Virtualization Technology for TCP/IP Offload Engine

En-Hao Chang, Chen-Chieh Wang, Chien-Te Liu, Kuan-Chung Chen, *Student Member, IEEE,* and Chung-Ho Chen, *Member, IEEE*

Abstract—Network I/O virtualization plays an important role in cloud computing. This paper addresses the system-wide virtualization issues of TCP/IP Offload Engine (TOE) and presents the architectural designs. We identify three critical factors that affect the performance of a TOE: I/O virtualization architectures, quality of service (QoS), and virtual machine monitor (VMM) scheduler. In our device emulation based TOE, the VMM manages the socket connections in the TOE directly and thus can eliminate packet copy and demultiplexing overheads as appeared in the virtualization of a layer 2 network card. To further reduce hypervisor intervention, the direct I/O access architecture provides the per VM-based physical control interface that helps removing most of the VMM interventions. The direct I/O access architecture out-performs the device emulation architecture as large as 30%, or achieves 80% of the native 10 Gbit/s TOE system. To continue serving the TOE commands for a VM, no matter the VM is idle or switched out by the VMM, we decouple the TOE I/O command dispatcher from the VMM scheduler. We found that a VMM scheduler with preemptive I/O scheduling and a programmable I/O command dispatcher with deficit weighted round robin (DWRR) policy are able to ensure service fairness and at the same time maximize the TOE utilization.

Index Terms—Hypervisor, I/O Virtualization, TCP/IP Offload Engine, VMM scheduler.

1 Introduction

As cloud computing becomes widespread, there are many emerging issues which have been discussed and addressed, such as security, quality of service [1], data center networks [2], and virtualization [3], [4]. For cloud applications, virtualization is one of the key enabling technology, which possesses two features that make it ideal for cloud computing, i.e., service partitioning and isolation. With partitioning, virtualization is able to support many applications and operating systems to share the same physical device while isolation enables each guest virtual machine to be protected from system crashes or viruses in the other virtual machines.

Virtualization abstracts the physical infrastructure through a *virtual machine monitor* (VMM) or *hypervisor*, which is the software layer between virtual machines and the physical hardware. VMM enables multiple virtual machines or guest operating systems to share a single physical machine securely and fairly. VMM makes the virtual machine under the illusion that it has its own physical device.

Cloud computing provides computing resources as a service over a network, and therefore servers need to provide high network bandwidth to effectively support cloud applications. To this end, one approach is to offload the TCP/IP protocol stacks from the host processors to an accelerator unit called *TCP/IP Offload Engine* (TOE) [5]. Unfortunately, in a virtualization environment, the network performance may be burdened by the overheads of network I/O virtualization.

For instance, the major overheads of network I/O virtualization can be distilled down to four categories: (1) packet copy, (2) packet demultiplexing, (3) VMM intervention, and (4) interrupt virtualization. Specifically, the received packets must be copied to the driver domain for software packet demultiplexing, and then the driver domain copies the packets to the target guest domain. Also an I/O instruction from the guest is trapped by the VMM, and the VMM handles and emulates the instruction. Physical interrupts are also trapped by the VMM, and then the VMM will issue the corresponding virtual interrupts to the designated guest operating system. These virtualization overheads have caused the loss of 70% network throughput when compared to the native system [6].

Cloud computing service also has to satisfy users' various requirements, i.e., I/O-intensive domains or CPU-intensive domains. An I/O-intensive domain may not get enough CPU resource to achieve high network throughput or it may not be scheduled in time to do so [7]. As a result, the I/O-intensive application will suffer from poor network throughput because of the inadequate scheduling policy.

- *E.-H. Chang, C.-T. Liu, K.-C. Chen, and C.-H Chen are with the Institute of Computer and Communication Engineering, Department of Electrical Engineering, National Cheng Kung University, Tainan City 701, Taiwan. E-mail: {enhowenhow, ufoderek, edi751001}@gmail.com, chchen@mail.ncku.edu.tw.*
- *C.-C. Wang is with the Information and Communications Research Laboratories, Industrial Technology Research Institute (ITRI), Hsinchu 31040, Taiwan. E-mail: ccwang.jay@itri.org.tw*

Manuscript received 1 Sept. 2013; revised 29 Jan. 2014; accepted — 2014; published online —.
Recommended for acceptance by —.
For information on obtaining reprints of this article, please send e-mail to: tcc@computer.org, and reference IEEECS Log Number TCC-2014-xx-xxxx.
Digital Object Identifier no. 10.1109/TCC.2014.x.

This paper first identifies three critical factors that affect the performance of a TCP/IP Offload Engine (TOE) in a virtualization environment: (1) I/O virtualization architectures, (2) quality of service (QoS) of network bandwidth, and (3) VMM scheduler. We address the system-wide architecture issues of TOE virtualization with respect to the above three factors and present the architectural designs as a whole.

The major contributions of this paper are as follows:

- We develop an electronic system level (ESL) network platform for the full system evaluation of TOE virtualization, which includes a behavioral TOE and an on-line TCP/IP verification system.
- We develop two TOE virtualization architectures, including device emulation and direct I/O access. In the device emulation architecture, our approach has eliminated the major overheads of network I/O virtualization such as packet copy and packet demultiplexing. In the direct I/O access architecture, we develop a multi-channel TOE to offload the management of virtual device in device emulation. This design further eliminates most of the VMM intervention overheads occurred in the device emulation and achieves higher network throughput.
- To guarantee the quality of TOE service while at the same time maximize the TOE utilization, we propose a decoupled design of the VMM scheduler and the TOE command dispatcher. Our approach allows the TOE commands from a VM to be issued according to the chosen I/O dispatching policy, not affected by the VMM scheduler.

The remainder of this paper proceeds as follows. Section 2 presents the background of the virtualization and TCP/IP Offload Engine. Section 3 describes the TOE virtualization using device emulation architecture and direct I/O access architecture, respectively. Section 4 presents the full system virtualization platform. Section 5 gives the performance evaluation results. Section 6 reviews the related works. We conclude the paper in Section 7.

2 Background and Preliminary

In this section, we first introduce the network I/O virtualization and then the TCP/IP Offload Engine.

2.1 Network I/O Virtualization

I/O virtualization architecture can be classified into three types, including full virtualization, para-virtualization, and software emulation. CASL Hypervisor [8] and VMware ESX server [9] are implemented in full virtualization architecture. VMware ESX server is a standalone hypervisor and it controls the I/O devices as well as emulates multiple virtual devices for a virtual machine. When an application in the virtual machine raises an I/O request, a virtual device driver in the virtual machine first handles the request and translates it into a privileged IN or OUT instruction. Once the privileged instruction is executed on the virtual machine, the VMM will trap the instruction, and the emulation function in the VMM will handle the instruction and direct the physical device to complete the request. This approach is efficient and transparent; however, the VMM needs to control the physical hardware by itself without the aid of legacy device driver. This is more complicated to implement since one needs to know the control interface of the device in detail, and if the device is updated, the VMM requires modification.

In VMware ESX server, when a packet is received, the NIC first transmits the packet into the buffer of the VMKernel, and then issues an interrupt to notify the VMKernel that the packet is in the buffer. A vSwitch in the VMKernel demultiplexes the packet and copies the packet to the buffer of designated virtual machine. Finally, the VMKernel notifies the VMM to raise a virtual interrupt to the designated virtual machine. The virtual machine receives the packet as if it were from the physical directly.

2.2 TCP/IP Offload Engine (TOE)

With the increasing demand for high network bandwidth, high-end servers have become burdened with the processing of the TCP/IP protocol stacks. It has been reported that a single host CPU cannot perform TCP/IP processing satisfactorily as the network bandwidth is greater than 10 Gbit/s because one CPU clock is required for 1bps of TCP/IP work [10]. Therefore, TCP/IP Offload Engine (TOE) has been proposed to offload the TCP/IP protocol stacks from the host CPU to the network processing unit for high network performance and alleviate the CPU loading [5], [11], [12], [13], [14]. The specialized hardware can accelerate the protocol processing to enhance the network performance.

In this work, we use the *electronic system level* (ESL) approach [15] to build an approximate-timed model of the TOE. This ESL methodology enables us to verify the TOE driver, the socket API, and the TOE functional hardware at the same time. Fig. 1 shows the system architecture of the TOE and the host system. The TOE consists of a control channel, a TOE engine, and a network interface while the host consists of the TOE driver and the socket API. The socket API provides the communication between the user application and the TOE driver. The driver controls the I/O operation of the TOE and handles the interrupt event from the physical hardware. The host communicates with the TOE engine through the control channel and transmits/receives packets via the network interface.

