Optimizing Virtualized I/O Processing over High Speed Networks

Guangdeng Liao and Laxmi N. Bhuyan

Abstract

As IT industry is shifting to cloud computing, virtualization has become the key enabling technology for server consolidation and elastic resource management in cloud infrastructures. On the other hand, increasing networking demands of cloud workloads is driving network speed rapidly migrate to 10Gbps and beyond. High speed networks require general purpose servers to provide highly efficient network processing in virtualized environment.

In this paper, we start with detailed performance analysis to fully understand I/O virtualization performance on mainstream severs with 10GbE. Our experiments reveal major challenges faced by I/O virtualization over high speed networks: data movement and packet switching. Then we propose two optimizations for the scheduler inside Virtual Machine Monitor (VMM) to optimize data movement. In order to avoid cache misses on packets, we propose integrating a data movement engine considering VMM scheduling information to inject packets into right cores where corresponding guest domains are running. Lastly but not least, we also design a simplified bridge to do packet switching. Our combined optimizations significantly reduce major bottlenecks in virtualization environment and improve performance up to xx%.

1. Introduction

Virtualization technology separates hardware and software management and offers many useful features including functional isolation, server consolidation and live migration [1, 4, 7]. In addition to these capabilities, it also provides a strategy to more fully utilize the vast compute cycles available on commodity multi-core platforms. For these reasons, virtualization technology is rapidly gaining popularity in cloud computing and have been key enabling technologies in cloud infrastructures like Amazon EC2. This trend toward server consolidation where multiple virtual machines (VM) share a machine's resources naturally includes sharing of network interface cards (NIC). In such an environment, the virtual machine monitor (VMM) virtualizes the machine's NIC to give the illusion of ownership to

multiple operating systems running concurrently in different VMs. Thus, server consolidation promises not only full utilization of compute resources, but also of available network bandwidth. As more and more network-active servers are consolidated in cloud computing, we see a strong motivation toward high performance networks in the virtualized endpoint server.

Ethernet continues to be the most widely used network architecture today for its low cost and backward compatibility with the existing Ethernet infrastructure. It dominates in data centers and is replacing specialized fabrics such as InfiniBand [13], Quadrics [29], Myrinet [2] and Fiber Channel [3] in high performance computers. Driven by increasing networking demands of cloud workloads such as Internet search, video servers and web hosting etc, network speed rapidly migrates from 1Gbps to 10Gbps and beyond [8]. High speed networks require general purpose servers to provide highly efficient network processing in virtualized environment. Unfortunately, traditional designs of processors, cache hierarchies and systems focused on CPU/memory-intensive applications, and have often been decoupled from I/O considerations, thus being inefficient for network processing in virtualized environment.

In this paper, we conduct detailed performance analysis of virtualized network processing over 10GE based on Xen [1] to fully understand its performance bottlenecks. We also quantify performance due to architectural events and overheads in the processor by using hardware performance counters to instrument the functions along the packet processing path. Our results show that virtualized network processing only achieves 3.5Gbps of network bandwidth. We also notice that a virtual switch to de-multiplex packets has surprisingly high switching cost. The data movement in virtualized environment is more complicated than native environment and needs extra consideration.

Motivated by our performance analysis, we develop two VMM scheduler optimizations to improve data movement performance by co-scheduling backend driver and frontend driver into the same cache domain and stealing credits from idling VCPUs to favor I/O VCPUs. In order to avoid cache misses on packets, we propose integrating a small hardware data movement engine. The engine considers VMM scheduling information and can inject packets into cores where corresponding guest domains are running. Since packet switching has surprisingly high switching cost, we also design a simplified bridge to de-multiplex packets to corresponding guest domains. Our combined optimizations significantly reduce major bottlenecks in virtualization environment and improve performance up to xx%.

To evaluate our designs, we implemented all system optimizations in Xen on Intel servers. We also enhanced the full system simulator Simics [24] with detailed embedded timing to evaluate the performance benefits of the data movement engine on virtualized I/O processing. We developed a 10GbE NIC as a device module of the simulator and a corresponding driver with the support of Large Receive Offload (LRO) [8] in Linux. In the experiments, both the micro-benchmark Iperf [14] and the macro-benchmark SPECWeb [33] are used. Experimental results demonstrate that our optimizations improve the network processing efficiency by xx% and web server throughput by xx% without introducing high hardware complexity.

The major contributions in this paper can be summarized as follows:

- We did a detailed performance analysis of virtualized network processing performance and revealed its major performance bottlenecks.
- We developed two VMM scheduler optimizations and proposed a small hardware data movement engine to address the data movement challenge in virtualized environment.
- We implemented a simplified bridge to reduce packet switching overheads.

The remainder of this paper is organized as follows. We revisit network I/O virtualization in Section 2 and then present a detailed overhead analysis over 10GbE in Section 3. Section 4 elaborates our optimizations in detail, followed by performance evaluation in Section 5. Finally we discuss related work and conclude our paper in Section 6 and 7, respectively.

2. Network Virtualization

Virtualization is a broad term that refers to the abstraction of physical computer resources. A typical virtualized platform consists of a software virtual machine monitor that “virtualizes/abstracts” the physical resource of the platform and provides a simulated environment that appears to the operating system as hardware. Network virtualization was invented and implemented in IBM’s System/360 and System/370 [31]. Each virtual machine in these initial virtualized architectures was exclusively assigned a particular set of physical devices. Data transfer relied on channel programs executing in the VMM, which ensured resource isolation.

In this paper, we choose Xen as our study, the most popular open source para-virtualized virtual machine monitor. The network I/O architecture used in Xen is illustrated in Figure 1. It provides each

guest domain with a number of virtual network interfaces called Frontend, which are used by the guest domain for all its network communications. Corresponding to each virtual interface in a guest domain, a backend Interface is created in the driver domain, which acts as the proxy for that virtual interface in the driver domain.

Once a packet arrives at the NIC, NIC delivers the packet into main memory through DMA transactions over PCI-E bus and then generates a physical interrupt. Once the VMM receives the interrupt, it forwards to the driver domain. When the driver domain acquires CPU, the *NIC driver* delivers received packets in form of SKB data structure to *Linux Bridge*. Linux Bridge functions as an internal crossover Ethernet bridge to switch the received packets to the corresponding attached *backend driver* through their MAC address. The backend diver employs a data copy mechanism by default to directly copy data from the driver domain address space to DMA buffers provided by guest domain. In order to ensure that DMA buffers from guest domain do belong to guest domain address space, guest domain should first provide DMA buffers to VMM via grant table operations for memory protection and address translation. While the backend driver is moving packets among domains, it directly calls in VMM to move data into the corresponding DMA buffer. Once the packet reaches the guest domain, the backend driver requests the VMM to send a virtual interrupt to notify the target domain of the new packet. Then, the packet is processed from the *frontend driver* and network stack inside the kernel space to applications in the user space of guest domain as if it had come directly from the physical NIC. Note that the frontend driver employs polling model to receive incoming packets.

