. The TCP and UDP throughputs are measured using iperf and nuttcp [22], respectively. However, when the source address is not aligned, the TCP throughput of the vanilla EOP driver is significantly worse than that when the source address is aligned, sometime by a factor of more than 2. The optimized version of the EOP driver effectively resolves this misalignment-induced performance problem by closing the TCP throughput gap between the aligned source address case and the unaligned source address case to less than 6% when the MTU is 64KB.

When the MTU is 64 KB and the source and destination address are aligned, the TCP and UDP throughputs are 19.6 Gbps and 19.9 Gbps, respectively, which are slightly lower than that of HRDMA (20.3 Gbps). To reach this level of performance, we increased the socket buffer size to 16 Mbytes, turned off SACK and turned on FACK [34]. The performance difference between HRDMA and UDP is due to additional data copying and context switching. The performance difference between UDP and TCP is due to packet losses at the interface between the IP stack and the EOP driver on the sender side. Because such packet losses tend to be bursty, FACK is better at handling such multi-segment losses than SACK.

5.4 Latency Analysis

Using the method in section 5.2, we measured the CPU-based round-trip remote memory write latency. The resulting average of two writes delay reports 17μsec, meaning roughly 8.5μsec one-way remote write latency. To better understand where time is spent, we measured the round-trip delay of a CPU-based memory *read* operation to a configuration register in a local NTB device, and that

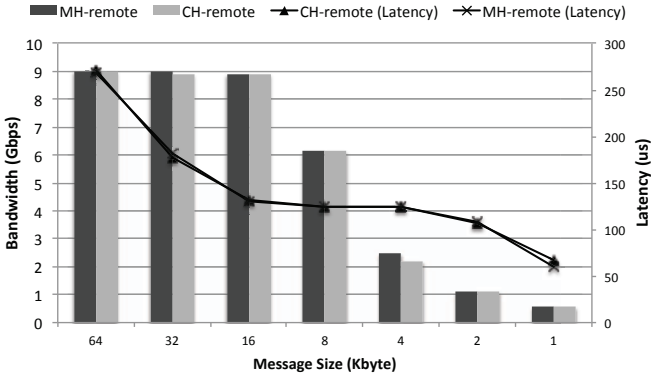

Figure 9: *Throughput and latency of inter-rack communication between a compute host attached to the* Marlin *switch and a remote server in another rack.*

to a remote host's memory location. The first memory read operation takes 2.48μs, which includes the traversal times through 16 PCIe bridges and such PCIe protocol overhead as TLP creation and processing. Because the traversal latency of each PCIe bridge is around 150 ns [11], 0.88μs of this memory read latency is due to PCIe protocol processing. Similar results were reported in [36]. The latency of a CPU-based memory read operation from one attached host to another attached host's memory incurs a round-trip delay of 20.18 μs shown in upper Table 2, which includes the traversal times of 56 PCIe bridges [3] and 4 NTBs, and the PCIe protocol processing overhead. By comparing the latencies of these two memory read operations, we concluded the per-NTB traversal time is 3.52μs, which is spent on PCIe device ID look-up, address translation, and creation of new TLP in the translated domain.

Table 2 shows the latency of sending a 64-byte packet using EOP and performing a 4-byte remote write operation using CMMC, and their break-down. Sending a packet using EOP involves the following steps:

1. *EOP pre-processing*: The EOP driver fetches the packet from the IP layer, and attaches a proper EOP header to it.
2. *Pre-DMA processing*: The HRDMA driver prepares a DMA descriptor for the packet and initiates a DMA transaction based on this descriptor.
3. *DMA transfer*: The DMA engine transfers the packet to the destination according to the DMA descriptor.
4. *Post-DMA processing*: After the DMA transaction is completed, a completion interrupt arrives to notify the sending EOP driver, which in turn triggers a packet arrival interrupt to prompt the receiving EOP driver to pick up the packet.

CMMC follows the same procedure except in the first step the sending CMMC driver performs a virtual-to-physical address translation, and in the last step the receiving CMMC driver delivers a signal to the receiver user application.

