國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

藉由繞過驅動程式域改善Xen虛擬機器之網路吞吐量

Improving Xen Network Throughput by Bypassing Driver Domain

研究生：戴函昱

指導教授：張瑞川 教授

中 華 民 國 九 十 六 年 六 月

藉由繞過驅動程式域改善Xen虛擬機器之網路吞吐量

學生：戴函昱　　指導教授：張瑞川教授

國立交通大學資訊科學與工程研究所

論 文 摘 要

在原本的 Xen 網路架構中，一個客虛擬機器要必須要透過主虛擬機器才能夠存取網路。這個架構的好處是可以重用主虛擬機器中的作業系統的驅動程式，而不需要另外開發。但是這個架構也造成了額外的負擔，因為主虛擬機器和客虛擬機器之間必須互相配合溝通來完成網路的存取，因此使得客虛擬機器的網路吞吐量降低。

在這篇論文中，我們提出一個 driver-domain-bypassing network (DBNet) 架構，藉此降低主客虛擬機器之間溝通所造成的額外負擔，並且提高客虛擬機器的網路吞吐量。我們將網路卡驅動程式從主虛擬機器中搬移到 virtual machine monitor (VMM)中，但是為了避免複雜化 VMM，我們只搬移驅動程式中和效能最相關的部分，其餘的部分保留在主虛擬機器中。另外，為了讓 VMM 可以正確地將封包發送到遠端機器或是客虛擬機器，並且避免複雜化 VMM，我們在 VMM 中實做了一個簡化過的 bridge。最後，為了保護 VMM 不會受到驅動程式錯誤的影響而造成當機，我們將網路卡驅動程式放在一個獨立的 segment 中，並且降低網路卡驅動程式的執行權限，藉此保護 VMM。

在測量效能的實驗中顯示，DBNet 架構可以有效的提升客虛擬機器的網路效能。在只有一個客虛擬機器的環境下，且在傳輸封包的測試中幾乎可以達到和 Linux 相同的效能，而在接收封包的測試中則可以用較少的 CPU utilization 達到更多的網路吞吐量。在多個客虛擬機器的環境下，不論是傳送或是接收封包的測試中，DBNet 都可以有效的達到比原本 Xen 網路架構更好的效能。

Improving Xen network throughput by bypassing driver domain

Student: Han-Yu Tai **Advisor: Prof. Ruei-Chuan Chang**

Computer Science and Engineering College of Computer Science

National Chiao Tung University

Abstract

In the original Xen network architecture, each guest virtual machine has to access the network through a host virtual machine. The most advantage of this architecture is to reuse device drivers within the operating system running within the host virtual machine. However, this architecture causes lots of additional overhead to perform communication between the guest virtual machine and the host machine while reduces the network throughput.

In this thesis, we propose a driver-domain-bypassing network (DBNet) architecture to reduce the communication overhead and increase the network throughput of a guest virtual machine by bypassing the host virtual machine. We migrate a NIC driver from the host virtual machine into the virtual machine monitor (VMM). In order to avoid complicating the VMM, we only move the performance-critical part of the NIC driver into the VMM and keep other parts in the

host virtual machine. Moreover, for allowing the VMM can dispatch a packet to a target guest domain or a remote machine correctly, and avoiding complicating the VMM, we implement a simplified bridge in the VMM. Finally, to protect the VMM from crashing by driver faults, we put the NIC driver into an independent driver segment and lower its privilege level.

Our performance measure shows that in the one-guest-domain environment, the DBNet architecture can achieve nearly the same performance with Linux in TCP transmission test, and achieve higher network throughput while use less CPU utilization in TCP reception test. Beside, DBNet architecture can also achieve better performance for each guest virtual machine than the original Xen network architecture in the multi-guest-domain environment.

致謝

首先感謝我的指導老師 張瑞川教授以及學長 張大緯教授。這兩年在兩位老師費心的教導下，學生方能順利完成此篇論文。於受業期間，老師們耐心的指導我正確的研究態度與研究方法，讓我受益良多。在撰寫論文期間，感謝大緯學長給予的建議。感謝在實驗室一起努力的同學們，宗恆，旻儒，子榮，彥百，和你們一起互相討論才能夠順利地解決許多問題。感謝博士班學長國政以及亭彰在找題目時給我的建議，也非常感謝明幫學長給我許多 Xen network 和 assembly language 方面的觀念，並給於許多方面的幫助。還要謝謝智文和昱雄兩位學弟的幫忙，讓我們可以專心的完成論文。

感謝我的父母和妹妹，在我念研究所的日子裡不斷的關心和鼓勵我。最後要特別感謝我的女朋友瓊儀，尤其是在碩二撰寫論文時能體諒我的壓力，陪伴我度過這些日子，讓我覺得很窩心、感動。最後僅以此論文獻給我最親的家人、女朋友以及所有關心我的人，由衷地謝謝他們。

目錄

圖目錄

Chapter 1. Introduction

1.1. Motivation

Due to IT industry grows up dramatically; enterprises demand to enhance not only the system performance, but also the system availability. Event a transient fault may crash the whole system and lead to a great financial loss for some e-commerce web site, for example, ebay and Amazon etc. Beside, enterprises use a large number of servers internally for different purpose. The mass of servers will increase the administration and maintenance cost.

Virtualization technology provides enterprises with a great solution. Enterprises can reduce the system administration cost by consolidating multiple physical machines to administrate them with a single console. Dynamically transfer a virtual server between different physical machines according to the workload can increase the hardware utilization. Dynamic transferring technology can also repair hardware without losing the service, and thus enhances the system availability.

There are some common virtual machine productions on the market. For example, Workstation, ESX server and Virtual Center [29] are developed by VMware Corporation; Virtual PC [28] is developed by Microsoft Corporation. In terms of open-source, Xen is an virtual machine software with fine performance which is developed by Cambridge University. Many researches in virtual machine base on this platform [1], [9], [16], [18], [19].

Xen uses split driver model [9]. Only driver domain (dom0) can access hardware device directly through the device driver residing in it. All guest virtual machines (guest domain; domU) which need to perform device I/O have to send I/O requests to the backend driver residing in dom0 through its own frontend driver. The backend driver will then forward the I/O requests to the device driver. The advantages of this model are that all domUs can reuse the device drivers residing in dom0 and all devices are centralized and managed by dom0.

However, the disadvantage of the split driver is that it results in lots of inter-domain communication which causes context switch because every domain executes in an individual address space (like different processes in OS). The additional context switch will increase the TLB miss rate which is caused by TLB flush. Inter-domain communication also complicates device I/O. Thus it increases the working set size and L2 cache miss rate. In the Xen network subsystem, both dom0 and domU need to access the packets. Xen performs page grant operation between dom0 and domU to avoid copying whole packets. Although page grant operation can avoid the overhead of coping packet, however, it still increases the working set size and L2 cache miss rate [19].

In order to improve the network throughput in Xen, previous researches propose OS bypass [22] and VMM bypass [16]. They allow a process or a guest domain accessing physical NIC directly to reduce the context switch and mode switch. However, both OS bypass and VMM bypass need hardware support. A. Menon [18] proposes Xen network optimization which uses TSO to reduce the page grant operation. However, TSO can only reduce the page grant operation, but cannot eliminate it. P. Willmann [30] allows NIC transferring incoming packets into domU without any page grant operation. However it needs hardware support too.

In this paper, we propose a new network architecture on virtual machine to improve the network throughput of Xen and overcome the drawbacks which we described in the above. We divide our works into three parts. Firstly, we do not need any inter-domain communication to transmit/receive a packet. We execute the backend driver and NIC driver in VMM safely. When a guest domain needs to transmit/receive a packet, it will request the backend driver residing in the VMM, instead of that residing in the dom0. Thus we avoid the TLB flush. Secondly, we eliminate the page grant operation in packet transmission and reduce the page allocation overhead of dom0 in packet reception. By executing the backend driver

and NIC driver in VMM, they can access the address space of domU directly without page grant operation. Thirdly, in order to avoid complicating VMM, we only move the performance-critical part of driver into VMM. The remainder parts of driver are kept in dom0.

Although executing the backend driver and the NIC driver in VMM can improve the network throughput, however, the NIC driver will result in security problem. In order to avoid VMM be crashed by the NIC driver residing in it, we use x86 hardware protection mechanism to limit the privilege and the memory range of the NIC driver.

We implement our architecture on Xen version 3.0.4 and xenoLinux version 2.6.16.33. According to the result of evaluation, we can improve at least 40% throughput in TCP packet transmission/reception.

1.2. Organization

The rest of the thesis is organized as follows. We describe the related works in chapter 2. In chapter 3, we explain the design and implementation of driver-domain-bypass architecture and then evaluate this architecture in chapter 4. Finally, the thesis is concluded in chapter 5.

Chapter 2. Related Works

In this chapter, we classify related works as two categories. The first one focuses in kernel protection or VMM protection and the other one focuses in the network throughput of virtual machine environment. We describe these researches as following.

2.1. Kernel and VMM Protection

A. Chou's [6] proposed that device drivers have higher probability to make a system crash than other components of the operating system. This is because that most device drivers are provided by manufactures and written by programmers with less experience in kernel development. They have higher probability to write unstable code. Besides, according to the R. Short's research [23], 85% crash of Windows XP is result from defective device drivers. Therefore, we have to protect kernel and VMM from drivers faults.