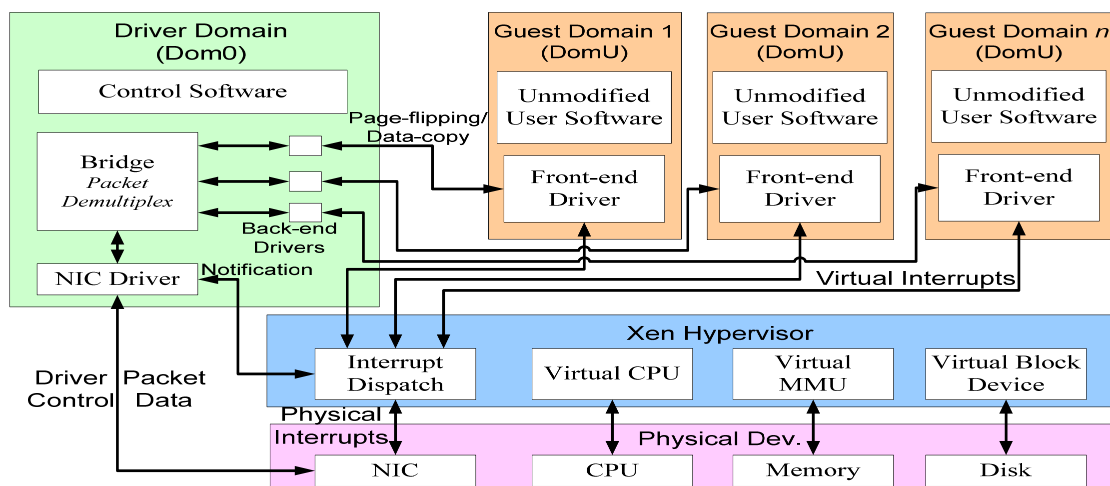

Figure 1. Virtualized I/O Architecture (Xen)

3. Understanding Virtualized Network Processing Overhead

In this section, we conducted extensive experiments to understand virtualized network processing overheads on Intel mainstream servers with 10GbE. Our testbed consists of a pair of server (system under test) and client hosted by Xen 3.1.3 and Vanilla Linux kernel 2.6.21 respectively. Each of them is an Intel server, which contains two Quad-Core Intel Xeon 5335 processors [15]. Each core is running at 2.66GHz frequency and each processor has 2 last level caches (LLC) of 4MB each shared by 2 cores. The servers are connected by two PCI-E based Intel 10Gbps XF server adapters [12]. We retain default settings of the Linux network subsystem and the driver, unless stated otherwise.

In the experiments, the micro-benchmark Iperf with 8 TCP connections is run and its server is inside a guest domain on system under test. Since in current implementation, backend driver has not been parallelized and guest domain does not support receive side scaling (RSS)[32], we only configure the guest domain with one virtual CPU. We find from our experiments that network processing in virtualized environment only achieves 3.5 Gbps bandwidth while saturating two physical cores (assuming ideal implementation of parallelized backend driver and RSS in guest domain, almost 6 cores are required for a line rate bandwidth). The high processing overhead motivates us to breakdown the per-packet processing overhead. In this section, we choose a typical I/O size 16KB as our case study. Note that I/Os are not packets over Ethernet and large I/Os larger than MTU are segmented into several packets (<=MTU). With 16KB I/O size in our experiments, per packet size on average is about 1.5KB.

Table 1. Component description

Component	Description
Driver	Default 10GbE NIC driver, same as native Linux
Buffer management	SKB buffer allocation/release, same as Native Linux
Linux Bridge	De-multiplexing/Multiplexing packets into corresponding BE.
Backend driver (BE)	Acts a proxy in driver domain for a guest domain and communicates with FE
Event operations on I/O channel (event-ops)	Communicate request/response information among BE and FE
Domain copy (domain-copy)	Copy packets among driver domain and guest domain
Frontend driver (FE)	Virtual NIC driver for guest domain
TCP/IP	The TCP/IP protocol stack, same as Native Linux.
Kernel-to-user data copy (user-copy)	After TCP/IP processing, data is moved out from kernel to user buffers, same as Native Linux.
Iperf	A user level benchmark to test TCP/IP capability.
Others	VMM scheduling, context switch, hypervisor calls and system calls etc.

3.1 Per-packet processing overhead

We use the tool Xenoprof [25] to collect system-wide function overheads while Iperf is running inside a guest domain over 10GbE. Along the network processing path in virtualized environment, we group all profiled functions into components. Those components are listed and explained in Table 1. Per-packet processing time breakdown is calculated and illustrated in Figure 2.

We obtain the following observations from Fig.2: 1) unlike native environment, packet movement in virtualization environment becomes much more complicated. It consists of packet movement from the driver domain to guest domain (denoted as *domain-copy*) and from kernel to user buffers inside guest domain (denoted as *user-copy*). They take around 25% and 15% of the whole packet processing time, respectively. Although packets reside in caches after *domain-copy*, *user-copy* still consumes many CPU cycles. That is because that the current VMM scheduler usually schedules driver domain and guest domain into two cores without sharing a LLC. When we manually ping guest domain and driver domain into the same cache domain (cores with a shared LLC), we notice that the *user-copy* overhead can be reduced largely. 2) Besides packet movement, *Linux Bridge* used for switching packets into corresponding backend drivers burns 1600 cycles per packet, thus becoming another major bottleneck. Although some other components (e.g. NIC driver, SKB buffer management, TCP/IP protocol stack) also consume some overheads, they are not related to virtualization and some existing optimizations for native environment like SKB recycling, TCP onloading can be applied to reduce those overheads [20]