The Linux kernel currently does not support TOE hardware because of security and complexity issues [16]; therefore, existing applications should be modified and recompiled to use a TOE device. In this work, we develop our own TOE driver as well as the socket API similar to the BSD socket API used in Linux kernel. In

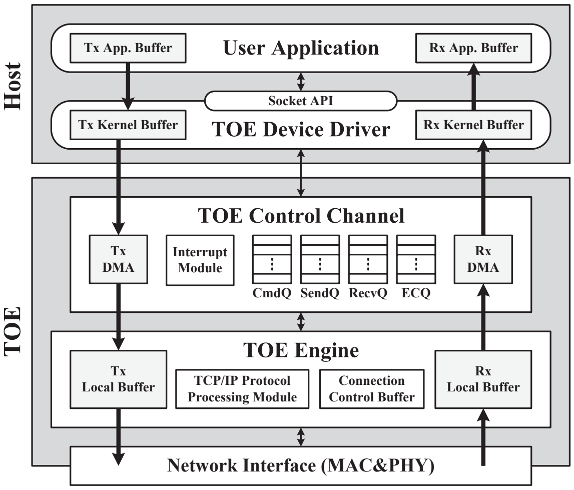


Fig. 1. System architecture of the TCP/IP Offload Engine.

the following, we illustrate how to design the TOE as well as its driver and socket API.

TOE Engine. The lower part of Fig. 1 shows the TOE engine including TCP/IP protocol processing module (PPM), connection control buffer (CCB), and local buffers. The PPM performs the TCP/IP protocol suite such as TCP, UDP, IP, ICMP, and ARP. An internal memory is needed for the inbound packet (Rx Local Buffer) and outbound packet (Tx Local Buffer) respectively, and the CCB is used to keep the information of the socket connection such as the Transaction Control Block (TCB) in TCP.

In the data transmission path, after the data payload is fetched from the host memory by the TxDMA, the PPM processes the data payload, and generates the packet header. Finally the MAC transmits the packet to the network. In the data reception path, the PPM receives the packet from the MAC for protocol processing. After handling the protocol, the PPM invokes the RxDMA to copy the data payload to the host memory. We have verified the above TOE operations in a real network platform with on-line packet transmissions.

TOE Control Channel. The middle part of Fig. 1 shows the control channel, i.e., the communication entity between the driver and the TOE engine similar to the work in [17]. The control channel consists of four queues, including a command queue, a send queue, a receive queue, and an event completion queue. These queues provide simple and abstracted communications between the kernel driver and the TOE engine. The driver organizes these queues in a series of ring buffers and the TOE manages the circular queues with the respective head and tail pointers.

The driver registers a new context through programmed I/O when the host driver wants to notify the TOE a new request. The TOE polls these circular queues, and after the request has been finished, the TOE updates its consumer pointer and adds finished information into the *event completion queue* (ECQ). Then, the TOE raises a physical interrupt to notify the driver that the request is finished.

The connection requests like listen(), accept(), close(), and connect() are registered in the *command queue* (CmdQ). Once the application wants to send data, the driver registers a context that includes the payload length, physical address, and some control flags through the *send queue* (SendQ). The contexts of the send queue point to the host buffers that will be transmitted by the TOE, and the intelligent DMA inside the TOE reads the contexts and moves the data from the host memory to the TOE's local buffer.

When an input packet comes, the TOE stores the packet into a fixed-length buffer in the host memory through DMA operations. When the buffer is full, the TOE raises an interrupt to notify the host. If the host needs the data immediately, the host can add a context to the *receive queue* (RecvQ), and the TOE will raise an interrupt as soon as the input packet is received regardless of whether the buffer is full or not.

We implement the one-copy mechanism in which the data payload needs to be copied to the kernel buffer and then to the user buffer. In this way, there is one extra copy compared to the zero-copy approach where the TOE can directly move the packet to the user buffer [5]. However, in this work, we focus on the TOE virtualization, and we implement the conventional one-copy approach.

TOE Driver and Socket API. The socket API manages the connection data structure and enables multiple processes or multiple threads to use the TOE, whereas the TOE driver is responsible for handling the interrupt from the TOE device and passes the commands from the socket API to the command buffers in the TOE.

In the transmission path, whenever the data buffer or the command buffer is full, the application process waits for the completion of the command. When the command is completed, the TOE registers a context which contains the information of the completed request in the event completion queue (ECQ) and then the TOE issues a physical interrupt to notify the host. The ECQ holds the context which includes, for instance, socket identifier, status flags, packet length, and packet pointer.

Once the driver receives an interrupt, a kernel thread in the driver is waked up by the interrupt service routine (ISR), and then the kernel thread gets the context data from the ECQ and stores into the control data structure of the destined socket. The kernel thread also determines which process to wake up based on the socket ID.

3 TOE Virtualization

Virtualized environment is usually used in high-end servers where many virtual machines provide services such as cloud computing through high bandwidth network. However, recent computer systems have suffered from the overheads of processing TCP/IP on very high speed network. A TCP/IP Offload Engine (TOE) can be used to improve the network performance as well as to alleviate the CPU loading. In this way, many of the CPU cycles used for TCP/IP processing are freed up and may be used for high-end servers to perform other tasks such as virtualization applications.

In TOE virtualization, all of the socket connections of the physical TOE are shared by all guest operating systems, and managed by the VMM. Thus, when the TOE receives a packet for a designated socket connection, the VMM knows which guest operating system the packet belongs to, and the TOE can send the data payload to the guest operating system using DMA. In this way, there is no need to demultiplex the inbound packets in the VMM and therefore the overheads of packet copy and packet demultiplexing in the VMM can be eliminated. In contrast, for the Layer-2 NIC virtualization, the inbound packet needs to be moved to the VMM for software demultiplexing [6].

In this paper, we present TOE virtualization using two architectures including device emulation and direct I/O access. We illustrate them in detail in the following.

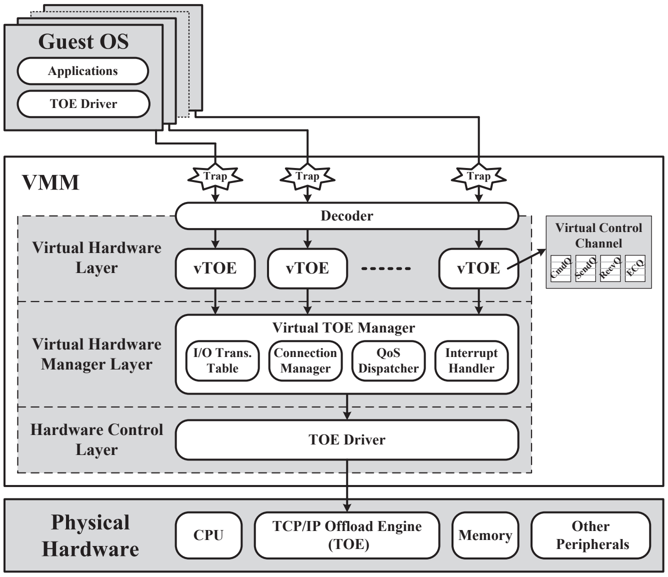


Fig. 2. System architecture of the TOE virtualization using device emulation.

3.1 Device Emulation Architecture

In order to virtualize an I/O device, the VMM must be able to intercept all I/O operations which are issued by the guest operating systems. The I/O instructions are trapped by the VMM and emulated in the VMM by software that understands the semantics of a specific I/O device.

Fig. 2 shows the system architecture of TOE virtualization based on device emulation. The VMM creates multiple virtual TOEs, and each virtual TOE is assigned to a guest virtual machine. The VMM manages the communication between the physical TOE and all virtual TOEs. The VMM is responsible for managing all guest operating systems, and provides all guests an efficient and transparent interface to the physical hardware.

There are three layers in the VMM to handle I/O operations, including a virtual hardware layer, a virtual hardware manager layer, and a hardware control layer. The virtual hardware layer emulates the virtual devices to the guest operating systems while the virtual hardware manager layer manages all I/O requests from the virtual devices, and the hardware control layer provides the device driver to control the physical device.

The virtual TOE manager is a critical component in our work since it enables the physical hardware resource to be shared by all the virtual machines and guarantees secure operations and service fairness. We illustrate the major modules of the virtual TOE manager in the following.

I/O Translation Table. In our work, a guest operating system translates the virtual address (VA) to the intermediate physical address (IPA) through the stage 1 page table. The guest operating system passes the IPA as a context of the command queue to the physical TOE. The VMM traps the command and translates the IPA to the physical address (PA) using the I/O translation table and then the PA is used to address the physical TOE. The DMA of the physical TOE can therefore copy the data payload from/to the right physical address according to the command.

Connection Manager. The TOE is a connection-based network device which supports multiple connections and can be shared by all guest operating systems. Thus, the virtual TOE manager needs to map the real physical socket connections in the physical TOE to the virtual connections of all of the guest operating systems.

QoS-based Programmable I/O Command Dispatcher. To share the physical TOE, a basic policy is that the VMM serves each virtual machine in a *first come first serve* (FCFS) manner with a credit-based scheduler. The VMM directly passes the commands from the current guest virtual machine to the physical TOE. Once the current virtual machine is switched out, the next guest can pass the command to the physical TOE. In this case, the credit-based scheduler will have difficulty to guarantee the quality of service (QoS) because the I/O dispatching operation and the CPU scheduler, i.e., the VMM scheduler, is tightly coupled.