Most device drivers are written by C language which can transfer a variable type arbitrarily, for example, from pointer type to unsigned long type. However, C language has no type-safe property and transfer between different types may cause a program to access a wrong memory address. Singularity [12], [24] is an operating system developed by Microsoft corporation and is written by C# language which has type-safe property. With this property, a compiler can restrict the transfer between different types strictly and prevent a device driver from accessing incorrect memory addresses. However, the most disadvantage of using type-safe language is that all device drivers have to be rewritten.

Micro-kernel [8], [17] executes the most functionalities of an operating system in user mode. Since a device driver is a process in a system, it will not influence the kernel when it faults. However, micro-kernel architecture results in low performance in early machine because it causes lots of IPC and context switch. P. Chubb [7], [14] migrate Linux device drivers to user mode and emphasize that the performance of user-mode drivers can approach

to the performance of kernel-mode drivers. Besides, Microsoft Corporation proposed User-Mode Driver Framework (UMDF) [11], [20] which also executes device drivers in user mode to improve the system stability. However, migrating device drivers into user mode will also need to rewrite them.

Nooks [25], [26] proposed that puts device drivers and a kernel in the same address space but different protection domains. Each device driver uses a unique page table which has the same address mapping with the kernel's page table, but the entries which correspond to the kernel's address space are set read-only. Before calling into a device driver, the kernel changes the original page table into the device driver's page table. Therefore, the device driver can read the whole address space of the kernel but cannot write it. However, Nooks still executes device drivers with the highest privilege while cannot prevent device drivers from executing privilege instructions.

Xen [9] and J. LeVasseur [15] execute the operating system which has completed device drivers in an independent logical fault domain (i.e. driver domain). This method can prevent device drivers from accessing the address space of VMM and other domains directly when they fail. Besides, because the driver domain does not execute with the highest privilege, all device drivers residing in it cannot execute privilege instruction or access I/O port directly. However, this method can only protect VMM and other domains, but cannot protect the driver domain itself. Device drivers may make the driver domain crash when they fail.

T. C. Chiueh [5] isolates the address space of a kernel and its kernel modules by using x86 hardware mechanism. Every kernel module is executed in protection ring 1 to prevent it from executing privilege instructions directly, and it is executed in an independent segment to prevent it from accessing the address space of the kernel directly. They also proposed a method that evades the x86 protection rule to allow a kernel calling into a lower-privilege kernel module. However, they do not consider that every device driver module needs to use

different I/O port ranges, and they do not prevent a device driver from accessing an I/O port range which does not belong it.

2.2. Virtual Machine Network Improvement

OS-bypass architecture [22]allows an application accessing a NIC which supports TCP-offload, for example, Myrinet [3] and iWarp Ethernet [10] directly without going through the kernel. This architecture reduces the context switch overhead between application and kernel. J. Liu [16] proposed a VMM-bypass architecture base on Xen, which allows a guest domain performing time-critical I/O operations to access an Infiniband NIC directly without involving of the driver domain and VMM. This architecture reduces the inter-domain communication and eliminates page grant operations between a guest domain and the driver domain. CDNA architecture [30] proposed a new Gigabit Ethernet NIC which supports multiplexing and demultiplexing. CDNA allows multiple guest domains transmitting or receiving packets concurrently without going through the driver domain, and thus eliminates the inter-domain communication and page grant operations. However, these architectures all need specific hardware support.

A. Menon [18] optimizes the Xen network architecture. They allow a virtual NIC supporting TCP segment offload (TSO), which increases the MTU of a packet, and thus allow transmitting data with fewer packets. This method reduces the overhead of inter-domain communication and page grant operation. However, it cannot eliminate these overhead.

Chapter 3. Design and Implementation

In this chapter, we describe the design and implementation of the driver-domain-bypass network (DBNet) architecture. In Section 3.1, we give an overview of the proposed DBNet architecture, and present its major differences from the original Xen network architecture. In Section 3.2 and 3.3, we describe the issues of migrating driver code into VMM and protecting VMM from driver faults. Finally, we explain the details of the packet transmission/reception flow on the DBNet architecture in Section 3.4.

3.1. Driver-Domain-Bypass Network Architecture

Before describing the driver-domain-bypass network architecture, we first present the packet transmission and reception flows in the original Xen network architecture in order to show the differences between the two architectures.

Figure 1(a) shows the packet transmission flow. As mentioned before, packet transmission is done by sending a request from the frontend driver residing in the user domain to the backend driver residing in dom0. Before sending the request, the frontend driver performs a page grant operation[1], granting dom0 read-only access to the domU's page that contains the packet. The page grant operation is needed for the backend driver and the physical NIC driver to process the packet. On receiving the request, the backend driver maps the packet page with read-only permission into the address space of dom0 and asks the bridge to identify the target physical NIC. Then, the driver corresponding to the target physical NIC is responsible for transmitting the packet. After the packet has been transmitted successfully, the backend driver unmaps the packet page from the address space of dom0 and informs the frontend driver about the transmission completion.

Figure 1(b) shows the packet reception flow. With a NIC card that supports DMA, an

[1] Page grant operation allows a domain to obtain access permission of pages that belong to another domain. It also allows transferring ownership of pages between two domains.

incoming packet will be transferred into dom0 directly by using DMA. Then, the bridge residing in dom0 finds out the virtual NIC of the target domain, and asks the backend driver to transfer the packet to that domain. To avoid page copying, the backend driver performs another kind of page grant operation that transfers the ownership of the packet page to the target domain. The frontend driver residing in the target domain can then remap the page into its address space, and notify the kernel about the packet reception. Note that such page ownership transfers increase the memory size of the target domain and decrease that of dom0. For balancing the memory sizes, a domain has to transfer some free pages to the VMM before it can receive any packets, and dom0 can claim free pages from the VMM when the number of its free pages is lower than a threshold.

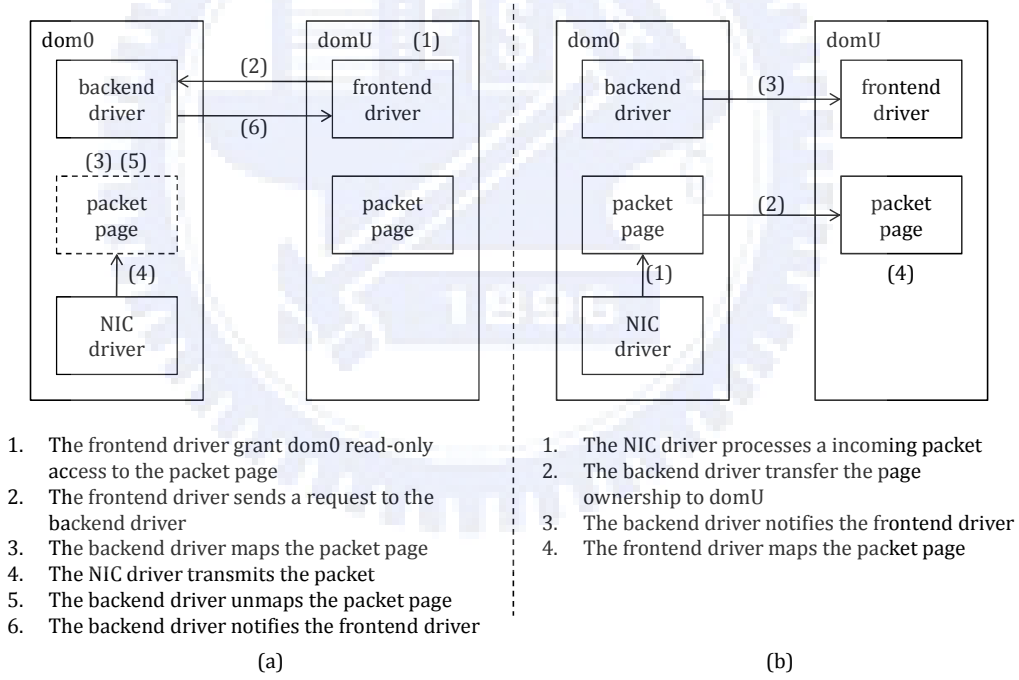
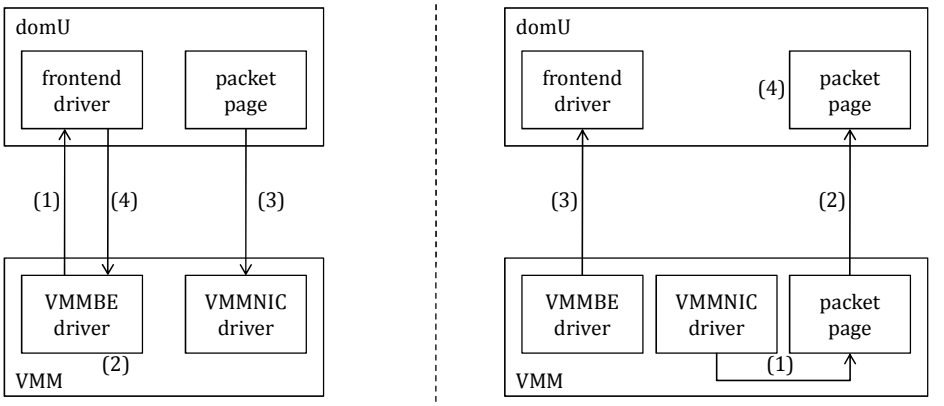


Figure 1. Network Architecture of Xen



1. The frontend driver sends a request to the backend driver
2. The VMMBE driver finds out the physical NIC which the packet will pass through
3. The VMMNIC driver transmits the packet
4. The VMMBE driver notifies the frontend driver about transmission completion

(a)

1. The VMMNIC driver processes the incoming packet
2. The VMMBE driver gives the packet page to domU
3. The VMMBE driver notifies the frontend driver
4. The frontend driver maps the packet page

(b)

Figure 2. Driver-domain-bypass Network Architecture

The driver-domain-bypass network architecture is described as follows. As shown in Figure 2, we move backend driver and NIC driver from dom0 into VMM, which are called VMMBE driver and VMMNIC driver respectively in the rest of the thesis. When domU demands to transmit/receive a packet, it asks VMM instead of dom0 to handle the packet transmission/reception. Note that handling packet transmission/reception in VMM avoids flushing TLB because switches between a domain and VMM do not need context switches. Thus driver-domain-bypass network architecture does not increase the TLB miss rate.