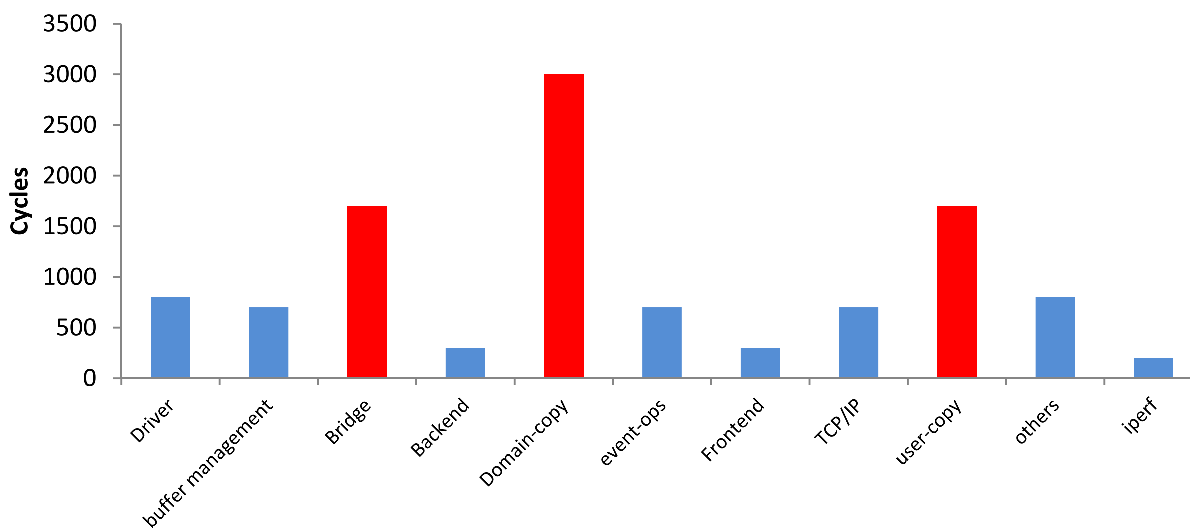

Figure 2. Per-packet processing overhead in virtualized environment

3.2 Architectural Analysis

In order to analyze the functional level overhead, we design a profiling methodology and develop a tool. Our tool can be used to quantify performance from the architectural characterization perspective. It instruments the VMM, driver domain, guest domain and network protocol stack along with the packet processing path. We adopt a performance counter based approach, where a small piece of code is manually inserted into the points of interest. Those code records the current time-stamp, retired instruction, L1 cache miss, L2 cache miss and TLB cache miss information of the measure point into a buffer using the corresponding Intel Performance counter. The overhead of the instrumental code is small (only 90 CPU cycles for a timestamp read and 70 cycles for a performance counter read) and is subtracted from the measurement.

One example getting L2 cache event count while running in *handle_bridge* (in *Linux Bridge*) routine is shown in the Table 2. It usually consists of two steps: set counter to select the architectural event of our interest and access performance counter to read the corresponding event count. In the left column of Table 2, we select the L2 cache miss event via writing into performance control register the corresponding encode value which is specific on Intel Core micro-architecture [15]. Once architectural event is selected, the right column attempts to read L2 miss event count via reading the corresponding performance counter. This subsection presents detailed architectural analysis for major components: *Linux Bridge*, *domain-copy* and *user-copy*.

Table 2. Performance counter example

Setting Counter	**Reading Counter**

//Enable Counter
set_in_cr4(X86_CR4_PCE);
val = 0x474024;
//Setting L2 Cache Event
wrmsr(0x186, val, 0);
	rdl2miss(){
	// read performance counter
	rdpmc(0,low, high);
	}
	Void handle_bridge() {
	//Reading L2 cache count
	Bridge_l2miss=rdl2miss();
	}

A) Linux Bridge

Linux Bridge is a way to connect two segments together in a protocol independent way [23]. Packets are forwarded based on Ethernet MAC address. The Linux Bbridge code implements a subset of the ANSI/IEEE 802.1d standard. In order to simplify the VMM design, Xen takes advantage of the existing Linux Bridge component in Linux Kernel to serve as a de-multiplexer. From Fig.2, we notice that 1600 cycles are consumed in the Linux Bridge module to switch each received packet to the designated

backend driver. It has surprisingly significant overhead and would perform much worse with integrating some filter rules. In this subsection, we architecturally breakdown the switching overhead for each packet and present results in Figure 3.

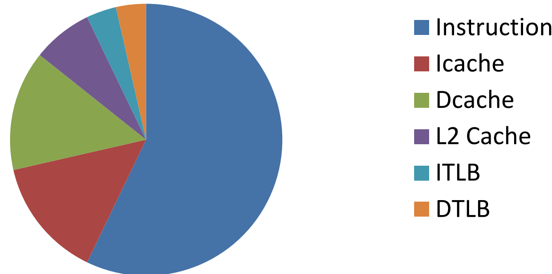

Figure 3. Linux Bridge overhead breakdown

We find from Fig. 3 that the biggest contributor of the Linux Bridge overhead is long path instruction execution, followed by data cache misses and instruction cache misses. That is because Linux Bridge was designed as a sophisticated firewall and switch framework to check with many plugged network filters/rules. We continue doing functional level profiling of Linux Bridge and list functional overheads in Table 3. We realize that functions relevant to network filter framework consume most of CPU cycles without any plugged filters and the core switching function itself (*Br_forward*) only requires 600 cycles. All of these observations indicate that a much simpler software switch is required for virtualization.

Table 1. Functional overhead in Linux Bridge

Functions/Macros	**Description**	**Execution time per packet (cycles)**
Handle Bridge	Bridge interface to NIC driver	100
Br_handle_frame	Netfilter framework to check with inserted filters/rules	400
Br_handle_frame_finish	Netfilter framework to check with inserted filters/rules	200
Br_forward	Performing switching functionality using Jhash algorithm	600
Br_forward_finish	Netfilter framework to check with inserted filters/rules	200
Br_dev_queue_push_xmit	Interface to backend driver	100

B) *Domain-copy*

After a packet is switched into a corresponding backend driver, it needs to be copied out from driver domain to guest domain address space. VMM provides a grant copy operation which maps the page, copies the packet and unmaps the page in a single hypercall. During a grant copy operation, VMM creates

temporary mappings into VMM address space for both source and destination of the copy. The VMM also pins (i.e. increment a reference counter) both pages to prevent the pages from being freed while the grant is active. We architecturally breakdown the domain-copy overhead for each packet and present results in Figure 4.

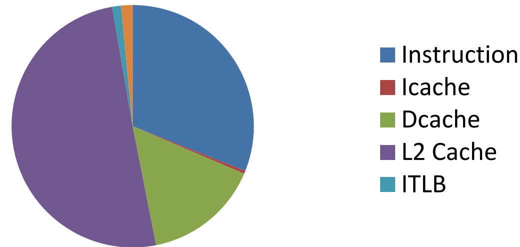

Figure 4. Domain-copy overhead breakdown

As shown in Fig.4, L2 caches misses and long instruction execution path are major contributors to high overheads in domain-copy. Since DMA transactions trigger cache invalidation to maintain cache coherence among caches and memory, Domain-copy incurs mandatory cache misses on packets and thus consumes a large number of CPU cycles. In order to copy packets between two domain address space, driver domain relies on grant table copy operations provided by VMM. The grant table operation consists of VMM enter/exit, page mapping/unmapping and expensive atomic instructions on the grant table, explaining high instruction execution overhead.