In order to decouple I/O command dispatching and CPU scheduling, we add a virtual control channel to each virtual TOE, which includes a virtual command queue, send queue, receive queue, and event completion queue to buffer the commands from each guest operating system. As Fig. 3 shows, a command from a guest is trapped and moved into the designated virtual command queue. In this way, the command dispatcher can still dispatch the commands from the virtual command queues for the idle or the switched-out virtual machines.

In order to serve the pending commands fairly or

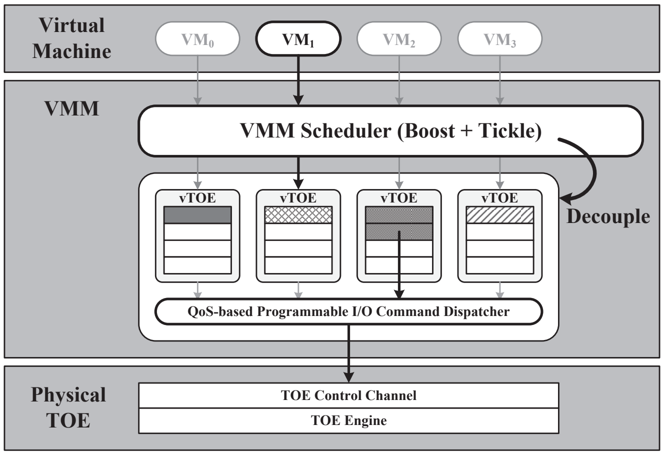


Fig. 3. Decoupled TOE command flow with QoS dispatcher.

according to the credit bought, we propose the use of a QoS-based programmable I/O command dispatcher (QoS dispatcher) which is decoupled from the VMM scheduler. In this paper, the QoS dispatcher provides quality of service for the bandwidth a virtual machine has been credited.

The QoS dispatcher can serve each virtual command queue in a round robin manner, for example. However, an ordinary round robin policy cannot guarantee the QoS in the case of different request message sizes as will be shown later in the experimental results. Thus, we employ the *deficit weighted round robin* (DWRR) algorithm [18] to remove this flaw. Using DWRR policy can handle packets of variable size without knowing their mean size. Fig. 4 shows the pseudocode of the DWRR policy in our work.

The QoS dispatcher serves each non-empty SENDQ$_n$ whose DeficitCounter$_n$ is greater than the message size and the remaining amount (DeficitCounter$_n$ - MessageSize) is updated in the DeficitCounter$_n$ which can be used in next round. Otherwise, the DeficitCounter$_n$ will add a Quantum$_n$ value and the QoS dispatcher will serve the next non-empty SENDQ. In order to support different requested bandwidth from a guest operating system, we can set different quantum value to each guest operating system. The guest having more quantum value has more credits to be debited, and it can get higher network bandwidth.

Interrupt Handler. In our work, a physical interrupt from TOE is first trapped by the VMM, and the VMM determines the target guest operating system of the physical interrupt. Then the VMM allocates the result context from the physical event completion queue (ECQ) to the corresponding virtual ECQ in the virtual TOE. After moving the result context to the virtual ECQ, the VMM raises a virtual interrupt to notify the target guest operating system that an event has occurred. The guest handles the interrupt as if the interrupt were from the physical device, and then the guest reads the context from its virtual ECQ, instead of the physical ECQ.

```
Initialization:
    FOR $n = 1$ to $N$    ($N$ is the number of virtual machines)
        $DeficitCounter_n \leftarrow 0$
    ENDFOR
    $RoundRobinPointer \leftarrow 0$
Enqueuing Operation:
    IF a send command comes from the n-th virtual machine  THEN
        Enqueue ( $SENDQ_n$ , SendCommand )
Dequeuing Operation:
    WHILE(1)
        $n \leftarrow RoundRobinPointer$
        IF  $SENDQ_n$ is NOT_EMPTY  THEN
            $MessageSize = Size( Head( SENDQ_n ) )$
            IF  $MessageSize \leqq DeficitCounter_n$ THEN
                Send ( Dequeue ( $SENDQ_n$ , SendCommand ) )
                $DeficitCounter_n \leftarrow DeficitCounter_n - MessageSize$
                BREAK
            ELSE
                $DeficitCounter_n \leftarrow DeficitCounter_n + Quantum_n$
                $RoundRobinPointer \leftarrow (n+1)$ mod $N$
            ENDIF
        ELSE
            $RoundRobinPointer \leftarrow (n+1)$ mod $N$
        ENDIF
    ENDWHILE
```

Fig. 4. The DWRR algorithm for QoS dispatcher.

3.2 Direct I/O Access Architecture

In the direct I/O access architecture, each guest operating system is connected to its own physical TOE control channel and the guest operating system can access the physical TOE directly.

Fig. 5 shows the comparison between the device emulation architecture and the direct I/O access architecture. In the device emulation architecture, each guest operating system is assigned a virtual TOE device where all control operations are intercepted and managed by the VMM while the direct I/O access provides multiple physical control channels in the physical TOE interface. Each physical control channel acts as if it is an independent control channel and therefore the device driver within the guest operating system can directly interact with its own physical control channel. Instead of assigning the ownership of the entire TOE to the VMM, the VMM treats each control channel as if it were a physical TOE and assigns the ownership of a control channel to a guest operating system.

With the direct I/O access, not only the communication overheads between the guest and the physical TOE are eliminated but it also avoids the frequent context switching between the guest operating systems and the

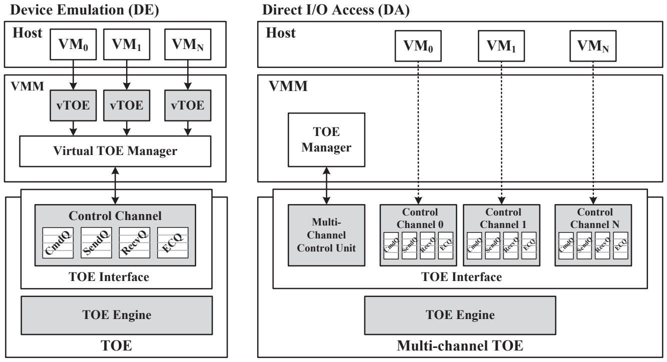


Fig. 5. The comparison of the device emulation and the direct I/O access.

VMM. However, the physical TOE now must provide multiple physical control channels and add control logic to multiplex these physical control channels, which increase the hardware complexity.

The TOE provides multiple physical control channels for the guest operating systems and also a control interface to the VMM. With the stage 2 page table in the MMU, the VMM assigns a control channel to a guest operating system by mapping the I/O locations of the designated control channel into the guest's address space. Therefore, a guest operating system can directly access its own control channel in the TOE interface without the VMM intervention. All control channels share the single execution resource. In addition, the direct I/O access provides an interrupt controller that enables all control channels assigned to each guest operating systems to share the same physical interrupt. In the following, we present the major components of the direct I/O access architecture.

Multiple Control Channel. The VMM assigns a physical control channel to a guest simply by mapping the I/O locations into the guest's address space. The VMM must ensure a VM will not access memory space which is allocated to other VMs during DMA operation.

To multiplex outbound network traffic, the TOE engine fairly serves all of the control channels in a round robin manner or based on the QoS policy. Note that the concept of QoS dispatcher in the device emulation architecture can also be implemented in the direct I/O access architecture. When a packet is received by the TOE, it is first demultiplexed, and then the TOE transfers the packet to the appropriate guest memory. After that, the TOE adds an event data into the event completion queue (ECQ) in the designated control channel.

In a multi-channel TOE, the number of control channels is limited by the hardware that the TOE provides. If the number of the guests exceeds the number of the control channels, some of the guest operating systems need to share the same channel. The guests sharing the same channel can use the device emulation approach presented before.

Interrupt Delivery. In the direct I/O access architecture, all physical control channels on the TOE must be able to interrupt their respective guest operating systems. Whenever the TOE fills up the receive buffer or sends the data payload from a guest operating system, the TOE enqueues the result information into the ECQ in the designated control channel, and raises a physical interrupt to notify the guest operating system that a new event has occurred.

In the direct I/O access architecture, all control channels share the same physical interrupt. The VMM needs to determine which guest operating system the interrupt is for. The TOE keeps tracking with the ECQs in all of the TOE channels. If a new event has occurred since the last physical interrupt, the TOE records the interrupt information in an interrupt bit vector in the multi-channel control unit.

After the VMM receives a physical interrupt from the TOE, it first reads the interrupt bit vector in the multi-channel control unit, and the VMM then decodes the pending interrupt bit vector and raises the corresponding virtual interrupts for the target guest operating systems. When a guest operating system is activated, the guest operating system will receive the virtual interrupt as if it were sent from the physical TOE.