Figure 2(a) shows the flow of packet transmission under the DBNet architecture. In order to transmit a packet, the frontend driver sends a request to VMMBE driver, which retrieves the memory address of the packet from the request and finds out the target physical NIC. Then VMMBE asks VMMNIC driver to transmit the packet. After the packet is transmitted successfully, VMMBE driver notifies the frontend driver about the transmission completion. Note that packet transmission does not involve dom0. Thus, no extra domain switches are required. Furthermore, no page grant operation is needed because the drivers residing in

VMM can access the address space of domU directly.

Figure 2(b) shows the flow of packet reception. An incoming packet is transferred into VMM directly by using DMA. In order to find out the target domain, VMMBE driver looks up the destination MAC address of the packet in its look-up table. Then, VMMBE driver performs a page grant operation to transfer the ownership of the packet page to the target domain and notifies the frontend driver, which remaps the packet page into its address space and notifies its kernel of about the packet reception. As mentioned above, the page ownership transfer increases the memory size of the target domain. To balance the memory sizes, the target domain should transfer some free pages to VMM before it can receive any packet. Note that the memory balance logic in DBNet is simpler than that in the original Xen network architecture since the former only involves the target domain and the VMM. This helps to reduce the CPU load and the working set size. Moreover, packet reception does not involve dom0, and thus no extra domain switches are required.

3.2. Driver Migration

In this section we describe the issues and details of migrating the backend driver and the NIC driver into VMM. Xen was designed with a micro-kernel concept that keeps only the most critical functionality such as domain scheduling and memory management in VMM. Other management functions such as device drivers are kept in dom0. This helps to reduce the complexity of the VMM and thus makes it easier to guarantee the reliability of the VMM.

To maintain the design concept, we do not migrate all of the driver code into the VMM. Instead, only the performance critical driver code is migrated. Specifically, we migrate the packet transmission/reception and interrupt service routines into VMM. Therefore, the original backend driver is split into two parts: VMMBE driver running in the VMM and the D0BE driver running in the dom0. Similarly, the original physical NIC driver is also split into two parts: VMMNIC driver running in the VMM and the D0NIC driver running in the dom0.

In addition to code splitting, we also have to consider the following issues during driver migration. First, the VMM part of the drivers may use some data structures and functions that are not implemented in VMM originally. Second, the split drivers should be cooperate with each other to maintain the functional correctness. In Section 3.2.1, we describe the data structures and functions that need to be implemented for network driver migration. In Section 3.2.2, we describe how to split original drivers into the D0 drivers and the VMM drivers, and the cooperation between them.

As mentioned above, packet bridging that maps a virtual NIC to a physical NIC and vice versa is needed when transmitting or receiving a packet. However, VMM does not have the bridging capability, and migrating the original bridge from dom0 to VMM requires too much effort. Therefore, we implemented a simplified bridge in VMM, which performs the mapping by using a look-up table. We will describe the simplified bridge in Section 3.2.3.

3.2.1 Data Structures Definition and API Implementation

In the original Xen network architecture, the NIC driver records the information about a physical NIC and a packet by using the net_device and sk_buff data structures, respectively. In addition, the backend driver uses the netif_t data structure to describe a virtual NIC. However, such data structures are not defined in VMM because VMM does not need to process packets or access NICs. Since the performance critical driver code, which involves handling both packets and NICs, is migrated into VMM, we have to define the aforementioned data structures in VMM. We did this simply by copying the definition of the data structures from dom0.

Besides defining the data structures, we have to implement the functions needed by VMMBE/VMMNIC drivers. The functions include socket buffer API and DMA API. The former is used by VMMBE/VMMNIC drivers to access a packet (a socket buffer records all information about a packet), and the latter is used by VMMNIC driver to perform DMA

operation between physical NIC and main memory. Note that, instead of implementing the APIs from scratch, we copied the implementation from the dom0 and performed some modifications. Some socket buffer API needs to call memory management functions (e.g., kmalloc()/kfree()), which is provided by the dom0 kernel. We replaced these function calls with the memory management functions provided by VMM (e.g., xmalloc()/xfree()). Similar to the socket buffer API, some DMA API needs to invoke functions to translate virtual addresses to physical addresses, and vice versa, which are provided by dom0 kernel. We replaced the invocations of these functions with those of the functionally-equivalent functions in VMM.

3.2.2 Driver Code Migration

In this section, we firstly define the performance-critical code of the NIC driver and the backend driver and then describe how to split them into D0/VMM drivers. D0 drivers are responsible for copying initialized data structures into VMM and then notifying VMM drivers to allocate I/O resources. In the remainder of this section we firstly define the completion time point which D0 drivers can copy the data structures safely and then describe the cooperation between D0/VMM drivers.

We divide the functions of a NIC driver into five parts: packet transmission, packet reception, interrupt handling, initialization, and utility functions. The former three parts are performance-critical since they are highly related to network throughput, while the other parts are used occasionally or even only once during the runtime of the driver. Thus, we migrate the former three parts into VMM to be VMMNIC driver and keep the other parts in dom0 to be the D0NIC driver. Taking the D-Link DL2000 Gigabit Ethernet driver (i.e., dl2k.c in the Linux 2.6.16.33 source tree) as an example, we migrate the start_xmit(), receive_packet(), rio_interrupt() functions and their helper functions into VMM to be VMMNIC driver. Other functions in the dl2k.c file such as rio_probel(), rio_open(), rio_close() and rio_ioctl()

functions are kept in dom0 as the D0NIC driver.

Similarly, the performance critical part of a backend driver resides in the netback.c file, which implements the communication mechanism with the frontend driver. Therefore, we moved the whole netback.c file into VMM to be VMMBE driver. Other parts of the backend driver such as the initialization and utility functions are kept in dom0 to be the D0BE driver.

As mentioned before, net_device and net_if structures are used to represent physical and virtual NIC devices, respectively. According to the aforementioned split rule, these data structures have to be initialized by the D0 drivers and then managed by the VMM drivers. This is done by copying the content of the data structures after they are completely initialized. To achieve this, we have to identify the initialization completion time of the data structures.

We identify the initialization completion time of the data structures by tracking the initialization process of the corresponding drivers. For the net_device data structure, the D0NIC driver extracts hardware information (e.g. MAC address, I/O port, range IRQ number, and etc) from the physical NIC, saves the information into the net_device data structure, and registers it to dom0's kernel. Finally, the D0NIC driver activates the physical NIC. Since the net_device data structure should have been initialized completely before the activation of the physical NIC, we define the initialization completion time of the data structure as the point that D0NIC driver activates the physical NIC, and we copy the content of the structure into VMM at that time. The initialization of the net_if data structure follows a similar way except that the set up of that data structure and the activation of the virtual NIC are all done by the frontend driver. Therefore, we define its completion time as the point that the frontend driver activates the virtual NIC and copy its content into VMM at that time.

In addition to copying the data structures, we have to update all pointer fields in those data structures since the pointers reference data in dom0. Thus, the pointer fields should be

updated to point to the equivalent data in VMM. For example, the net_device data structure includes some function pointers which point to the utility functions of the NIC driver. We have to update these pointers to the equivalent utility functions of VMMNIC driver after copying the net_device data structure from dom0.

The initialization of net_device/netif_t data structures are presented in Figure 3. Although we keep the code for initializing and opening the device in the D0NIC driver, we take out the code for allocating I/O resources, such as IRQ numbers, I/O ports and DMA ring buffers, from that driver. Since the performance critical code in VMM uses those IO resources, we should allocate them in VMM. To achieve this, the D0NIC driver issues a hypercall to VMMNIC driver to perform the IO resource allocation. Note that, under this situation, the D0NIC driver can not access I/O resources directly. Instead, it has to use a hypercall to send IO request to VMMNIC driver. When the net_device data structure is completely initialized, VMMNIC driver copies it from dom0 and updates its pointer fields. For the netif_t data structure, we modify the function which activates the virtual NIC in the frontend driver to send a *copy* message to the D0BE driver, which asks VMMBE driver (through a hypercall) to copy the netif_t data structure to VMM and update its pointer fields.