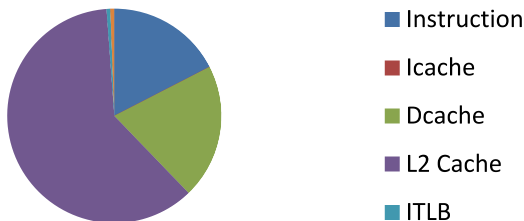

Figure 5. Kernel-to-user data copy overhead breakdown

C) *user-copy*

After protocol processing, user applications in guest domain are scheduled to copy packets from in-kernel SKB buffers to user buffers. We study its architectural overhead breakdown as shown in Figure 5. Fig.5 shows that L2 cache misses are the major overhead (~57%, ~3.5 L2 misses/packet), followed by data cache misses (~23%, ~50 misses/packet) and instruction execution (~17%). Although domain-copy

already fetches packets into caches, driver domain and guest domain are usually scheduled by VMM to run on two cores without sharing a LLC, thus still incurring L2 cache misses during the kernel-to-user copy. Existing optimizations like memory copy engine [34] on data copy in native environment help little in virtualized environment. Memory copy engine moves data in memory but the movement here is among separate caches. DCA injects data into cores where driver domain is running and cannot avoid those cache misses during data copy from kernel-to-user buffers. Thus, a new data movement scheme is required to avoid high packet movement overheads in virtualized environment.

4. Optimizations

The credit scheduler is designed to load balance workloads on multi-core platforms. Unfortunately, it tends to schedule driver domain and guest domain to cores without sharing a last level cache, incurring high packet movement overheads as shown in Section 3. In this section, we start with detailed study of credit scheduler and then propose two VMM scheduler optimizations to improve network processing performance in virtualized environment.

4.1 Credit Scheduler in VMM

VMM functions as an abstraction layer of the real physical devices. As a result, scheduling in virtualization is based on Virtual CPUs (VCPU) because Physical CPUs (PCPU) are transparent to domains. Each domain can be arbitrarily allocated with multiple VCPUs. Besides the default credit scheduler, VMM also keeps its legacy scheduler Simple Earliest Deadline First (SEDF) [22]. SEDF provides weighted CPU sharing in an intuitive way and uses real-time algorithms to ensure real time guarantees. However, it lacks global load-balancing on multiprocessors and is becoming obsolete. In this study we focus on the default credit scheduler [5], a proportional fair share CPU scheduler built to achieve load balance on SMP hosts. Its overall objective is to allocate the processor resources fairly.

The scheduler organizes a local run queue of online runnable VCPUs for each PCPU and always picks a workload (VCPU) from the head of the queue to run. This queue is sorted by VCPU priority. A VCPU's priority can be one of three values: OVER, UNDER and BOOST. OVER, UNDER represents whether or not this VCPU has used up its fair share of CPU resource in the ongoing accounting period. The BOOST state provides a mechanism for domains to achieve low I/O response latency. All the VCPUs in BOOST state are placed in front of those in UNDER state in the runqueue, while those with OVER state are kept

in the tail portion. Based on the predefined weight, each domain is initially allocated a corresponding credit which is fairly shared among all the VCPUs that are affinitized to the domain. As a VCPU runs, it consumes credits. Every so often, a system-wide accounting thread re-computes how many credits each active domain has earned and bumps the credits.

When it comes to multi-core architecture, there are a few twists while the scheduler functions. First of all, when there is not a VCPU of priority UNDER on a PCPU's local run queue, the scheduler will search other PCPUs for one. This load balancing ensures each domain receives its fair share of PCPU resources system-wide. Before a PCPU goes idle, the scheduler will look on other PCPUs to find any runnable VCPU. This guarantees that no PCPU idles when there is runnable work in the system. Secondly, VCPU migration might happen based on priority difference for event notification. Whenever an event is notified to a target VCPU while it is idle, the scheduler tickles the designated PCPU and re-evaluates to see if the target VCPU preempts the current running VCPU. If there are at least two runnable VCPUs in that PCPU, the scheduler would migrate some of them to the idlers in the system to achieve load balance. Last but not the least, the scheduler checks the state of the current running VCPU during each timer interrupt and redistributes the PCPU if necessary. The running VCPU will be migrated to the online neighbor PCPU with the most idling neighbors PCPU. This policy distributes work across distinct sockets first and then distinct cores in the same socket.

4.2 Cache-aware Scheduler

The default credit scheduler is unaware of core topology in multi-core systems, where some of cores are sharing a last level cache (LLC) while others are sitting in different sockets. It blindly migrates the VCPU running on PCPU with high workloads to PCPU with lightweight workloads.

To make the best use of the resource and to make inter-core communication efficient, cores in a physical package share some of the resources. Our system under test (SUT) has two CPU cores sharing the L2 cache which is called Intel Advanced Smart Cache [15]. Each processor has four cores in a physical package with two L2 caches. Each L2 cache is shared by two cores. The current credit scheduler is designed for SMP load balance, but is not cache-aware and cannot co-schedule the two VCPUs with data sharing on the two cores sharing L2 cache (a.k.a. cache domain). Since Dom0 is designed for serving I/O requests to de-multiplex packets and move packets to designated DomU (I/O DomU), there is intense

data sharing between Dom0 and I/O DomU. Co-scheduling Dom0 and I/O DomU in the same cache domain will give I/O DomU a free ride to access the data in the cache and avoid cache misses on packets.

In order to co-schedule Dom0 and I/O DomU, the first step is to identify them in the VMM. Currently we identify them by counting how often I/O events of boosting VCPUs are triggered during each time slice. If the number of triggers exceeds a threshold (default 150), both the boosting and the boosted VCPUs are considered as I/O VCPUs (in receive side, boosting VCPU is I/O VCPU in Dom0). Note that our extension of the scheduler is only based on VCPUs with intense I/O operations, and doesn't sacrifice the system-wide load-balance on multi-core platforms. After the identification of I/O VCPUs, the VMM scheduler always intelligently schedules boosting and boosted VCPUs to the cores sharing same L2 cache.

In default credit scheduler, when an event is notified to a target VCPU while it is idle, it is awaken with the state of BOOST. Then other idle PCPUs and PCPU hosting the VCPU are notified to re-evaluate where the VCPU will be running. In cache-aware scheduler, instead of notifying all idle PCPUs, VCPU with the state of BOOST is inserted into the runqueue of PCPU sharing L2 cache with the PCPU currently hosting boosting VCPU. An example is shown in Figure 6. The left side in the figure is the original system state where boosting VCPU and one running VCPU are sitting in the same cache domain and boosted VCPU is running on the core 4. Cache-aware scheduler will automatically migrate boosted VCPU into the same cache domain as boosting VCPU to take advantage of shared cache. The running VCPU is preempted into the core 4 for securing the system level load balance. The system state after migration is shown in the right side of the figure.