SystemMMU and Memory Protection. During DMA operation, an I/O device must use physical address, which is invisible to the guest operating system. The intermediate physical address can be translated to the machine address in the VMM in device emulation architecture. However, in the direct I/O access architecture, without the VMM intervention, the DMA is not able to read or write the device using correct physical address. Moreover, without the memory protection of the VMM, some untrusted guest operating systems can even access the memory allocated to other guests through DMA and crash the whole system.

In order to get the correct physical address and guarantee the memory protection in the direct I/O architecture, we integrate a SystemMMU [19] into our platform. Similar to the MMU of a processor, the SystemMMU translates the addresses requested by the physical device into valid physical address.

Unprivileged physical address is not accessible to the device because no mapping is configured in the SystemMMU, which guarantees the memory protection. The VMM is responsible for managing the SystemMMU, and the SystemMMU can translate the intermediate physical address to the physical address for the DMA engines in the TOE.

4 Full System Virtualization Platform

Our virtualization platform is a full system based on the ARMv7 processor system, including a hardware simulation platform, a CASL Hypervisor [8] for the VMM and a Network Virtual Platform (NetVP) [20], [21] for on-line TCP/IP verification. In the following, we illustrate how

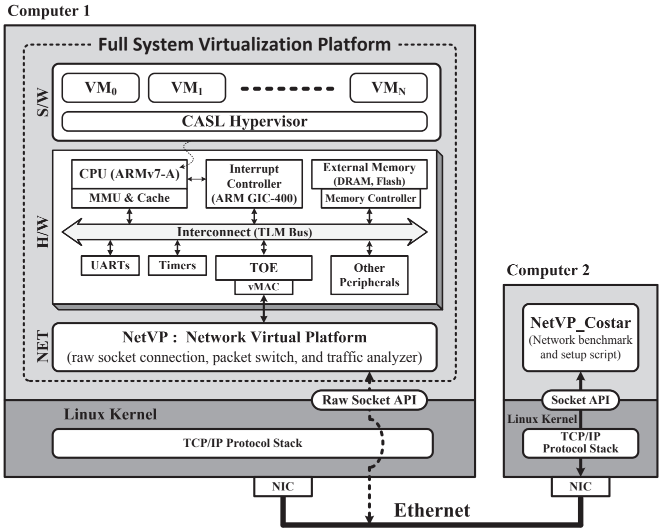


Fig. 6. System framework of the full system virtualization platform.

to virtualize the platform that enables multiple guest operating systems to share the same physical machine and communicate with real world computers in TCP/IP.

4.1 Hardware Simulation Platform

The simulation time is different between various abstract models of simulation accuracy [15]. In this paper, an approximate-timed simulation platform is developed using SystemC to help co-verifying software and hardware interwork while at the same time keeping reasonable simulation performance. In order to develop and verify the hypervisor, we implement a hardware simulation platform, as shown in the central part of Fig. 6. The hardware simulation platform consists of an instruction set simulator (ARMv7), an interrupt controller, memory modules, and other peripherals (timers and UARTs), all of which are necessary to successfully boot the Linux operating system. All of the system modules are connected via a transaction level modeling bus (TLM 2.0) released by OSCI (Open SystemC initiative).

An instruction set simulator (ISS) is a computer program which mimics the behavior of a target processor. The ISS implemented in this paper is based on the ARMv7 architecture with virtualization extensions and its correctness has been verified by successfully booting a Linux kernel.

In the ARMv7 architecture, the address translation, access permissions, attribute determination and checking are controlled by the *memory management unit* (MMU). With virtualization extensions, the MMU can support two stages of virtual address translation. If the MMU is configured to use only one stage, the output address is either the physical address (PA) or the intermediate physical address (IPA). On the other hand, if the MMU is configured to use two-stage address translation, the MMU also translates the intermediate physical address to the physical address (PA). Through the use of routed second stage data abort, the hypervisor can trap a specific guest operating system's I/O access and do the necessary emulation.

4.2 CASL Hypervisor Architecture

The CASL Hypervisor [8] is a virtual machine monitor designed for the ARM architecture; it can virtualize ARM Linux without any source-level modification. Based on the ARMv7 architecture with virtualization extensions, the CASL Hypervisor enables multiple guest operating systems to share the same physical platform, including the TOE design described previously.

CPU Virtualization. To support full virtualization, ARMv7 virtualization extension adds an additional CPU mode called *Hypervisor* mode. The CASL Hypervisor runs under the Hypervisor mode while the guest operating system runs under the *Supervisor* mode. Since the CASL Hypervisor uses full virtualization, only hypervisor traps, physical interrupts, and routed data aborts can reclaim the control from the guest operating systems. The CASL Hypervisor can trap privileged operations from the virtual machines, also the guest data abort can be routed to the hypervisor and all physical interrupts are handled directly by the hypervisor.

VMM Scheduler. The CASL Hypervisor schedules guest virtual machines based on Xen's credit scheduler [22]. The credit scheduler can fairly share the CPU resource. The baseline credit scheduler divides the virtual machines into two states: UNDER or OVER. If a virtual machine has credits remaining, it is in the UNDER state; otherwise, in the OVER state. Credits are debited on periodic scheduler interrupts that occur every 10 msec while a typical virtual machine switch interval is 30 msec [23]. When the sum of the credits for all virtual machines goes negative, all virtual machines are given new credit.

When making scheduling decisions, the baseline credit scheduler only considers whether a virtual machine is in UNDER or OVER state. Guests in the same state are simply serviced in a first-in, first-out manner. A new or switched guest is inserted into the tail of a run queue of the same state. The scheduler selects the guest requiring the CPU from the head of the run queue in UNDER state first. If there are no CPU requests in UNDER state, the scheduler serves the guests in OVER state in the same way.

The baseline credit scheduler can fairly share the processor resources; however, it does not consider the I/O performance. As an example, a CPU-intensive domain can solely consume 30 msec before it is switched out. On the other hand, an I/O-intensive guest may consume far less than 30 msec of CPU time waiting for I/O responses.

In Fig. 7, four VMs share a 1 Gbit/s TOE using device emulation for virtualization. A VM runs either an I/O domain application or a CPU-intensive application. The baseline credit scheduler is used. As Fig. 7 shows, the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2306425, IEEE Transactions on Cloud Computing

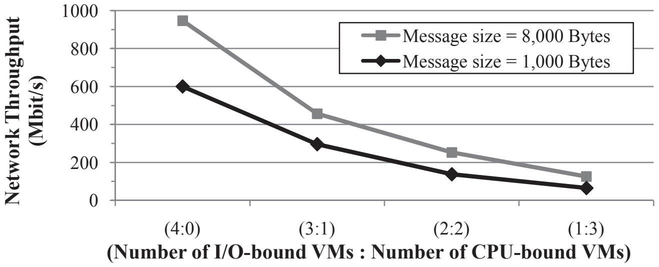


Fig. 7. Total network I/O performance with different number of I/O-intensive VMs and CPU-intensive VMs (1 Gbit/s TOE).

network throughput gets lower when the number of CPU-intensive applications is increased. For the TOE, this in turn can significantly reduce the TOE utilization. The detailed simulation system is illustrated in a later section.

In order to enhance the network throughput, the credit scheduler adds an additional state: BOOST which has higher priority than the UNDER and OVER state. A guest enters the BOOST state when it receives an event while it is idle. Moreover, the credit scheduler applies the *Tickle* mechanism which enables the VM in the BOOST state to preempt the current virtual machine and execute immediately. Therefore, the waiting time for an I/O domain can be lowered. The Boost and Tickle mechanism can improve the I/O throughput as will be shown later in the experimental results.

Memory Virtualization. In a system where each guest OS is running inside a virtual machine, the memory that is being allocated by the guest OS is not its true physical memory, but instead it is an intermediate physical memory. The VMM directly controls the allocation of the actual physical memory and therefore the guests can share the physical resources arbitrated by the VMM.

There are two approaches in handling the two-stage address translation (VA to IPA and IPA to PA) [19]. In systems where only one stage of memory address space translation is provided in hardware, for example using the MMU in the CPU; the VMM hypervisor must manage the relationship among VA, IPA, and PA. The VMM maintains its own translation tables (called shadow translation tables), which are derived by interpreting each guest OS translation table. However, this software address translation mechanism causes performance overheads. The alternative is to use hardware assistance for both stages of translation.

In our work, there are two page tables being used in the translation process under the virtualization environment. The fully virtualized guest operating system retains the control of its own stage 1 page table. This table translates a virtual address (VA) to an intermediate physical address (IPA). The intermediate physical address is then translated via the stage 2 page table managed by the hypervisor. Note that the guest operating systems cannot be aware of the existence of the stage 2 page table.