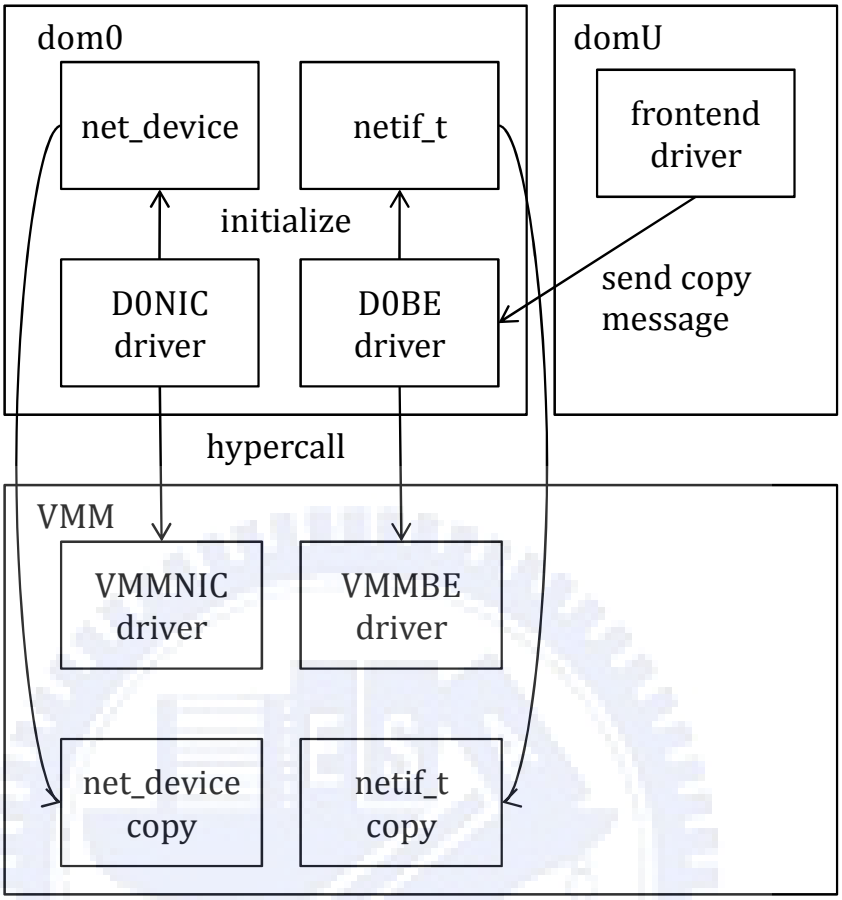


Figure 3. Cooperation between D0 Drivers and VMM Drivers

3.2.3 The Simplified Bridge

Before the presentation of the in-VMM simplified bridge, we firstly introduce the architecture of the bridge used in Xen. The purpose of the bridge is to find out the destination NIC, either physical or virtual, according to the destination MAC address of a packet. Figure 4(a) shows the abstract architecture of the bridge in Xen. On transmitting a packet, the backend driver hands the packet to the bridge, which finds out the target physical NIC and then asks the driver of that NIC to send the packet. Similarly, the physical NIC driver hands the packet to the bridge when a packet is received. The bridge then finds out the target virtual NIC and asks the backend driver to forward the packet to the domain corresponding to the virtual NIC.

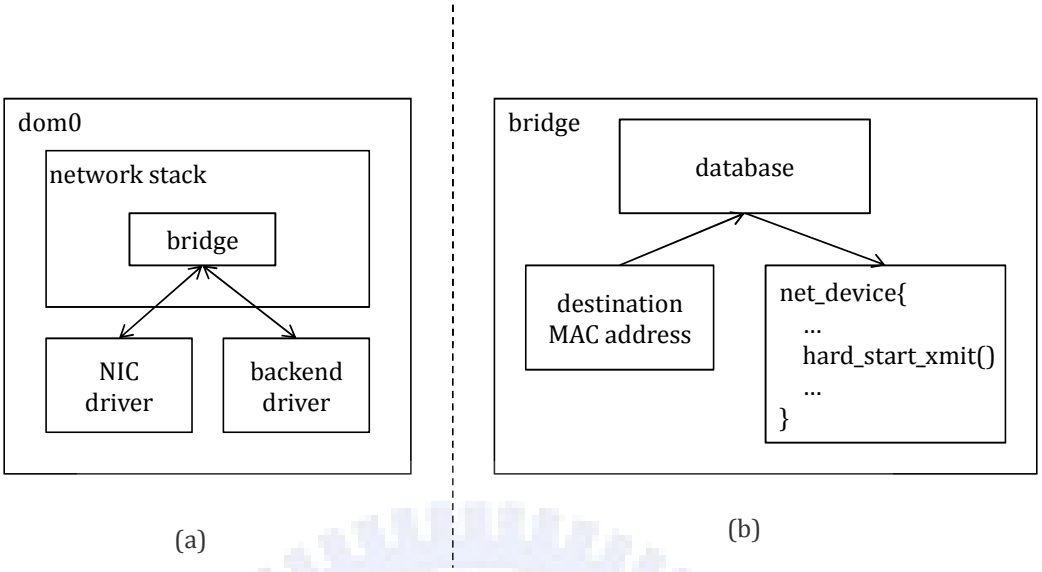


Figure 4. The Bridge in Xen

Figure 4(b) shows the internal architecture of a bridge under the virtual machine system configuration. When the bridge receives a packet from either a physical NIC driver or a virtual NIC driver (i.e., the backend driver), it uses the destination MAC address of the packet to consult its internal database so as to find out the target net_device data structure through which the bridge can forward the packet to the target driver by calling hard_start_xmit() function pointer. Note that in the original Xen, either a physical NIC or a virtual NIC has to register its net_device data structure to dom0 kernel. For example, if the destination MAC address of the packet belongs to the a virtual NIC, the bridge will find out its net_device data structure through which the bridge can ask the backend driver to transmit the packet by calling hard_start_xmit() function pointer. If the MAC address of the packet belongs to a remote machine, the bridge will also find out its net_device data structure through which the bridge can asks the NIC driver to transmit the packet by calling hard_start_xmit() function pointer.

We can replace the original bridge with a simple mapping between virtual and physical NICs. Note that a virtual NIC will only send packets through a specific physical NIC because

a bridge only includes a physical NIC. In the original Xen, a user can assign a physical NIC to a specific bridge when starting the Xen daemon program and assign each virtual NIC to a specific bridge when starting a guest domain. We add a field which is the pointer of net_device data structure in the netif_t data structure to record the mapping. When domU wants to send a packet to a remote machine, VMMBE driver can find out the net_device data structure of the target physical NIC by using this field and call into VMMNIC driver. For the packet reception, we implement a look-up table in VMMBE driver. When VMMNIC driver forwards an incoming packet to VMMBE driver, it retrieves the destination MAC address of the packet and queries the look-up table to find out the netif_t data structure of the target virtual NIC. Then VMMBE driver can find out the event channel number from the netif_t data structure and notify the target guest domain.

3.3. VMM Protection

In this chapter, we describe VMM protection. To avoid VMM being crashed by device driver, we have to prevent device driver from arbitrarily accessing VMM resources which include privilege instructions, I/O port and VMM memory. In section 3.3.1, we describe our design by using x86 hardware protection mechanism. We execute VMMNIC driver in an independent driver segment which is in the address space of VMM with lower privilege. Section 3.3.2 describes the memory layout of the driver segment. However, executing VMMNIC driver in the driver segment will result in additional problem that VMM cannot call into VMMNIC driver directly. Therefore, we use a tricky method to solve this problem and describe it in section 3.3.3.

3.3.1. Segmentation and Privilege Level Protection

In this section, before describing the protection mechanism, we firstly describe three goals which we want to achieve to protect VMM. First, VMMNIC driver cannot use privilege instructions directly. Privilege instructions are the most important instructions in a system, for

example, enable/disable interrupts and change page table. Only the program with the highest privilege can use privilege instructions. However, many operating systems like Linux execute kernel and device drivers with the highest privilege. Kernel may crash because device drivers use privilege instructions inappropriately. For example, a device driver may disable interrupts without enabling them again and thus kernel will never receive interrupts. Second, VMMNIC driver can only access the I/O port which belongs to it. I/O port is used by a device driver to access the registers of a device. If a NIC driver can access the I/O port which belongs to a disk driver arbitrarily, it may write wrong data into the disk. Third, VMMNIC driver cannot write data into VMM directly. Since many operating systems like Linux execute kernel and device drivers in the same address space. When a device driver fails, it may write data into wrong memory address of kernel and makes kernel crash.

We use x86 hardware mechanism to protect VMM from the faults of VMMNIC driver. We execute VMMNIC driver in an independent driver segment with lower privilege. By lowering the privilege of VMMNIC driver, we can achieve the first two goals. X86 architecture defines four privilege levels in a system from ring 0 which has the highest privilege to ring 3 which has the lowest privilege. Xen executes VMM in ring 0 and executes all domains in ring 1. Since we execute VMMNIC driver in ring 1, if it uses any privilege instruction, it will violate the x86 protection rule. The system will raise an exception to notify VMM. Therefore, we can achieve the first goal. In x86 architecture, a system uses an I/O port bitmap to describe the permission of each I/O port. When a program which does not have the highest privilege wants to access an I/O port, the system will check the access permission of the program by using the I/O port bitmap. If the program has no permission to access the I/O port, the system will raise an exception to notify VMM too. We can set the I/O port bitmap properly to allow VMMNIC driver only access the I/O port of the NICs. Therefore, we can achieve the second goal. Finally, executing VMMNIC driver in an independent driver

segment allows us achieving the third goal. VMMNIC driver can only access the memory in the driver segment or else the system will also raise an exception to notify VMM.