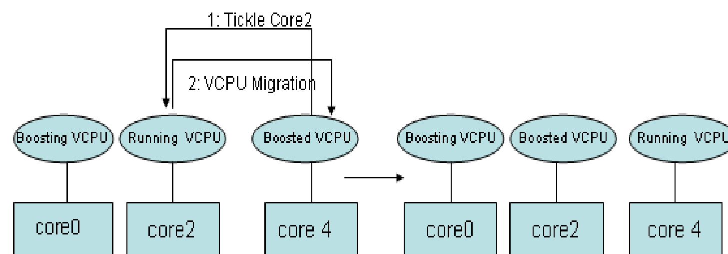

Figure 6. An example of Cache-Aware scheduler

Additionally, VCPU migration in current scheduler also occurs when a VCPU remains BOOST for a while and some PCPUs are idle. It chooses the target PCPU with the largest number of idle neighbors in its grouping. This option will distribute workload across distinct packages first and result in maximum

resource utilization since there is no shared resource contention. However, virtualized network processing with data sharing between Dom0 and I/O DomU will suffer heavy inter-package communication penalty from this mechanism. Cache-aware scheduler dynamically migrates the boosted VCPU and boosting VCPU to the same cache domain when this migration is triggered. Although our technique might preempt the running VCPU on the PCPU, the preempted VCPU could be migrated into other PCPUs to sustain system-level workload balance on multi-core platforms.

4.3 Credit-Stealing for I/O VCPU in Dom0

The number of VCPUs in Dom0 is configured by default as the number of cores in the platform. In credit scheduler, all VCPUs affiliated to the same domain are allocated fairly with the same credit. However, all of the interrupts from NIC are usually directed to a specific VCPU to improve the cache locality of interrupt processing in a non-virtualized environment. This credit allocation mechanism results in performance degradation in virtualized environment mainly because more VCPUs in Dom0 lead to less shared credits for each VCPU. I/O VCPU cannot be allocated with sufficient computing resource to satisfy packet processing. We propose to dynamically and temporarily steal some credits from other idling VCPUs to favor I/O VCPUs during each time slice while I/O VCPUs are busy with processing packets. The principle to steal credits is formalized in the following equation:

$$Steal = Credit(Idle_VCPUs)/(2*Num(IO_VCPUs))$$

where $Steal$ means the stolen credit for each I/O VCPU, $Credit(Idle_VCPUs)$ is for the credit of all idling VCPUS. $Num(IO_VCPUs)$ represents the number of I/O VCPUs. It shows that each idling VCPU's credit is dynamically cut in half to favor I/O VCPUs to eliminate their burden while working with intensive NIC interrupt requests. Since our policy steals credits from idling VCPUs, it does not hurt the overall system performance.

4.4 Data Movement Engine

Although the above VMM scheduler optimizations improve packet movement, they are unable to eliminate all cache misses on packets along the processing path. In virtualized environment, Direct Cache Access (DCA) [11, 18, 19] injects packets into the first physical core where NIC interrupts are delivered and cannot avoid cache misses on packets. In this subsection, we propose a small hardware data movement engine to consider VMM scheduling information to accurately inject incoming packets into

right cores where corresponding domains are running. The overview of architecture is illustrated in Figure 7.

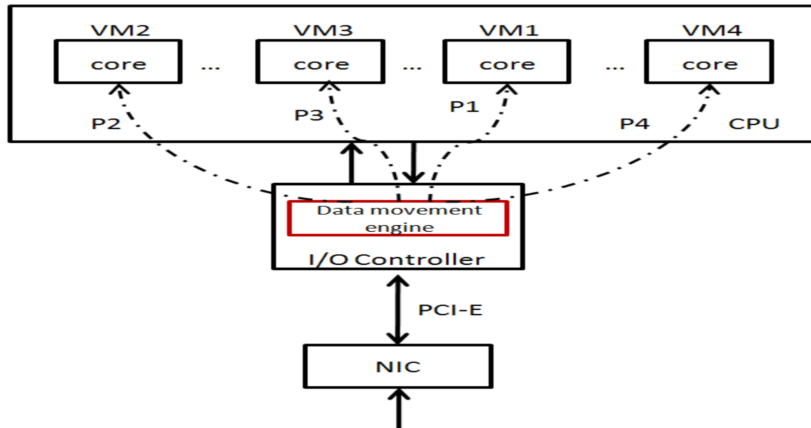

Figure 7. New architecture overview

In the new architecture, we add one small hardware unit (denoted as data movement engine) into I/O controller. When NIC receives a packet, it reads DMA descriptors to know DMA buffer address and then leverages DMA transactions over PCI-E interconnect to send the packet to I/O controller. The I/O controller passes the received packet into our new data movement engine. The data movement engine maintains VM-to-Core mapping information which is periodically updated by VMM scheduler. Thus, the engine can find out the destination core where the corresponding domain is running and directly inject packets into corresponding caches. For instance, as shown in Fig. 7, all packets belonging to VM1 will be delivered to the third core where VM1 is scheduled by VMM scheduler to be running. The detailed architectural designs of our data movement engine are illustrated in Figure 8.

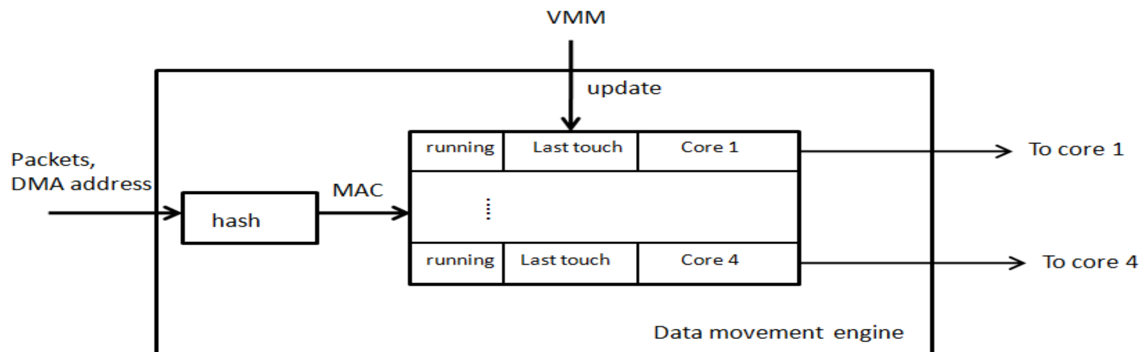

Figure 8. Date movement engine

Inside the data movement engine, we use a mapping table to maintain VM-to-Core mapping information. When VMM scheduler finishes scheduling VMs across multiple cores, it updates the mapping table. Each row in the mapping table represents one VM. The first running field indicates whether the corresponding VM is running or not. Last touch means who is the last to own the row, VMM or NIC. The last field is the destination core. When the data movement engine receives a packet, it extracts the packet's MAC address and hashes into mapping table. Data movement engine checks whether the corresponding VM is running or not. If yes, it obtains the destination core and then injects packets into corresponding caches. If not, data movement engine injects packet into a random core and then marks the last touch field as NIC. When VMM receives interrupts from NIC and schedules VM across cores, it checks with this mapping table to see whether the last touch field in the corresponding row has been set by NIC. If yes, it obtains the core information and schedules VM on the core. Otherwise, the default scheduling policy is applied. By leveraging VMM scheduling information, the new architecture is able to directly inject packets into correct cores and avoids cache misses on packets.