5.5 Inter-Rack Socket Communication

The management host of a *Marlin* switch allocates to each attached compute host a *Marlin* switch a VF on an SRIOV-capable

[3]Bridges along the path include on-board host bridges, eight PEX8732 switches, two PEX8717 NTBs, and one PEX8748 switch, with each consisting of 4, 16, 4, and 2 P2P bridges, respectively.

	Ethernet[1]	InfiniBand[2]	PCIe[3]
Total Bandwidth (Gbps)	10 × 24	20 × 24	32 × 24
Power per ToR Switch	176	130	90
Power for Adapters	5.5 × 24	15.5 × 24	4.9 ×24 + 5.5
Total Power (W)	308	502	213.1
Power/Bandwidth (W)	1.283	1.045	0.277

Table 3: *Power consumption comparison among Ethernet, Infiniband and PCIe as the interconnect for 24 servers.*

[1] 176W: Brocade TurboIron switch, 5.5W: Intel 82599 NIC.
[2] 130W: Mellanox InfiniScale III [6], 15.5W: Mellanox ConnectX MHQH19.
[3] 90W: PLX eFabric switch, 4.9W: PEX8717 NTB adapter [11].

NIC so that the compute host could communicate with a remote host using the standard socket communication interface. We measured the packet latency and throughput of a network connection from a host connected to the *Marlin* switch of one rack to a remote server in another rack, where the source is either a management host or a compute host. The remote server is equipped with another Intel SRIOV NIC, which is connected back-to-back to the SRIOV NIC attached to the PEX8748 switch. To measure the packet latency, we used the NetPIPE benchmarking tool, which employs ping- pong tests and measures the half round-trip time between two servers. To measure the packet throughput, we ran Iperf for 300 seconds and took the average of the highest 90% measurements reported by Iperf. Figure 9 shows the throughput and latency comparison between the following two configurations under various packet sizes: compute host (CH) to remote host and management host (MH) to remote host. The fact that the latency and throughput measurements of these two configurations are almost the same means that sharing an NIC over a PCIe network, as in the case of CH to remote host, does not introduce any noticeable throughput or latency penalty.

5.6 Power Consumption

We compare *Marlin* with existing blade servers in terms of network component expenses and power consumption. Consider a cluster of 24 servers, each of which is equipped with a 10GbE NIC and connected to a top-of-rack 10GbE switch. With server's CPU supporting on-board NTB and DMA channels [43], applying *Marlin*'s architecture immediately saves the cost of 23 NICs by allowing the 24 servers to share one 10GbE NIC via the PCIe switch fabric. The top-of-rack 10GE switch is replaced by the standard PCIe switch, which connects the 24 servers and the built-in layer-2 switch on the shared 10GE NIC, and forwards packets among these 24 servers.

Reducing the number of I/O devices cuts down the hardware, the electricity, and cooling cost, as well as the space requirement. Table 3 lists the power consumption of *Marlin*, compared with Ethernet-based and InfiniBand-based clustering technology. We compare the total power consumption per gigabit bandwidth of a 24-port Ethernet, InfiniBand, and PCIe switch, and their adapters. *Marlin* proves to be a more competitive solution because its I/O sharing and consolidation design affords at least 4 times reduction in power consumption.

6. RELATED WORK

Rack disaggregation, advocated recently by the Virtual I/O group in Open Compute Project [3, 8, 9], proposes open hardware design model to combine and re-combine compute, memory, networking, and storage components as a new path to greater efficiency. By virtue of SAN and technologies such as FCoE, and iSCSI, storage system is ubiquitously disaggregated from the individual server's hardware. To disaggregate networking components, PCI-SIG defines Multi-Root I/O virtualization (MRIOV) [14], which connects multiple hosts (root complexes) and endpoints with the MRA (Multi-Root Aware) PCIe switch to form a multi-host PCIe fabric. The PCIe-based I/O devices under MRA switch are decoupled from the individual server and shared among multiple hosts by attaching to a virtual PCIe hierarchy. Unfortunately, due to lack of MRA switch in the market, many proprietary attempts have been made to achieve multi-root sharing using SRIOV compliant I/O devices with customized hardware boxes [4, 7, 10, 28, 44] or relying on NTB [32, 40, 41] as an alternative. Besides, PCIe could be an ideal power-efficient interconnect for intra-rack communications [5, 20, 26], by mapping parts of the client host's memory to a global memory address space [23, 24]. As a point of comparison, *Marlin* maps the hosts' *entire* memory address space to avoid extra data copying and guarantees security by leveraging existing hardware components.