3.3.2. Memory Layout of The Driver Segment

In this section, we describe the memory layout of VMM and the driver segment and present it in Figure 5. Besides, we also describe memory copy avoidance between the VMM segment and the driver segment. Xen assigns the last 64 MB of linear address space to VMM and assigns the remainder of linear address space to domains. We add the driver segment which has 16 pages in VMM. In order to add a new segment, we add a new segment descriptor in GDT and set its DPL as 1 to represent that the driver segment is executed in ring 1. Because the address space of the driver segment is a subset of that of VMM, VMM can access the driver segment arbitrarily while VMMNIC driver can only access those 16 pages in the driver segment.

We divide the driver segment into several partitions. The driver code partition is used to store VMMNIC driver code. The DMA ring partition is used by VMMNIC driver to perform DMA operation. The stack partition is used as the stack of VMMNIC driver. We use the parameter area partition to avoid memory copy between VMM segment and the driver segment. Because VMMNIC driver can access only the memory inside the driver segment, before calling into VMMNIC driver, VMM has to copy all outside data which is needed by VMMNIC driver into the driver segment However, it will cause too much overhead if we copy a large data structure. Therefore, we allocate all data structures needed by VMMNIC driver in the parameter area partition. For example, we allocate all net_device data structures in the parameter area partition. VMMBE driver can pass the offset of net_device data structure related to the start address of the driver segment as a parameter to VMMNIC driver, and then VMMNIC driver can access net_device data structure directly. Note that VMMNIC driver can transmit a packet in domU while does not need to access the memory outside the

driver segment. It is because that VMMNIC driver only needs the physical address of the packet which is passed by VMMBE driver as a parameter to perform DMA operation.

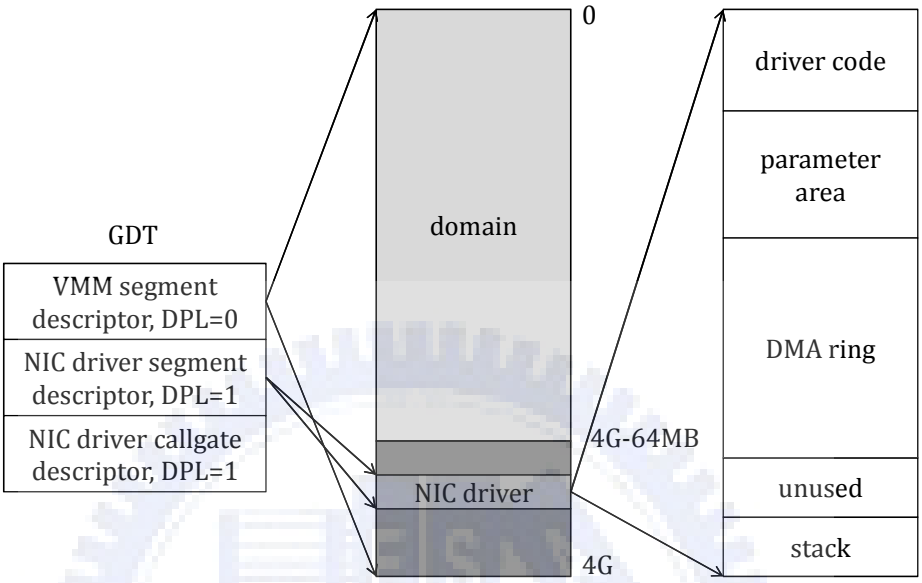


Figure 5: Memory layout of the driver segment

3.3.3. x86 Protection Rule Avoidance

In this section, we describe the method which is used to avoid x86 protection rule. In x86 architecture, a high-privilege program cannot call into a low-privilege program directly. Normal program flow is that a low-privilege program can request a high-privilege program through a system call to perform some specific tasks and the high-privilege program can only "return" to the low-privilege program. Therefore, VMM cannot call into VMMNIC driver directly.

In order to allow VMM calling into VMMNIC driver directly, we prepare a fake stack and pretend that VMM returns to VMMNIC driver. In x86 architecture, when a high-privilege program returns to a low-privilege program, lret instruction will retrieve the return address and the point of the low-privilege stack from current (high-privilege) stack. The content of the high-privilege stack is presented in Figure 6(a) and the content of the fake stack is presented

in Figure 6(b). We imitate the high-privilege stack to fill the fake stack. We push the address of the stack partition residing in the driver segment and the address of the entry point of VMMNIC driver into the fake stack orderly. After preparing the fake stack completely, VMM uses lret instruction to enter in VMMNIC driver.

In order to allow VMMNIC driver returning back to VMM, we add a new callgate and pretend that VMMNIC driver calls into VMM through the new callgate. We add a new callgate descriptor which is presented in Figure 5 in GDT and set its DPL as 1. VMMNIC driver can use lcall instruction to return back to the VMM through the callgate and the system will switch the stack of driver segment back to the stack of VMM automatically.

Generally, ISR has to be executed with the highest privilege. When VMM receives an interrupt, it will call into the corresponding ISR. Because VMM is executed in ring 0, if the ISR is executed in ring 1, VMM will call into a lower-privilege program directly and violate the x86 protection rule. However, the ISR of VMMNIC driver is executed in ring 1. In order to overcome the problem, we allow VMM calling into the ISR indirectly. We register a fake ISR to VMM which is only responsible for raising a softirq and also register a corresponding handler of the softirq to VMM which is responsible for "returning" into real ISR in the driver segment. When VMM receives an interrupt, it will call into the fake ISR and then the fake ISR will raise a softirq. The handler of the softirq then "returns" into the real ISR directly.

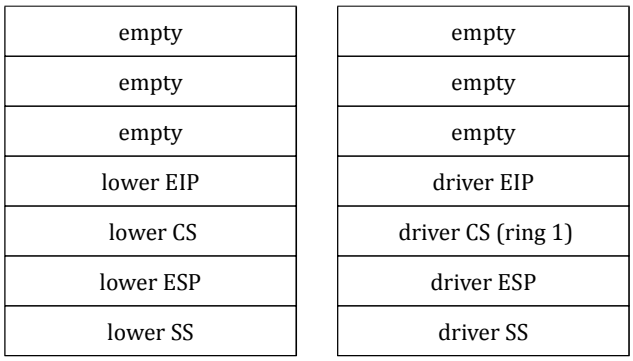


Figure 6: the high-privilege stack and the fake stack

3.4. The packet flow of DBNet architecture

In this chapter, before we describe the packet transmission/reception flow in detail, we describe the scheduling problem in DBNet architecture. We describe an address space problem in packet transmission firstly and then describe an address space problem in packet reception. In Xen architecture, dom0 is responsible for transmitting every packet which comes from any domU. In order to transmit packets fairly, dom0 divides the bandwidth of NIC between every VNIC and schedules every VNIC by round-robin. However, we cannot use round-robin to schedule every VNIC in VMM. In DBNet architecture, VMM transmits a packet directly without page grant operation. However, without page grant operation, VMM can only access the address space of the current domain (i.e. the guest domain which owns CPU currently). A domU can transmit packets only when it becomes the current domain. So we transmit only the packets of current domain and schedule only the VNICs of the current domain by round-robin.

We describe the address space problem in packet reception as following. In DBNet architecture, an incoming packet will be transferred into VMM firstly and then be dispatched to a domU. However, VMM can dispatch only packets which belong to the current domain because VMM can access only the address space of the current domain. Thus every incoming packet has to be stored in VMM temporarily. When the destination domU becomes the current domain, we dispatch the packet. We will explain the flow in detail in the following section.

3.4.1. The packet transmission flow of DBNet architecture

In this section, we describe the packet transmission flow in detail which includes the communication between the frontend driver and VMMBE driver, bridging and the segment switch operation between the VMM segment and the driver segment. The frontend driver is responsible for providing the meta-data of the packet for VMMBE driver. In order to transmit

a packet, the frontend driver puts a request which includes the virtual address and the physical address of the packet into a descriptor ring which is a shared memory between dom0 and domU. Then the frontend driver calls into VMMBE driver through a hypercall with the VNIC id as a parameter.

VMMBE driver is responsible for demanding VMMNIC driver to transmit the packet. First, VMM driver has to retrieve the request. By the VNIC id which passed which hypercall, VMMBE driver can know which VNIC want to transmit packets and find out the netif_t data structure which includes the address of the descriptor ring of VNIC. Then VMMBE driver can retrieves the request from the descriptor ring. Second, VMMBE driver has to find out the PNIC through which it will send the packet. As we mentioned in section 3.2.3, VMMBE driver can find out the corresponding net_device data structure of PNIC by the pointer which recorded in the netif_t data structure.

Third, VMMBE driver performs segment switch operation to enter in/return from the VMMNIC driver. We describe segment switch operation as following. Firstly, VMMBE driver prepares a fake stack which we have mentioned in section 3.3.1. Then VMMBE driver write the physical address of the packet and the offset of the net_device data structure as parameters into the parameter area partition. Besides, VMMBE driver opens the I/O port for VMMNIC driver by setting the I/O port bitmap. Finally, VMMBE driver uses lret instruction to enter in VMMNIC driver and switch the stack of VMM to the stack of VMMNIC driver. After VMMNIC driver completes, it uses lcall instruction to return back to VMMBE driver. The system will also switch the stack of VMMNIC driver back to the stack of VMM automatically.