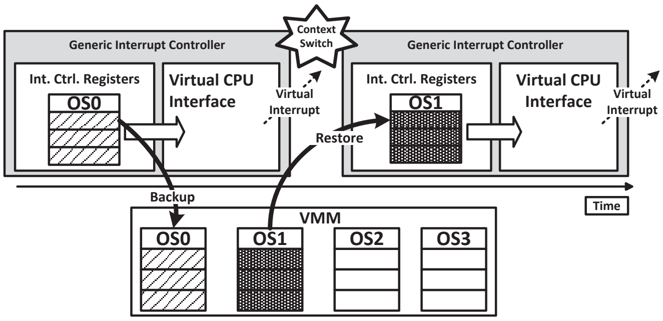


Fig. 8. Backup (OS0) and restore (OS1) interrupt control registers during virtual machine context switch.

The results of the second stage translation can be either a valid physical address or a translation fault. If the translation succeeds, the guest operating system's memory-mapped I/O is directed to the physical memory, or a physical device; otherwise, the hypervisor traps the translation fault and emulates a specific load/store instruction. Virtual devices can be implemented using routed second stage data abort, and pass-through devices can be redirected to its corresponding guest by the stage 2 page table.

Interrupt Virtualization. Virtual interrupt is an important mechanism for device virtualization by which the VMM needs to manage all physical interrupts and map a physical interrupt to a virtual interrupt for the guest operating system. The Generic Interrupt Controller (GIC) in ARM provides a hardware mechanism to support interrupt virtualization [24].

The Virtual CPU Interface inside the GIC can forward virtual interrupt requests to a target processor and trigger it to take virtual IRQ exceptions. A virtual machine receives its virtualized interrupts from this interface as if it were sent from the CPU Interface.

The GIC Virtual Interface Registers provide an extra programming interface to the hypervisor. The hypervisor can use this programming interface to forward interrupt requests to the GIC Virtual CPU Interface. As Fig. 8 shows, with the help of the interrupt control registers in the GIC, the VMM stores the interrupt information for each guest operating system. Whenever a context switch occurs, the VMM backups the control registers in the physical GIC and restores the control registers of the next-switched guest to the physical GIC control registers. The virtual CPU interface then raises the virtual interrupt based on the control registers in the physical GIC and the guest operating system therefore receives their interrupt.

4.3 Network Virtual Platform (NetVP)

Developing a complex network system such as a TOE is of a great challenge since it needs software/hardware co-development to verify the correctness of the design.

TABLE 1
Configurations of the Host Environment.

Toolchain	Version
Host OS	Suse 12.1
GCC (Native)	4.7.1
GCC (Cross Compiler)	4.4
SystemC	2.2.0
TLM	2.0.1
Boost Library	1.4.48

To this end, we employ the ESL design methodology that provides a faster simulation environment. Moreover, we integrate a Network Virtual Platform (NetVP) [20], [21] into our platform to provide on-line verification capability. In this way, our virtualization system can communicate with a real outside world computer system for functionality verification. See the lower part of Fig. 6 for the NetVP system.

The NetVP connects the vMAC in the simulation platform with an outside real network using semi-hosting method through the RAW Socket API of the Linux kernel and therefore a user can employ a packet analyzer, such as Wireshark [25], to trace the packet traffic of the network. In addition, we develop a multithreaded microbenchmark called NetVP_Costar, as shown in the right hand side of Fig. 6, which can support multiple simultaneous connections to communicate with the multiple guest operating systems in the simulation platform. The NetVP_Costar uses the standard TCP/IP protocol stacks in the Linux kernel to form a golden testbench to verify the correctness of the TOE. We employ the IP aliasing function in Linux to associate more than one IP address to a network interface.

5 Experimental Results

The system configuration of the host environment is shown in Table 1. Our full system virtualization platform has been implemented in SystemC modules that can be scaled and reconfigured easily and promptly. The approximately timed SystemC instruction set simulator model is fully compatible with the ARMv7 architecture and has been verified by booting the Linux OS. The detailed parameters of the target architecture for simulation are listed in Table 2. We configure the TOE bandwidth for 1 Gbit/s or 10 Gbit/s and evaluate the network throughput under different virtualization approach. A programmed-I/O based kernel system call is used to transfer the data payload between an application and the kernel buffer in virtual machines.

5.1 Single Virtual Machine Evaluation

Fig. 9 (a) compares the packet transmission performance of the two evaluated TOE virtualization architectures and the native system for 10 Gbit/s TOE. The achieved network throughput of the native system (without virtualization) is about 5 Gbit/s for the 8,000-byte message size. This performance limit is due to the programmed I/O operations performed by the 2 GHz CPU system.

TABLE 2
Configurations of the Virtualization Platform.

Platform Component	Configuration
Processor Model	ARMv7-A ISS
D-Cache, I-Cache	32 KB
Processor Frequency	2 GHz
Device Frequency	400 MHz
DRAM Capacity	1024 MB
DRAM Bandwidth	6.4 Gbit/s
Memory per Guest OS	128 MB
Virtual Machine Switch Interval	30 ms
Guest OS Context-Switch Interval	2.5 ms
Busybox version	1.19.4
Newlib version	1.19.0

The device emulation (DE) architecture has communication overheads resulted from the VMM intervention while the direct I/O access (DA) architecture can eliminate the communication overhead and improve the network performance. However, in the direct I/O access architecture, a physical interrupt is still intercepted by the VMM and therefore the performance of the direct I/O access architecture is slightly lower than the native non-virtualized system.

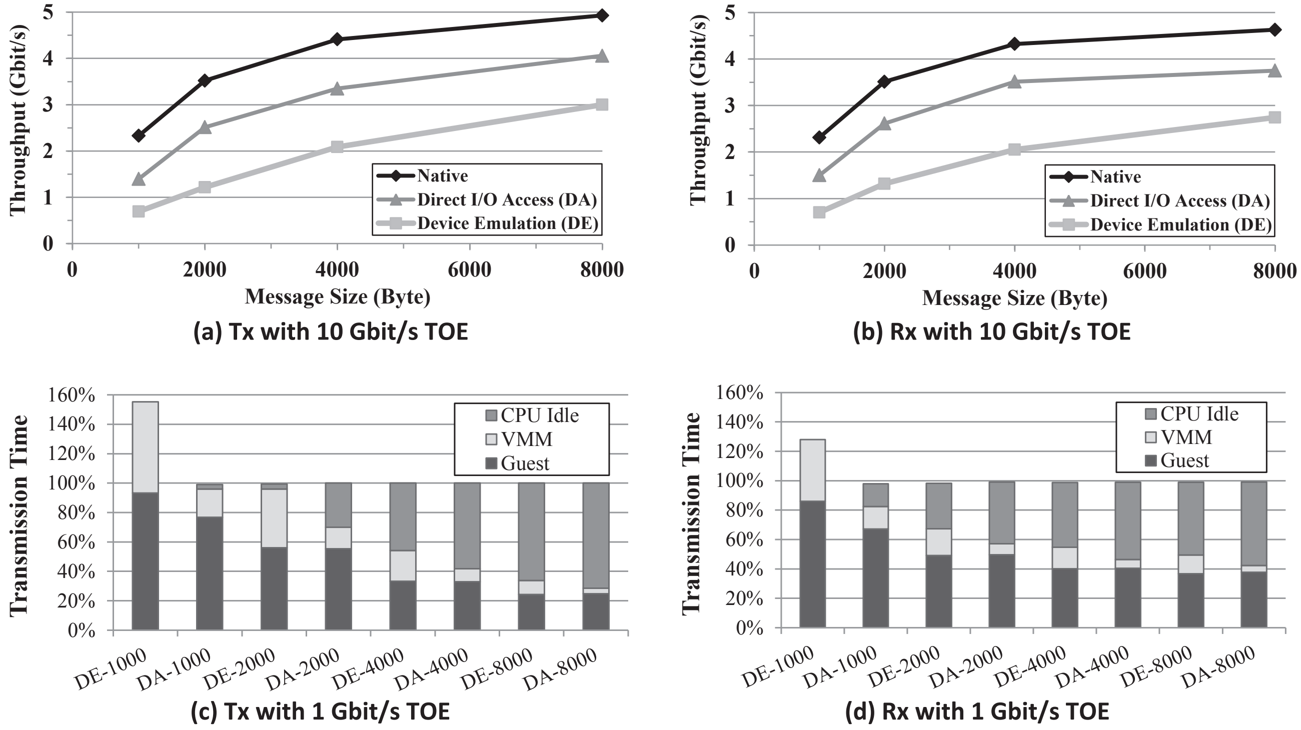
For the message size of 8,000 bytes, the transmission performance of the native system can achieve 4,991 Mbit/s, and the device emulation can only reach 3,022 Mbit/s, about 60% of the native system. In contrast, Xen's NIC virtualization has obtained about 30% of the native network throughput [6]. After removing most of the VMM interventions, the performance of direct I/O access is 32% higher than the device emulation architecture, or 80% of the native system.