SeaMicro [1,39] proposes a 10U system with disaggregated compute, storage, and network elements. Up to five petabytes of storage and single 10Gbps Ethernet uplink are exposed to each core. EnergyCore system [2, 13] is another I/O disaggregation built on ARM Cortex A9 cores. Its EnergyCoreFabric virtualizes Ethernet interfaces so that the EnergyCards don't need physical network ports. Finally, memory disaggregation allows memory resources to be consolidated in a memory blade. *Marlin*'s memory-based addressing model is native for memory sharing between hosts. However, without installing custom hardware, it's difficult to maintain cache coherence between local memory and remote memory [29–31].

Many studies leverage user-level direct hardware access as a means of achieving high bandwidth transfers and avoid OS's software overheads, such as context switches and data copying [19,21,37,42,47]. *Marlin*'s CMMC defines the minimum set of interfaces, allowing the user applications to do directly remote memory access. Compared with Intel's PCIe NTB Ethernet driver which uses the sender side CPU to copy payload and requires additional payload copying at the receiver side [33], *Marlin*'s EOP saves the CPU cycles by offloading the operation to the HRDMA and the global memory address allows the sender to directly place the payload into the receiver's buffer, avoiding additional payload copying.

Marlin is capable of mapping the main memory of all the hosts connected to a Marlin switch into a global memory space, and then use a normal DMA engine to copy data within this global memory space efficiently and securely. This approach is architecturally different from InfiniBand (network-oriented) and from SHRIMP (hybrid of network and shared memory) [18]. *Marlin* uses similar ideas in UDMA to reduce the DMA setup overhead [17], but is not as aggressive in pushing these low-level software optimizations. Additionally, the PCIe networking approach of *Marlin* enables not only hardware-based RDMA, but also direct remote I/O device access. That is, one host can directly access another host's disks or SSDs. Most previous works in low-latency networking work did not support such a capability. Finally, as a comparison to Ethernet, *Marlin* enjoys the reliable transmission service over PCIe because PCIe protocol supports end-to-end retransmission and credit-based flow control.

7. CONCLUSION

The emerging *disaggregated rack* architecture requires direct shared accesses from multiple hosts to I/O devices such as SAS controllers or NICs, so as to decouple I/O device upgrades from CPU/memory upgrades. At this point PCIe is the most promising technology to support such I/O device sharing. Moreover, we show in this pa-

per that PCIe could also double as an effective rack area network for both intra-rack and inter-rack communications, without changing the higher-level network software stack. Compared with other system area network technologies, PCIe is more power-efficient because its transceiver is designed for short distance connectivity and thus is simpler and consumes less power. PCIe's memory-based addressing model is especially interesting because it enables one machine to directly address any memory location of another machine. *Marlin* exploits this capability to build a remote DMA mechanism that is truly hardware-based and requires no software involvement in payload transfer. On top of this remote DMA mechanism *Marlin* supports an Ethernet-over-PCIe (EOP) interface for socket-based communications, and a cross-machine memory copying (CMMC) interface for zero-copy application-to-application data transfer. Combining all these technology components leads to a new top-of-rack switch architecture that automatically allows intra-rack communications to go through the PCIe network and inter-rack communications to go through the standard Ethernet. The specific research contributions of this work thus include

- A comprehensive rack area architecture that is built on the same design principles underlying modern software defined networks and effectively combines PCIe network and Ethernet technologies into a coherent whole,
- A hardware-based remote DMA mechanism that allows one application on one machine to copy data to another application on another machine in a secure and efficient way,
- A non-cache-coherent shared memory abstraction that enables a global lock mechanism and a direct inter-CPU interrupt mechanism among nodes connected to a *Marlin* switch, and
- A detailed performance evaluation of the proposed rack disaggregation architecture and the associated intra-rack and inter-rack inter-machine communication method based on a fully operational prototype.

8. REFERENCES

[1] AMD SeaMicro SM15000 Fabric Compute Systems. http://www.seamicro.com/SM15000.