In order to perform DMA operation, VMMNIC driver retrieves the physical address of the packet and write it into the DMA descriptor ring. Then VMMNIC driver returns to VMMBE driver. VMMBE driver keeps transmitting packets until complete all requests in the descriptor

ring. After the packet has been transmitted, NIC will raise an interrupt. A fake ISR in the VMM will forward the interrupt by raising a softirq. The handler of the softirq then perform segment switch operation to enter in the real ISR of VMMNIC driver. After the interrupt is handled completely, VMMNIC driver cleans the corresponding entry in the DMA descriptor ring and then return. Finally, VMMBE driver is responsible for demanding the frontend driver to clean the packet and the corresponding entry in the descriptor ring.

3.4.2. The packet reception flow of DBNet architecture

The frontend driver is responsible for providing the grant page operation for VMMBE driver. Before receives any incoming packet, the frontend driver has to release some free pages to VMM and put some requests into the descriptor ring in advance. Similar to the frontend driver, before receives any incoming packet, VMMBE driver has to allocate DMA pages in advance in which all incoming packets will be transferred by DMA operation and then write the physical address of DMA pages into DMA descriptor ring.

When the NIC receives an incoming packet, the NIC will transfer the packet into a DMA page by using DMA operation and then raises an interrupt. Note that a DMA page can only store a incoming packet. The real ISR of VMMNIC driver will handle the interrupt as we have mentioned in the section 3.4.1. Then VMMNIC driver cleans the corresponding entry in the DMA descriptor ring and notifies the VMMBE driver.

VMMBE driver is responsible for dispatching the packet to domU. VMMBE driver retrieves the destination mac address of the packet and queries the database to find out the destination domU. Then VMMBE driver inserts the packet into the receive queue which we create for every VNIC in VMM and notifies the frontend driver of the destination domU. When the destination domU becomes the current domain, the frontend driver enters in the VMMBE driver through a hypercall with the VNIC id as a parameter. VMMBE driver then finds the netif_t data structure of VNIC and retrieves the information about page grant

operation from the request in the descriptor ring. Finally, VMMBE driver clear the receive queue of VNIC and performs page grant operation which grant the ownership of all DMA pages to the current domain. The frontend driver is responsible for remapping DMA pages into the address space of domU.

Chapter 4. Evaluation

In this section, we evaluate the performance of the proposed DBNet architecture. We first measure the performance of DBNet architecture in one-guest-domain environment and compare them with original Xen architecture and native Linux. We also measure that in multi-guest-domain environment and compare it with original Xen architecture. Then we adjust two parameters which include rx delay time and rx batch number to describe the relation between throughput, CPU utilization and these parameters. Finally, we measure the overhead of the protection mechanism on DBNet architecture.

The experimental environment consists of a client and a server machine, which are connected through a D-Link DL2000-based Gigabit Ethernet. The server machine run Xen version 3.0.4 with an Intel Pentium 4 2.8GHz CPU and 512MB DDR RAM. The client machine run Linux version 2.6.20 with an Intel Pentium 4 3.2GHz CPU and 512MB DDR RAM. The performance is measured by using Netperf [] benchmark version 2.4. We run Netperf program on the server machine to measure the transmission performance of original Xen architecture, DBNet architecture and native Linux and also measure their reception performance by running Netperf program on the client machine.

4.1. One-domain evaluation

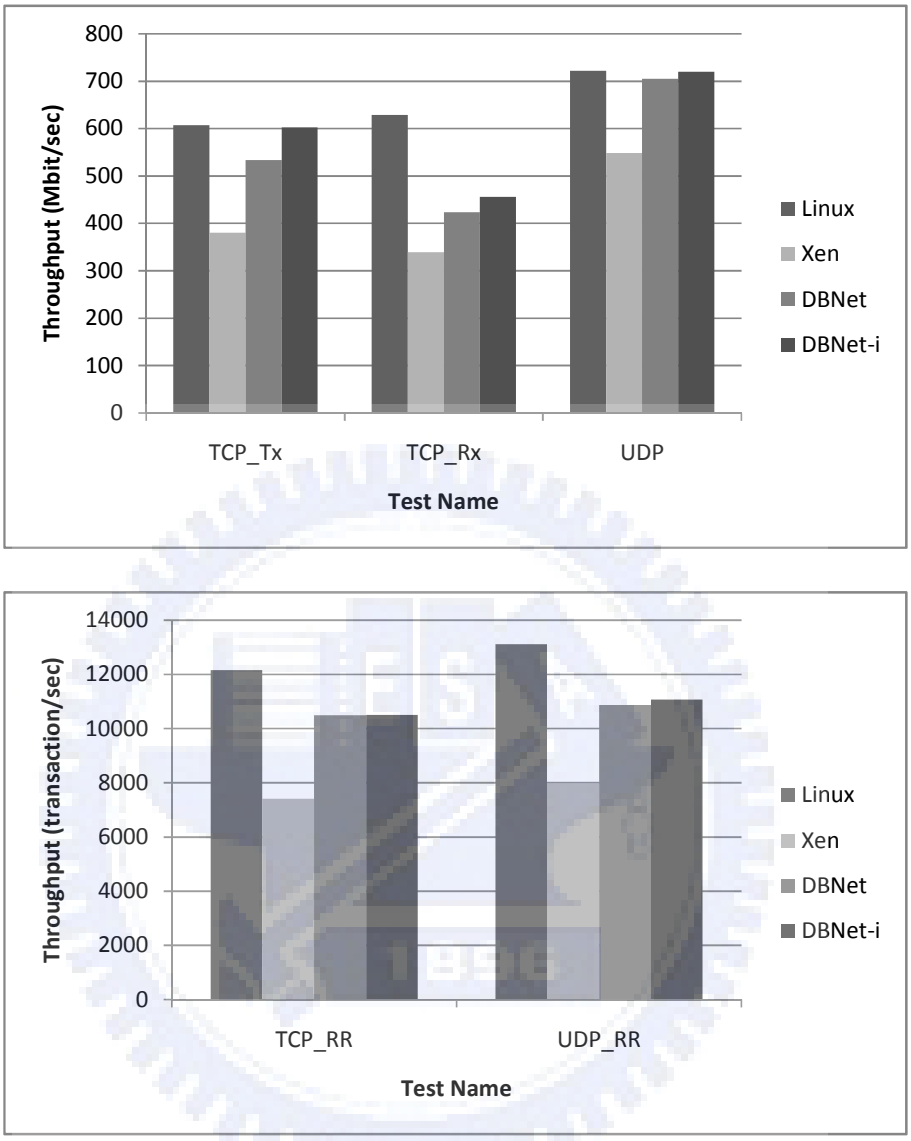


Figure 7. Throughput comparison between Linux, Xen, DBNet, and DBNet-i

Figure 1 shows the throughput comparison between native Linux, Xen and DBNet architecture. The x-axis represents the test type which include TCP transmission, TCP reception, UDP transmission, TCP round-robin and UDP round-robin. The y-axis indicates the throughput number reported by Netperf. From the figure we can see that DBNet architecture can improve 40% and 24% throughput in TCP_Tx and TCP_Rx test respectively and 28% ~ 40% throughput improvement in other tests.