Fig. 9 (b) shows the receiving performance where the network performance can achieve 4,633 Mbit/s with the native system while the device emulation architecture can only achieve 2,745 Mbit/s for the 8,000-byte message size. After eliminating most of the VMM interventions, the direct I/O access architecture can achieve 3,750 Mbit/s that is 36.6% higher than the device emulation architecture. The receiving performance is slightly lower than the transmission path. This is because the host is only notified by the TOE after the kernel receiving buffer is full.

Fig. 9 (c) compares the CPU utilization of single guest operating system between the device emulation and direct I/O access with 1 Gbit/s TOE. We measure the transmission time of sending 128 Mbytes of data in different message size. The transmission time of 1 Gbit/s TOE to send 128 Mbytes data is normalized to 100% of time as shown in the figure. The CPU idle time becomes larger as the message size is increased because the CPU can handle more data payload with the same number of system calls.

With the same message size, there are 5% to 23% more CPU idle time in the direct I/O access than in device

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2306425, IEEE Transactions on Cloud Computing



(a) Tx with 10 Gbit/s TOE

(b) Rx with 10 Gbit/s TOE

(c) Tx with 1 Gbit/s TOE

(d) Rx with 1 Gbit/s TOE

Fig. 9. Network throughput and CPU utilization with single virtual machine.

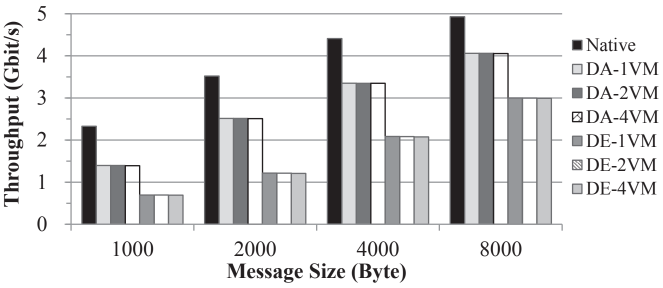


Fig. 10. The comparison of network throughput for 10 Gbit/s TOE.

emulation primarily due to the less time spent in the VMM. For the DE-1000 case, the system can not achieve 1 Gbit/s sending rate, so it shows a transmission time larger than 100%. Note that the DA-1000 case can achieve 1 Gbit/s.

Fig. 9 (d) compares the CPU utilization between the device emulation and direct I/O access during packet reception with 1 Gbit/s TOE. With the same message size, direct I/O access uses fewer time in hypervisor and gets 7% to 15% more CPU idle time than device emulation.

As indicated in Fig. 9 (a)(b), the achieved network throughput of a 10 Gbit/s TOE is limited to 5 Gbit/s by the performance of the CPU system used while the same CPU system is able to sustain 1 Gbit/s TOE except the DE-1000 case.

5.2 Scalability

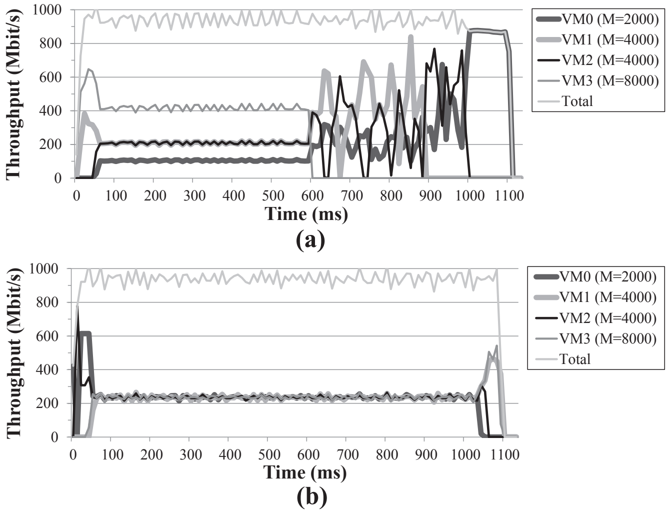
Fig. 10 shows the network throughput using 10 Gbit/s TOE. We evaluate the performance of device emulation and direct I/O access with different message size and the number of virtual machines. For either the device emulation or the direct I/O access, the total network throughput has shown no degradation as the number of guests is increased.

In our work, the TOE has no need to do hardware context switch because the connections in the TOE are shared by the virtual machines. Therefore, the context switch overhead of the VMM is low and it is too small to have significant influence on the network throughput.

5.3 Bandwidth Quality of Service

In the virtualization environment, the behavior of guest systems may be different from each other and can be classified into CPU-intensive domain and I/O-intensive domain. In this section, we evaluate the network I/O performance with different requirement of either CPU-intensive or I/O-intensive applications. For the evaluation, the CPU-intensive domains run infinite while loop to fully utilize a guest's processor resources while the I/O-intensive domains run the network application to fully utilize a guest's network resources.

In order to share the physical 1 Gbit/s TOE fairly in the virtualized environment, the VMM allocates the TOE resource to each guest operating system in the round robin manner as default. In the case that four I/O-intensive guest operating systems transmit the packets



(a)

(b)

Fig. 11. Network throughput with different dispatching policy. (a) Basic round robin. (b) DWRR.

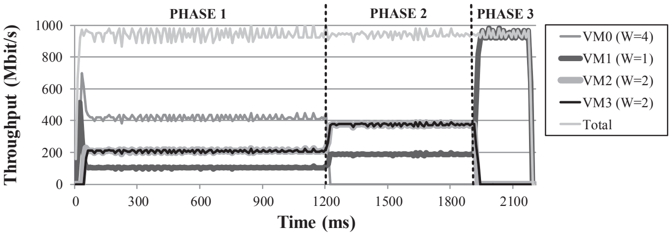


Fig. 12. Network throughput with different weighted values of different guests.

with the same message size at the same time, the TOE can be shared fairly. However, as shown in Fig. 11 (a), once the message size is different with the four virtual machines, the round robin based dispatcher cannot guarantee the fairness. To solve the problem, we use deficit weighted round robin (DWRR) based QoS dispatcher to ensure service fairness.

In Fig. 11 (b), four guest operating systems transmit the packets with the different message size simultaneously using the QoS dispatcher. Although the message size is different with the four virtual machines, the QoS dispatcher can still guarantee the bandwidth quality of service such that the guest operating systems can fairly share the TOE.

In cloud computing environment, different user may request different network bandwidth. The QoS dispatcher can support different requested network bandwidth from different guest virtual machines. Fig. 12 shows that the QoS dispatcher can guarantee the requested network bandwidth. Assume that the QoS dispatcher assigns the bandwidth requirement for the four VMs in the ratio of 4:1:2:2. In phase 1, the four guest operating systems use the TOE according to their requested network bandwidth ratio. In phase 2, VM0 does not use the TOE and in this case other guest virtual machines can share the TOE according to their proportion of weighted values (1:2:2) and the TOE can still operate to 1 Gbit/s. In phase 3, assuming only VM1 uses the TOE; in this case VM1 can own the whole TOE resource.

So far, we have discussed the cases that all virtual machines are I/O-intensive domains. However, in mixed domain applications, the total network throughput can be significantly reduced by the CPU-intensive guest. The solution for this lies in the VMM scheduler as will be examined in the following section.

5.4 CPU Scheduler with I/O Preemption

If the VMM scheduler has no knowledge of I/O requirement, this may result in poor I/O throughput as well as high latency. This is because the scheduler only guarantees the fairness in using the CPU resources rather than the I/O resources. When a CPU-intensive application has owned the CPU resource, I/O-intensive applications may not be scheduled in time to use the TOE; moreover, a CPU-intensive application can consume up to 30 msec of CPU time before VM switching. As illustrated in Fig. 7, the network throughput gets lower when the workload of CPU-intensive applications is increased, which in turn significantly reduces the TOE utilization.

In Fig. 13 (a), VM0 runs the network I/O job while the rest of VMs do CPU workload. The original credit scheduler gets very poor performance in network I/O throughput in this case. With only *Boost* mechanism, the network performance still cannot saturate the 1 Gbit/s TOE because the pending guest cannot preempt the current running guest to handle the I/O event immediately. The *Tickle* mechanism enables the pending guest to preempt the current guest if the priority of the pending guest is higher than the current running guest.

As Fig. 13 (b), (c), and (d) show, with the increase in the number of the I/O domains, the CPU resource can be shared more fairly with these three configurations. When the guest virtual machines all are of I/O-intensive domains, all of the scheduling policies can guarantee the fairness to share the CPU resource and also the full TOE utilization can be obtained.

6 Related Work

In this section, we review related works for network I/O virtualization, in terms of software-based solutions and hardware-based solutions.