4.5 Simplified Bridge

As shown in Section 3, packet switching function requires only 600 cycles, and Jhash algorithm used for multiplexing packets by hashing MAC addresses only consumes 120 cycles. It motivates us to design a simplified bridge tailored for packet switching in virtualized environment. However, it must retain the same user/kernel interface as original bridge so that the user space bridge utility still works in virtualization environment. Since bridge utilities in user space are being used by domain management tool residing in Dom0 to create/destroy backend drivers, the new bridge should comply with the original user/kernel interface to avoid interference with the current workable system. The new design is required to keep bridge as simple as possible with respect to packet switching's performance and scalability.

Packet processing path of both Linux Bridge and our tailored bridge are shown in Figure 9. It shows that we bypass most of the functions introduced by Netfilter interface and re-implement the internal interfaces to minimize extra function costs except the bridge (*Xen_br_forward*). The Jhash algorithm is still adopted in our design. Our prototype is implemented as a new feature of Linux Bridge to take advantage of its existence in mainstream kernel.

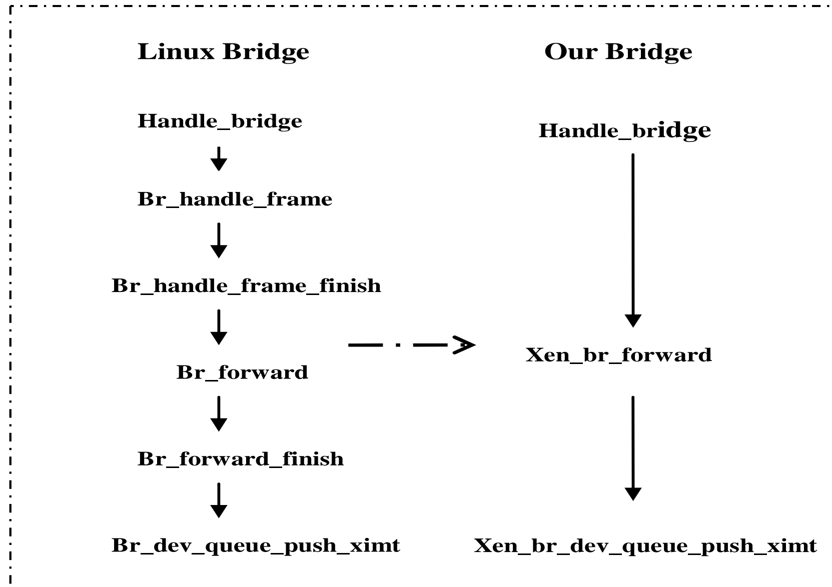

Figure 9. Linux Bridge vs. our bridge

5. Performance Evaluation

We implement our two VMM scheduler optimizations and the simplified bridge in Xen 3.1. Iperf is run over our Intel servers with our optimized Xen to understand performance impacts of our optimizations on network processing. We then study how much benefit web servers achieve by running the SPECWeb benchmark. Since no existing simulators support virtualization, we choose a full system simulator Simics and develop an experiment methodology to mimic virtualization environment. We enhance Simics with detailed cache, I/O timing models and modeling of the effects of network DMA. In order to mimic the virtualization overhead, we inject extra per-packet virtualization overheads from our profiling on real machines in the simulator. We extend the Digital Equipment Corporation 21140A Ethernet device with the support of interrupt coalescing using Device Modeling language DML to simulate a 10GbE Ethernet NIC. The device itself is connected to a lossless, full-duplex link of configurable bandwidth. The latency of a packet traversing the link is simply fixed to 1 *us*. We simulate two systems (client and server) running Linux 2.6.16 and interconnect them with 10GbE. The parameters we use in modeling the configuration are listed in Table 4. We are more interested in the relative behavior of these systems than their absolute performance, so some of these parameters are approximations.

Table 4. System configurations

Processor	four cores, 3GHz, in-order, two-issue
ICache/DCache	Private per core, 32 KB 2-way, 3-cycle hit latency
L2 Cache	Private per core, 2M, 8-way split, 14

	cycles hit latency
Memory	400 cycles
I/O register	1600 cycles
prefetch	Stream prefetch, degree: 4
Interrupt coalescing rate	64 packets per interrupt

5.1 System Optimizations on Xeon Servers

This subsection first studies performance benefits of all our three system optimizations (cache-aware and credit-stealing scheduler optimizations and simplified bridge) in terms of network bandwidth and core utilization per gigabit. We obtained these results by modifying Xen and running it on the Intel Xeon server. The results are presented in Figure 10. "Default" represents the original system with credit scheduler without any optimization. In the figure, bars represent the bandwidth and lines stand for core utilization per gigabit. We observe from Fig. 10 that our cache-aware scheduler increases bandwidth by 19%, and also saves 11% in core utilization per gigabit. The credit stealing policy for favoring I/O VCPUs further improves the network performance by 14% and saves 6% in core utilization per gigabit. It is observed that all three optimizations can increase the network bandwidth by 96% to 4.5 Gbps, and also save 36% in core utilization per gigabit. In our experiment, we notice that the total core utilization consumed by Dom0 is reduced from 105% to 84% by using all the optimizations.