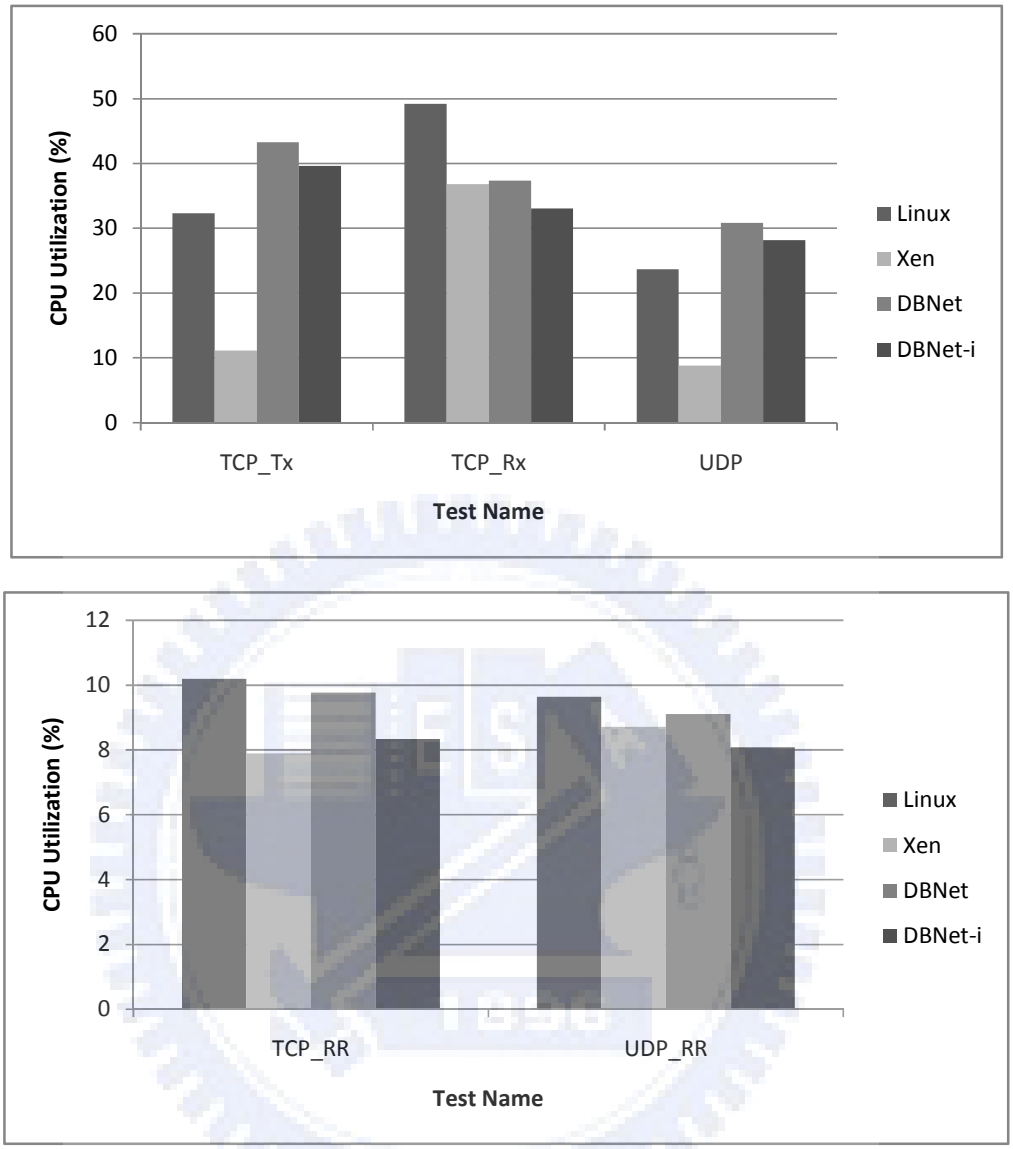


Figure 8. CPU utilization comparison between Linux, Xen, DBNet, and DBNet-i

In addition to the throughput evaluation, we also measure the CPU utilization. Figure 2 shows CPU utilization comparison between Linux, Xen, DBNet, and DBNet-i. From the figure, we can see that DBNet architecture has almost same utilization with Xen architecture in TCP_Rx, TCP_RR, UDP_RR tests while has 28% ~ 40% throughput improvement. However, in TCP_Tx and UDP tests, DBNet architecture seems increase too much CPU overhead. We measure the number of interrupt in Linux, DBNet and DBNet-i architecture and present the result in Figure 3. In the DBNet architecture, the number of interrupt is twice as

that of Linux in TCP_Tx test. This is because that in our original design, when a physical NIC raises an interrupt, it will be handled by a fake ISR which only raise an softirq and return without updating an interrupt-state register of the physical NIC. Instead, we delay the updating time until entering in the driver segment. Without updating the register, the physical NIC considers that the interrupt dose not process completely and raise the same interrupt again. It will cause too much unnecessary interrupt and cause too much overhead.

In order to solve this problem, we move a piece of code which only updates the interrupt-state register of physical NIC from the VMMNIC driver into the fake ISR. In the other word, we advance the updating time upon fake ISR while still perform the correspond operation after enter in the driver segment. We call the new architecture DBNet-i. From Figure 3, we can see that DBNet-i architecture can reduce unnecessary interrupts dramatically in TCP_Tx test and reduce 28% interrupts in TCP_Rx test. From Figure 1 and Figure 2, DBNet-i architecture can approach the throughput of Linux while only have more 5% ~ 6% CPU overhead than Linux in TCP_Tx and UDP test. In other tests, DBNet-i architecture can also increase the throughput while reduce the CPU overhead. Therefore, we choose DBNet-i architecture to measure its performance in the following evaluations.

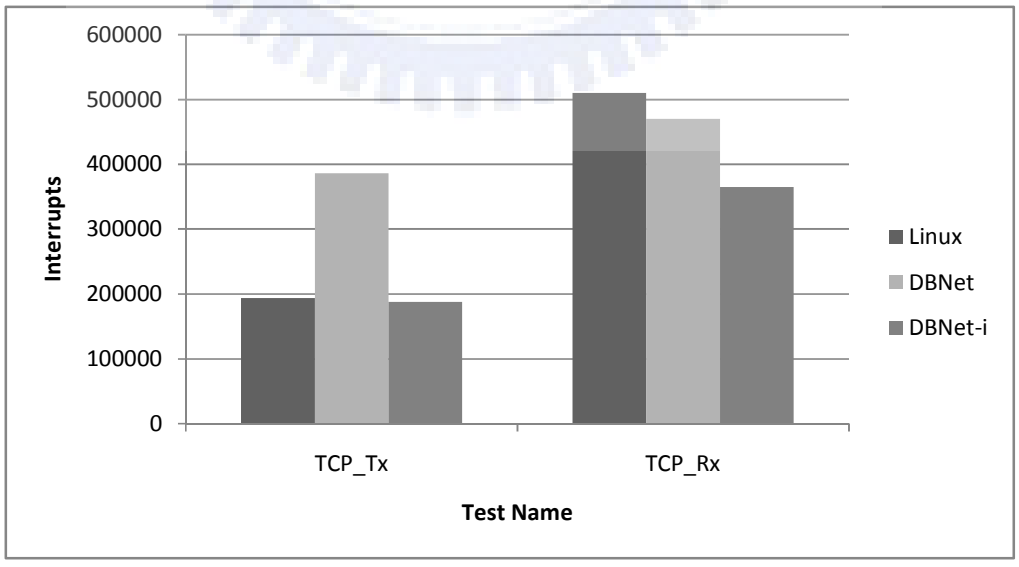


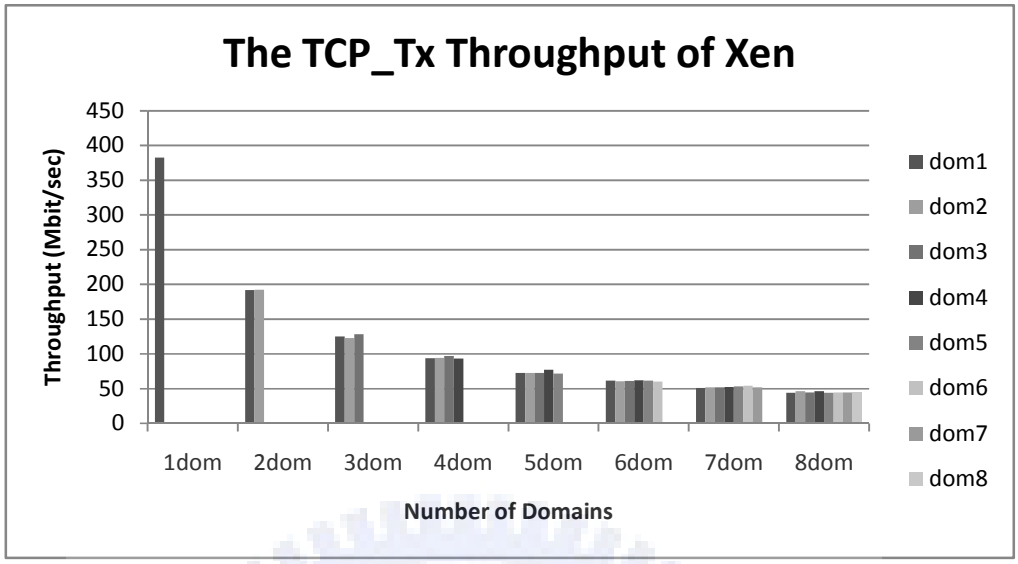
Figure 9. Interrupt comparison between Linux, DBNet and DBNet-i

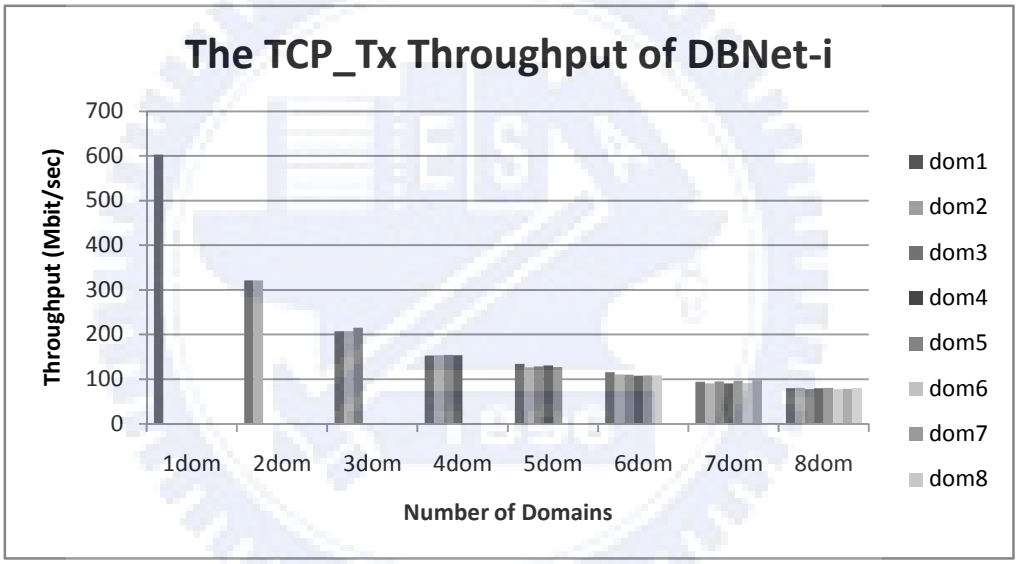
4.2. Multi-domain Evaluation

In this section, we create 1 ~ 8 guest domains on Xen and DBNet-i architectures and measure their performance. Figure 4 shows the TCP_Tx throughput comparison between Xen and DBNet-i architecture. The x-axis represents the number of concurrent activating domains in TCP_Tx test and the y-axis represents the throughput. In the last of Figure 4, above two lines are the total throughput of every domain on DBNet-i and Xen architectures respectively and the below two are the average throughput of every domain. Since the total throughput does not change with x-axis, every domain divides it equally and the throughput of each domain decreases with the x-axis. From the figure we can see that for each domain, DBNet-i architecture can improve at least 50% throughput than Xen in TCP_Tx test.