[2] Calxeda ECX1000 Product Brief. http://www.calxeda.com/wp-content/uploads/2012/06/ECX1000-Product-Brief-612.pdf.

[3] Intel shows off Rack Scale Architecture and Rack Disaggregation plans. http://semiaccurate.com/.

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

[5] kontron VXFabric - PCI Express Switch Fabric for High Performance Embedded Computing. http://www.kontron.com/vxfabric_whitepaper.

[6] Mellanox InfiniScale III. http://www.mellanox.com/related-docs/prod_siliconPB_InfiniScale_III.pdf.

[7] Micron I/O Virtualization White Paper, Micron Technology, Inc. http://www.micron.com/~/media/Documents/Products/White%20Paper/micron_io_virtualization_wp.pdf.

[8] Open Compute Project. http://opencompute.org/.

[9] Open Compute Project Virtual IO Charter. http://www.opencompute.org/wp/wp-content/uploads/2012/10/Open_Compute_Project_Virtual_IO_Charter_2012-09-10.pdf.

[10] PCI Express System Interconnect Software Architecture for x86-based Systems. http://www.idt.com/.

[11] PEX 8717, PCI Express Gen 3 Switch, 16 Lanes, 10 Ports. www.plxtech.com/download/file/2221?

[12] The Case for PCIe 3.0 Repeaters, PCI-SIG Developers Conference, 2011.

[13] The opposite of virtualization: Calxeda's new quad-core ARM part for cloud servers. http://arstechnica.com/.

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

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

[16] 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*.

[17] M. A. Blumrich, C. Dubnicki, E. W. Felten, and K. Li. Protected, user-level dma for the shrimp network interface. In *High-Performance Computer Architecture, 1996. Proceedings. Second International Symposium on*, pages 154–165. IEEE, 1996.

[18] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. *Virtual memory mapped network interface for the SHRIMP multicomputer*, volume 22. IEEE Computer Society Press, 1994.

[19] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the hamlyn sender-managed interface architecture. *ACM SIGOPS Operating Systems Review, 1996*.

[20] J. Byrne, J. Chang, K. T. 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.

[21] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *Micro, IEEE*, 18(2):66–76, 1998.

[22] B. Fink and R. Scott. nuttcp, v5. 3.1, 2006.

[23] R. Gillett and R. Kaufmann. Using the memory channel network. *Micro, IEEE*, 17(1):19–25, 1997.

[24] R. B. Gillett. Memory channel network for pci. *Micro, IEEE*, 16(1):12–18, 1996.

[25] I. Granovsky. Optimizaing PCIe Port Performance, 2006.

[26] T. Hanawa, T. Boku, S. Miura, M. Sato, and K. Arimoto. Pearl: Power-aware, dependable, and high-performance communication link using pci express. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 284–291. IEEE, 2010.

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

[28] 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.

[29] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *Cluster Computing, 2005. IEEE International*, pages 1–10. IEEE, 2005.

[30] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 267–278. ACM, 2009.

[31] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

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

[33] J. Mason. Intel PCIe NTB Driver: ntb_tx_copy_task and ntb_rx_copy_task . `http://lxr.linux.no/linux+v3.9/drivers/ntb/ntb_transport.c`.

[34] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining tcp congestion control. *ACM SIGCOMM Computer Communication Review*, 26(4):281–291, 1996.

[35] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[36] D. J. Miller, P. M. Watts, and A. W. Moore. Motivating future interconnects: a differential measurement analysis of pci latency. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 94–103. ACM, 2009.

[37] R. OpenFabrics. Protocols through ofed software.

[38] PLX. Draco DMA Performance, 2013.

[39] A. Rao. Seamicro technology overview. Technical report, Technical report, SeaMicro, 2010.

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

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

[42] L. Rizzo. netmap: a novel framework for fast packet i/o. In *USENIX ATC*, 2012.

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

[44] 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.

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

[46] C.-C. Tu, C. tang Lee, and T. cker Chiueh. Secure i/o device sharing among virtual machines on multiple hosts. In *ACM ISCA'13*.

[47] T. Von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: a user-level network interface for parallel and distributed computing (includes url). In *ACM SIGOPS Operating Systems Review*, volume 29, pages 40–53. ACM, 1995.

[48] 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.