Software-based Solutions. Xen's para-virtualization driver model uses a shared-memory-based data channel to reduce data movement overhead between the guest domain and driver domain [26]. Page remapping and batch packet transferring are proposed to improve the performance of network I/O virtualization [27]. XenLoop improves inter-VM communication performance also using shared memory [28]. Within these software-based solutions, the received packet still needs to be copied to the VMM for demultiplexing, which limits the

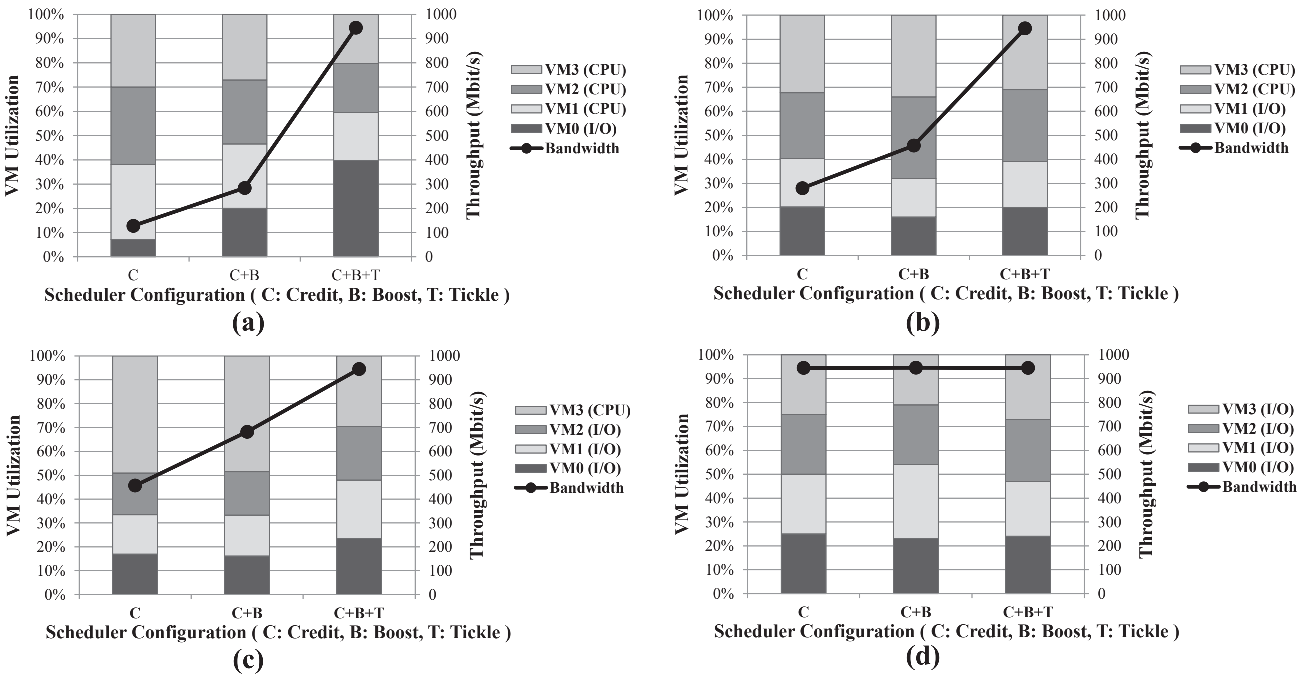


Fig. 13. The network throughput impact of the Boost and Tickle mechanism with various ratio of the I/O-intensive VM to the CPU-intensive VM. (a) 1:3 (b) 2:2 (c) 3:1 (d) 4:0.

performance of network I/O virtualization, and therefore there are many hardware-assisted solutions [6], [29], [30], [31], [32] which have been proposed to improve the performance of network I/O virtualization.

Hardware-based Solutions. Intel's Virtual Machine Device Queue (VMDq) offloads network I/O management burden from the VMM to the NIC, which enhances network performance in the virtual environment, freeing processor cycles for application work [29], [30]. It improves the performance of packet transactions from NIC towards the destined VM. As a packet arrives at the NIC, the dispatcher in NIC sorts and determines which VM the packet is destined based on the MAC address and the vLAN tag. The dispatcher then places the packet in the target buffer assigned to the VM. When a packet is transmitted from a virtual machine, the hypervisor places the transmitted packet in their respective queue located in the NIC.

To prevent network blocking and service each queue fairly, the NIC transmits the queued packets in round robin manner, to guarantee quality of service. In the similar concept, Ram et al. [31] utilized multi-queue network interface to eliminate the software overheads of packet demultiplexing and packet copying. However, in the above solutions, the VMM involvement is still required in packet processing for memory protection and address translation in Xen. For optimization, a grant-reuse mechanism between the driver domain and the guest domain has been proposed to mitigate the overhead of virtualization. The solutions above do not present how to virtualize the communication between the NIC and the virtual machines clearly. In our work, we illustrate how to virtualize the communication between the TOE interface and the virtual machines.

Willmann et al. [6] illustrate how to virtualize the communication between network interface and the VMM in detail, and they propose a concurrent direct network access (CDNA) architecture to eliminate the overheads of the communication, packet copy, and packet demultiplexing. When the NIC receives a packet, it uses the Ethernet MAC address of a VM to demultiplex the traffic, and transfers the packet to the appropriate guest memory using DMA descriptors from the context of target virtual machine.

In a non-virtualized environment, the NIC uses physical memory address to read or write the host system memory. The device driver in the host translates a virtual address to a physical address and notifies the DMA in the NIC with the correct physical address for moving packets. However, this direct I/O access architecture is dangerous in a virtualized machine since the VMM cannot find out whether there is a malicious driver in the virtual machine or not. If there is a buggy or malicious driver in the virtual machine, it could easily pollute the memory region of other virtual machines [6]. To prevent the malicious driver from illegal accesses, the VMM is required to stay in the path of enqueue operation for DMA memory protection, and this results in a protection overhead. In the direct I/O access architecture of our work, we can perform enqueue operation without the VMM involvement through an I/O MMU approach and ensure the memory protection.

Y. Dong et al. [32] propose a single-root I/O virtualization (SR-IOV) implemented in generic PCIe layers. SR-IOV inherits I/O MMU to offload memory protection and address translation, and to eliminate the overhead of the VMM intervention. SR-IOV can create multiple virtual functions (VFs) which can be assigned to virtual machines for direct I/O access while the physical device is shared by all the VMs. The main overhead in SR-IOV comes from handling the interrupts from a network device. This is because the interrupt is still intercepted and routed to the virtual machines by the VMM. As a result, they optimize the interrupt processing using mechanisms such as interrupt coalescing.

7 Conclusion

In this paper, we identify three critical factors to provide a robust network service in the virtualization environment: I/O virtualization architectures, quality of service (QoS), and VMM scheduler.

First, we develop two virtualization architectures including device emulation and direct I/O access to virtualize a TCP/IP Offload Engine. In the device emulation, the VMM intervention causes the communication overhead and limits the network performance. To this end, we employ the direct I/O access architecture to eliminate the VMM intervention overhead. The TOE provides the multiple control channels where each channel can be assigned to a guest operating system. Thus, the guest virtual machine can directly access the TOE and most of the VMM intervention overheads can be eliminated.

For quality of TOE service, in order to decouple the I/O command dispatcher and the CPU scheduler, we add virtual command queues into each virtual TOE to buffer I/O commands from the virtual machines. A command from a guest is trapped and decoded by the hypervisor and sent to the designated virtual command queue. In this way, the I/O command can be dispatched regardless the virtual machine domain is running or not. Moreover, we dispatch the commands across all virtual command queues using *deficit weighted round robin* (DWRR) algorithm rather than *first come first serve* (FCFS) algorithm to ensure the quality of service.

The VMM scheduler has a significant impact on I/O performance. In a traditional scheduler, an I/O-intensive domain may not get enough CPU resource or may not be scheduled in time. This causes poor I/O throughput. With the *Boost* and *Tickle* mechanisms, once an idle guest receives an I/O completion event, the guest VM enters the BOOST state and then it will preempt the current domain that is in the UNDER or OVER state. This approach favors an I/O-intensive guest machine and in turn improves the TOE utilization.

By decoupling the TOE command flow, our work shows that a VMM scheduler with preemptive I/O scheduling and a programmable I/O command dispatcher with DWRR policy are able to ensure service fairness and at the same time maximize the TOE utilization.

Acknowledgments

This work was supported in part by the National Science Council, Taiwan, under Grant NSC 102-2221-E-006-272.

References

[1] J.-W. Lin, C.-H. Chen, and J. M. Chang, "QoS-Aware Data Replication for Data-Intensive Applications in Cloud Computing Systems," *IEEE Trans. on Cloud Computing*, vol. 1, no. 1, pp. 101-115, 2013.

[2] K. Bilal, M. Manzano, S. U. Khan, E. Calle, K. Li, and A. Y. Zomaya, "On the Characterization of the Structural Robustness of Data Center Networks," *IEEE Trans. on Cloud Computing*, vol. 1, no. 1, pp. 64-77, 2013.