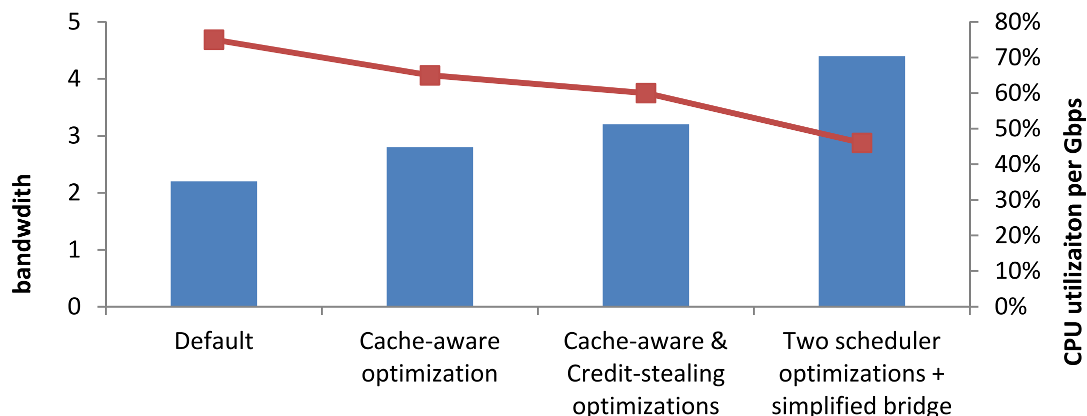

Figure 10. Network performance with system optimizations

Second, we study web server performance by running the web server benchmark SPECweb99 over 10GbE. The same configurations are used. Web server bandwidth with various configurations is illustrated in Figure 11. As shown in Fig.11, web server achieves 0.9Gbps, 1.2Gbps, 1.3Gbps and

1.5Gbps bandwidth without any optimization, with cache-aware scheduler, two VMM scheduler optimizations and all three optimizations, respectively. Reduced CPU utilization per gigabit in the figure points out the improved processing efficiency on web servers.

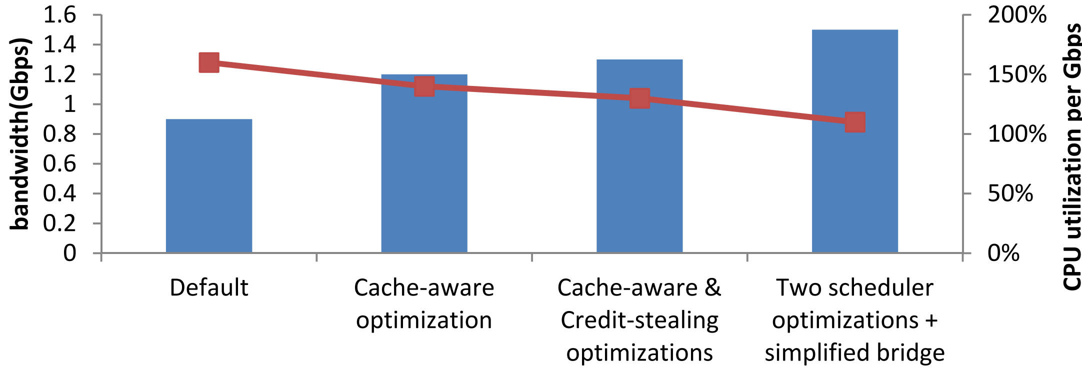

Figure 11. Web server performance with system optimizations

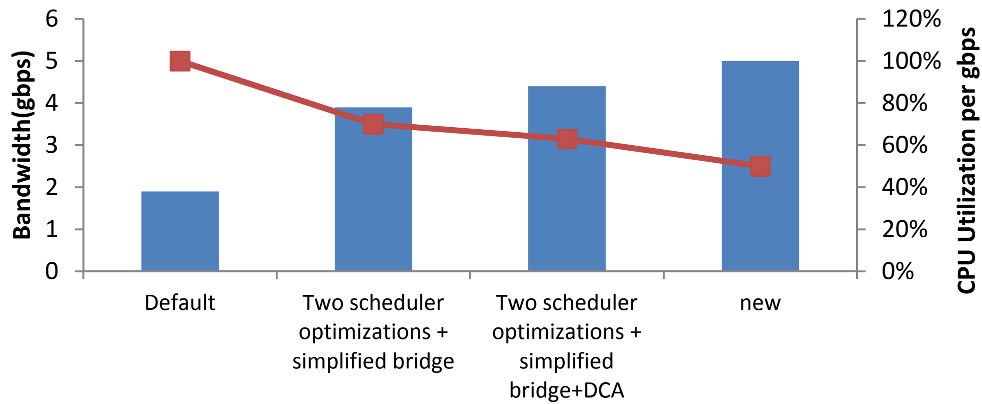

Figure 12 Network performance with architectural optimizations

5.2 Architectural Optimizations through Simulation

Besides three system optimizations, we also propose efficient architectural support to avoid cache misses along the packet movement. In this subsection, we first look at network performance in the receive side by running Iperf under various configurations: the default system without any optimization (default), all system optimizations, all system optimizations with DCA and the small data movement engine (new).

Figure 12 illustrates network bandwidth achieved by various configurations and corresponding CPU utilization. As shown in Fig.12, default can achieve only ~1.9 Gbps bandwidth by consuming ~100% CPU utilization per gigabit. By improving VMM scheduler and Linux Bridge, the network performance is improved by up to 3.9Gbps with 75% CPU utilization per gigabit. DCA is unaware of location of destination guest domain and injects packets into the first core, thus only achieving limited benefits. By

considering VMM scheduling information, the new architecture injects packets into right caches and continues improving network performance up to 5Gbps with 50% CPU utilization per gigabit.

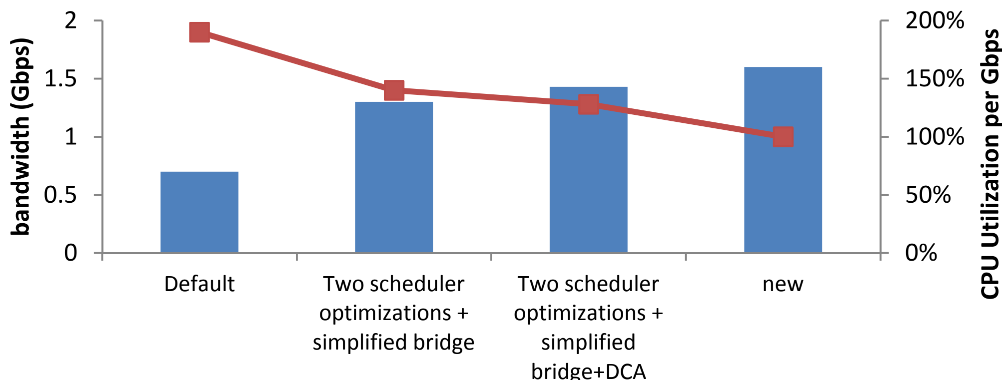

Figure 13. Web server performance with architectural optimizations

Similarly, we also investigate web server performance by running the web server benchmark SPECweb99 over 10GbE. The same configurations are used and results are illustrated in Figure 13. We find that the new architecture escalates web server performance by 120% compared to the default system while saving 90% CPU utilization per gigabit.