[3] M. Kesavan, I. Ahmad, O. Krieger, R. Soundararajan, A. Gavrilovska, and K. Schwan, "Practical Compute Capacity Management for Virtualized Data Centers," *IEEE Trans. on Cloud Computing*, vol. 1, no. 1, pp. 88-100, 2013,

[4] A. Amokrane, M. F. Zhani, R. Langar, R. Boutaba, and G. Pujolle, "Greenhead: Virtual Data Center Embedding across Distributed Infrastructures," *IEEE Trans. on Cloud Computing*, vol. 1, no. 1, pp. 36-49, 2013,

[5] H. Jang, S.-H. Chung, and D.-H. Yoo, "Implementation of an Efficient RDMA Mechanism Tightly Coupled with a TCP/IP Offload Engine," *Proc. IEEE Int'l Symp Industrial Embedded Systems (SIES)*, pp. 82-88, 2008.

[6] P. Willmann et al., "Concurrent Direct Network Access for Virtual Machine Monitors," *Proc. IEEE 13th Int'l Symp. High Performance Computer Architecture (HPCA)*, pp. 306-317, 2007.

[7] Z. Chang, J. Li, R. Ma, Z. Huang, and H. Guan, "Adjustable Credit Scheduling for High Performance Network Virtualization," *Proc. IEEE Int'l Conf. Cluster Computing (CLUSTER)*, pp. 337-345, 2012.

[8] C.-T. Liu, "CASL Hypervisor and its Virtualization Platform," Master Thesis, Institute of Computer and Comm. Eng., Nat'l Cheng Kung Univ., Taiwan, 2012.

[9] I. Ahmad, J. M. Anderson, A. M. Holler, R. Kambo, and V. Makhija, "An Analysis of Disk Performance in VMware ESX Server Virtual Machines," *Proc. IEEE Int'l Workshop on Workload Characterization (WWC)*, pp. 65-76, 2003.

[10] N. Bierbaum, "MPI and Embedded TCP/IP Gigabit Ethernet Cluster Computing," *Proc. IEEE Conf. Local Computer Network (LCN)*, pp. 733-734, 2002.

[11] D.-J. Kang, C.-Y. Kim, K.-H. Kim, and S.-I Jung, "Design and Implementation of Kernel S/W for TCP/IP Offload Engine (TOE)," *Proc. 7th Int'l Conf. Advanced Comm. Technology (ICACT)*, pp. 706-709, 2005.

[12] Z.-Z Wu and H.-C. Chen, "Design and Implementation of TCP/IP Offload Engine System over Gigabit Ethernet," *Proc. 15th Int'l Conf. Computer Comm. and Networks (ICCCN)*, pp. 245-250, 2006.

[13] Y. Ji and Q.-S. Hu, "40Gbps Multi-connection TCP/IP Offload Engine," *Proc. Int'l Conf. Wireless Comm. and Signal Processing (WCSP)*, pp. 1-5, 2011.

[14] H. Jang, S.-H. Chung, and S.-C. Oh, "Implementation of a Hybrid TCP/IP Offload Engine Prototype," *Proc. Asia-Pacific Computer Systems Architecture Conf. (ACSAC)*, pp. 464-477, 2005.

[15] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: a Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.

[16] Linux Foundation, http://www.linuxfoundation.org/

[17] D.-J. Kang, K.-H. Kim, S.-I. Jung, and H.-Y. Bae, "TCP/IP Offload Engine Module Supporting Binary Compatibility for Standard Socket Interfaces," *Grid and Cooperative Computing (GCC)*, pp. 357-369, Springer, 2005.

[18] M. Shreedhar and G. Varghese, "Efficient Fair Queuing using Deficit Round Robin," *ACM SIGCOMM Computer Comm. Review*, Vol. 25, No. 4, pp. 231-242, Oct. 1995.

[19] R. Mijat, A. Nightingale, "Virtualization is Coming to a Platform Near You," White Paper, ARM, 2011.

[20] C.-C. Wang and C.-H. Chen, "A System-Level Network Virtual Platform for IPsec Processor Development," *IEICE Trans. Information and Systems*, Vol. E96-D, No. 5, pp. 1095-1104, May 2013.

[21] C.-C. Wang, "A Dataflow-based Cryptographic Processing Unit for High-throughput IPsec Processors," PhD Dissertation, Institute of Computer and Comm. Eng., Nat'l Cheng Kung Univ., Taiwan, 2013.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2306425, IEEE Transactions on Cloud Computing

[22] D. Ongaro, A. L. Cox, and S. Rixner, ”Scheduling I/O in Virtual Machine Monitors,” *Proc. ACM SIGPLAN/SIGOPS Int'l Conf. Virtual Execution Environments (VEE)*, pp. 1-10, 2008.

[23] P. Barham et al., ”Xen and the Art of Virtualization,” *Proc. 19th ACM Symp. Operating Systems Principles (SOSP)*, pp. 164-177, 2003.

[24] ”ARM Generic Interrupt Controller Architecture Specification”, ARM, 2011.

[25] Wireshark, a open-source packet analyzer, http://www.wireshark.org/

[26] K. Fraser et al., ”Safe Hardware Access with the Xen Virtual Machine Monitor,” *Proc. 1st Workshop Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[27] A. Menon, A. L. Cox, and W. Zwaenepoel, ”Optimizing Network Virtualization in Xen,” *Proc. USENIX Annu. Tech. Conf.* , pp. 15-28, 2006.

[28] J. Wang, K.-L. Wright, and K. Gopalan, ”XenLoop: A Transparent High Performance Inter-VM Network Loopback,” *Proc. ACM 17th Int'l Symp. High Performance Distributed Computing (HPDC)*, pp. 109-118, 2008.

[29] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, ”Bridging the Gap between Software and Hardware Techniques for I/O Virtualization,” *Proc. USENIX08 Annu. Tech. Conf. (ATC)*, pp. 29-42, 2008.

[30] H. Raj and K. Schwan, ”High Performance and Scalable I/O Virtualization via Self-Virtualized Devices,” *Proc. ACM 16th Int'l Symp. High Performance Distributed Computing (HPDC)*, pp. 179-188, 2007.

[31] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner, ”Achieving 10 Gb/s Using Safe and Transparent Network Interface Virtualization,” *Proc. ACM SIGPLAN/SIGOPS Int'l Conf. Virtual Execution Environments (VEE)*, pp. 61-70, 2009.

[32] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, ”High Performance Network Virtualization with SR-IOV,” *Proc. IEEE 16th Int'l Symp. High Performance Computer Architecture (HPCA)*, pp. 1-10, 2010.

En-Hao Chang received the B.S. degree in engineering science and the M.S. degree in computer and communication engineering from the National Cheng-Kung University, Tainan, Taiwan, in 2011 and 2013, respectively. He is currently an R&D engineer at Mediatek Inc., Taiwan. His research interests include computer architecture, computer network, virtualization, and electronic system level (ESL) design.

Chen-Chieh Wang received the M.S. and Ph.D. degrees, both in computer and communication engineering from the National Cheng-Kung University, Tainan, Taiwan, in 2005 and 2013, respectively. From 2008 to 2011, he was an adjunct instructor with Department of Electrical Engineering, Feng-Chia University, Taichung, Taiwan. He is currently an R&D engineer at Information and Communications Research Laboratories (ICL), Industrial Technology Research Institute (ITRI), Hsinchu, Taiwan. His research interests include computer architecture, computer network, network security, heterogeneous system architecture (HSA), and electronic system level (ESL) design.

Chien-Te Liu received the B.S. degree in computer science and information engineering and the M.S. degree in computer and communication engineering from the National Cheng-Kung University, Tainan, Taiwan, in 2010 and 2012, respectively. He is currently an R&D engineer at Acer Inc., Taiwan. His research interests include computer architecture, virtualization, and electronic system level (ESL) design.

Kuan-Chung Chen received the B.S. degree in computer science and information engineering and the M.S. degree in computer and communication engineering from the National Cheng-Kung University, Tainan, Taiwan, in 2008 and 2010, respectively. He is currently pursuing the Ph.D. degree from the Institute of Computer and Communication Engineering, National Cheng-Kung University, Tainan, Taiwan. His research interests include computer architecture, virtualization, and electronic system level (ESL) design.

Chung-Ho Chen received the M.S.E.E. degree in electrical engineering from the University of Missouri-Rolla, Rolla, in 1989 and the Ph.D. degree in electrical engineering from the University of Washington, Seattle, in 1993. He then joined the Department of Electronic Engineering, National Yunlin University of Science and Technology. In 1999, he joined the Department of Electrical Engineering, National Cheng-Kung University, Tainan, Taiwan, where he is currently a Professor. His research areas include advanced computer architecture, graphics processing, and full system ESL simulation systems. Prof. Chen was a recipient of the 2009 outstanding teaching award and the 2013 teaching award from the National Cheng-Kung University. He was the Technical Program Chair of the 2002 VLSI Design/CAD Symposium held in Taiwan. He served the IEEE Circuits and Systems Society Tainan Chapter chair from 2011 to 2012 and received the 2013 IEEE Circuits and Systems Society Region 10 Chapter of the Year Award.