6. Related Work

Since the birth of VM, research in improving virtualized I/O performance never faded away. For system optimizations, Ongaro et al. [28] sorted the domains with the same states in the runqueue based on their remaining credits rather than arbitrarily insert the new domain at the end of each state section. However, they focused on the fairness of I/O performance with 1GE network and did not consider the VMM scheduler on mainstream multi-core systems where behaves significantly different from single core systems. With the same optimizations on our experiment environment under 10 GbE, we find that the blocking of scheduler tickle adversely glooms the I/O performance by a factor of 100 and the runqueue sort does not make any difference for I/O performance. In addition to VMM scheduler optimizations, lots of engineering optimizations have also been implemented to improve network I/O performance in virtualization environment. Menon et al. [25, 26, 27] analyzed virtualization performance overhead and then implemented numerous optimizations (e.g. reusing grant table, using large page size, moving data copy to guest etc) to bridge the gap between software and hardware techniques for I/O virtualization. Guo et al. [10] designed cache-aware scheduling for virtualization to improve web server performance. Liu et al. [21] adopted virtualization technology for HPC and allowed each domain to directly access the high

performance network. However, they targeted to the high performance network InfiniBand rather than Ethernet Network. In Ethernet Network, some researches including Crossbow [6] tried to address the performance issues by taking advantage of the new Ethernet NIC features like multiple TX/RX queues to allow domains to directly access the hardware. They heavily rely on hardware and hence sacrifice the features of portability and live migration, two major incentives for deploying virtualization in high end servers.

Numerous studies have been done in server architectures to efficiently tackle the network I/O virtualization challenge. In industry, Intel [16] offloads virtual switch (or packet de-multiplexing) from the driver domain to hardware NIC and deploys multiple queues to allow guest OS to directly access hardware queues. Recently, PCI-E standard group proposes single root IO virtualization (SR-IOV) [30] to self-virtualize a physical device into multiple lightweight PCI-E devices, significantly avoiding I/O virtualization overheads. However, they require extensive hardware platform support ranging from PCI-E, chipsets and PCI-E devices and also sacrifice virtual machine migration.

7. Conclusion

This paper analyzes the performance challenge of network virtualization over 10GbE network with a multi-core server. We found that virtualization under 10GE network adds significant performance overhead to network processing. We proposed two optimizations for the scheduler inside the VMM to improve the packet movement performance. In order to avoid cache misses along moving packets in virtualized environment, we proposed a small hardware data movement engine considering VMM scheduling information to accurately inject packets into cores where corresponding guest domains are running. We also redesigned a simplified bridge to switch packets to corresponding guest domains. Our combined optimizations are able to significantly reduce major bottlenecks in virtualization environment.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In 19th SOSP, Oct 2003.

[2] N. J. Boden, D. Cohen, R. E. Felderman et al., Myrinet: A Gigabit-per-Second Local Area Network. IEEE MICRO 1995

[3] L. Cherkasova, V. Kotov, T. Rokichi et al., Fiber Channel Fabrics: Evaluation and Design, 29th HICSS'96.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Wareld. Live Migration of Virtual Machines, OSDI.

[5] Credit scheduler, http://xen.org/files/summit_3/sched.pdf.

[6] Crossbow, http://opensolaris.org/os/project/crossbow/.

[7] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, Cambridge University, Aug 2004.

[8] S. GadelRab. 10-Gigabit Ethernet Connectivity for Computer Servers Volume 27, Issue 3, IEEE Micro, 2007.

[9] L. Grossman, Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In Ottawa Linux Symposium, Ottawa 2005.

[10] D. Guo, G. Liao, L. N. Bhuyan, Performance Characterization and Cache-Aware Core Scheduling in a Virtualized Multi-Core Server under 10GbE, International Symposium on Workload Characterization, Austin, TX, 2009.

[11] R. Huggahalli, R. Iyer, S. Tetrick, Direct Cache Access for High Bandwidth Network I/O. ISCA, 2005.

[12] Intel 82597 http://ark.intel.com/ProductCollection.aspx?series=32612.

[13] Infiniband Trade Association. http://www.infinibandta.org.

[14] Iperf, http://sourceforge.net/projects/iperf/.

[15] Inside Intel Core Micro-architecture: Setting New Standards for Energy-Efficient Performance, http://www.intel.com/technology/architecture-silicon/core.

[16] Intel Virtualization Technology Specification for the IA-32 Intel Architecture, April 2005.

[17] Jhash. http://www.burtleburtle.net/bob/hash/doobs.html.

[18] A. Kumar, R. Huggahalli, Impact of Cache Coherence Protocols on the Processing of Network Traffic. MICRO, 2007.

[19] A. Kumar, R. Huggahalli, S. Makineni, Characterization of Direct Cache Access on Multi-core Systems and 10GbE. HPCA, 2009.

[20] G. Liao, X. Zhu, L. N. Bhuyan, A New Server I/O Architecture for High Speed Networks, in 17th High Performance Computer Architecture, 2011.

[21] J. Liu, W. Huang, B. Abali et al., High Performance VMM-Bypass I/O in Virtual Machines, USENIX Annual Technical Conference, June 2006.

[22] I. M. Leslie, D. Mcauley, R. Black et al., The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. IEEE Journal of Selected Areas in Communication, 1996.

[23] Linux Bridge. http://bridge.sourceforge.net/.

[24] P. S. Magnusson, M. Christensson, J. Eskilson et al., Simics: A Full System Simulation Platform. IEEE Computer, February 2002.

[25] A.Menon, J. R. Santos, Y. Turner et al., Xenoprof - Performance profiling in Xen.

[26] A. Menon, J. R. Santos, Y. Turner et al., Diagnosing Performance overheads in the Xen Virtual Machine Environment, VEE, 2005.

[27] A. Menon, A. Cox, W. Zwaenepoel, Optimizing Network Virtualization in Xen, USENIX Annual Technical Conference, 2006.

[28] D, Ongaro., A. L, Cox., S, Rixne. 2008. Scheduling I/O in virtual machine monitors. VEE, 2008.

[29] F. Petrini, W. Feng, A. Hoisie et al., The Quadrics Network (QsNet): High-Performance Clustering Technology. HotI '01.

[30] PCI-E Specification, http://www.pcisig.com/specifications/pciexpress/base2/.

[31] M. Rosemblum, T. Garfinkel. Virtual Machine Monitors: Current Technology and Future trends. IEEE computer, 38(5): 39-47, 2005.

[32] Scalable Networking: Eliminating the Receive Processing Bottleneck. Microsoft WinHEC April 2004.

[33] Standard Performance Evaluation Corporation. SPECweb benchmark. http://www.spec.org.

[34] L. Zhao, L. Bhuyan, R. Iyer et al., Hardware Support for Accelerating Data Movement in Server platform, IEEE Transactions On Computer, Vol 56, No. 6, 2007.