We also measure the average CPU utilization of each guest domain in TCP_Tx test and present the result in Figure 5. Since the throughput of each domain decreases as the increasing of the number of domains, the average CPU utilization of each domain will decrease as the decreasing of throughput.

For the TCP_Rx test, we use same method to measure the performance of each domain and present the result in Figure 6 and Figure 7. Similarly, for each domain, DBNet-i architecture can improve at least 30% throughput than Xen in TCP_Rx test.


The TCP_Tx Throughput of Xen
Throughput (Mbit/sec)
450
400
350
300
250
200
150
100
50
0
1dom
2dom
3dom
4dom
5dom
6dom
7dom
8dom
Number of Domains
dom1
dom2
dom3
dom4
dom5
dom6
dom7
dom8


The TCP_Tx Throughput of DBNet-i
Throughput (Mbit/sec)
700
600
500
400
300
200
100
0
1dom
2dom
3dom
4dom
5dom
6dom
7dom
8dom
Number of Domains
dom1
dom2
dom3
dom4
dom5
dom6
dom7
dom8

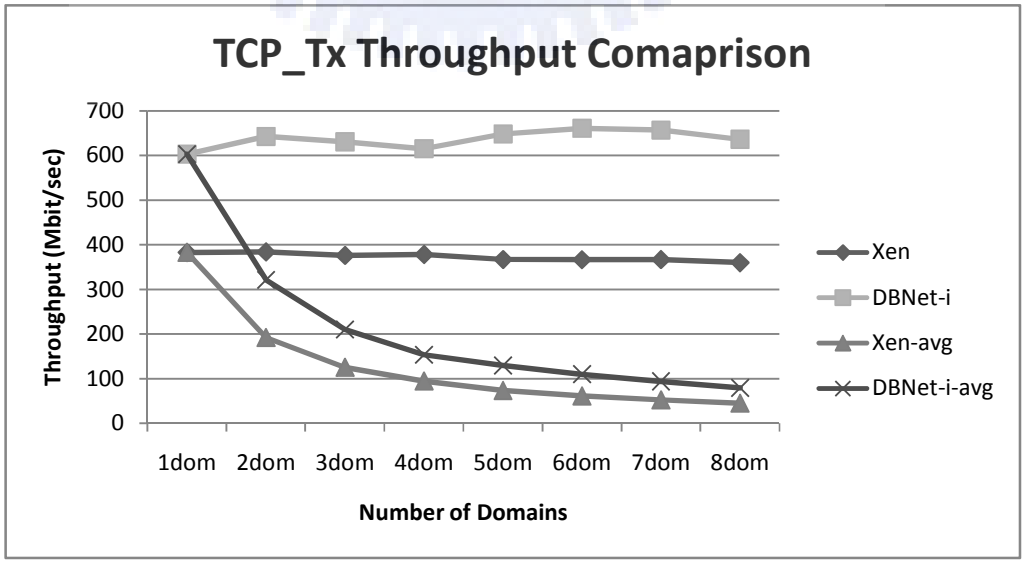

TCP_Tx Throughput Comaprison
Throughput (Mbit/sec)
700
600
500
400
300
200
100
0
1dom
2dom
3dom
4dom
5dom
6dom
7dom
8dom
Number of Domains
Xen
DBNet-i
Xen-avg
DBNet-i-avg

Figure 10. TCP_Tx throughput comparison between Xen and DBNet-i

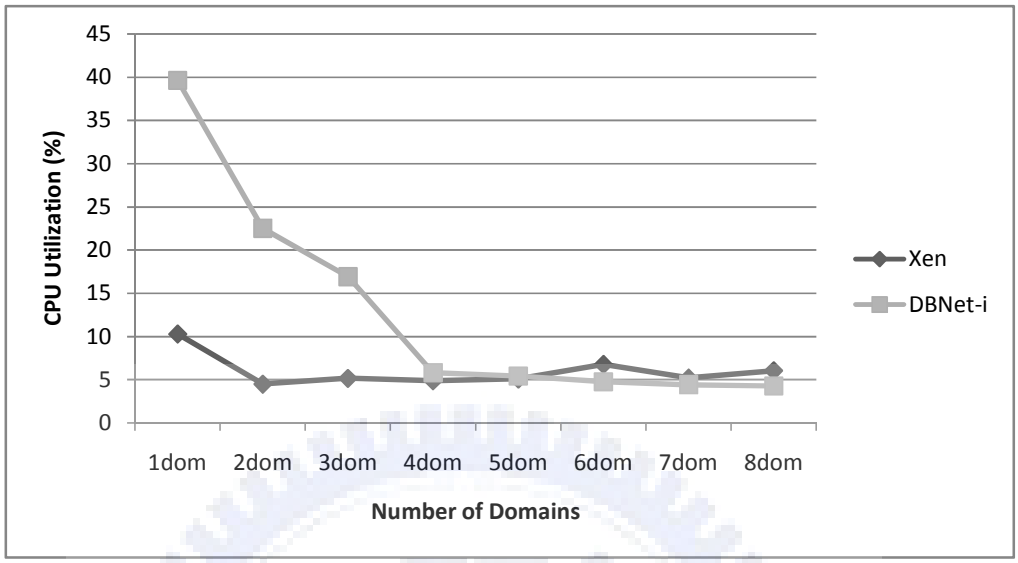
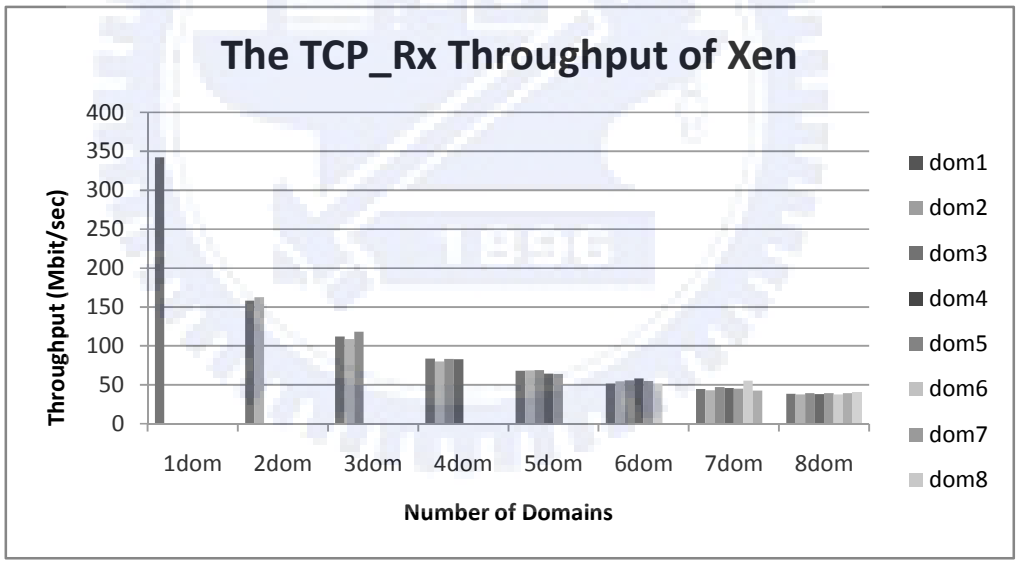
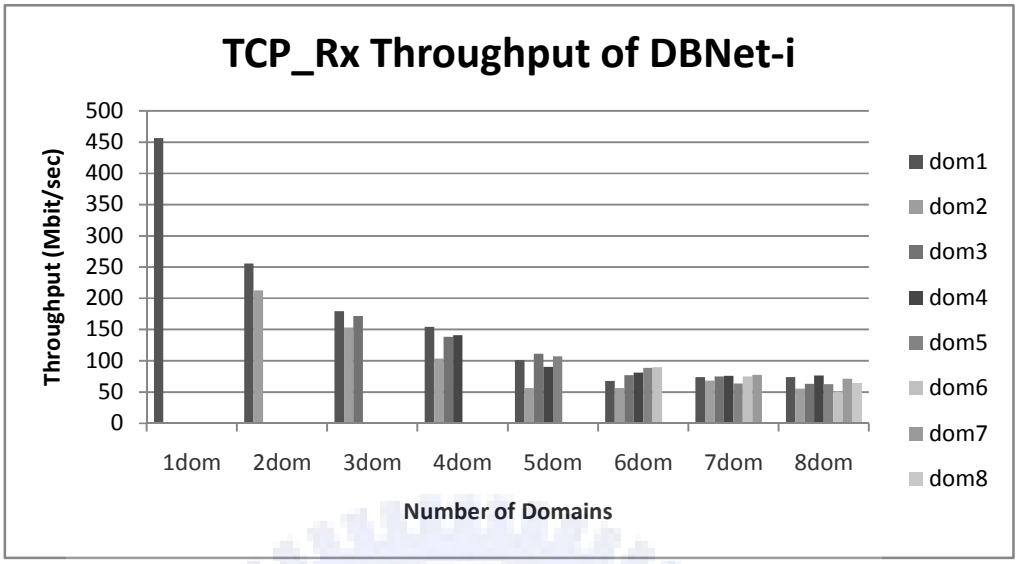


Figure 11. CPU utilization Comparison of TCP_Tx test between Xen and DBNet-i





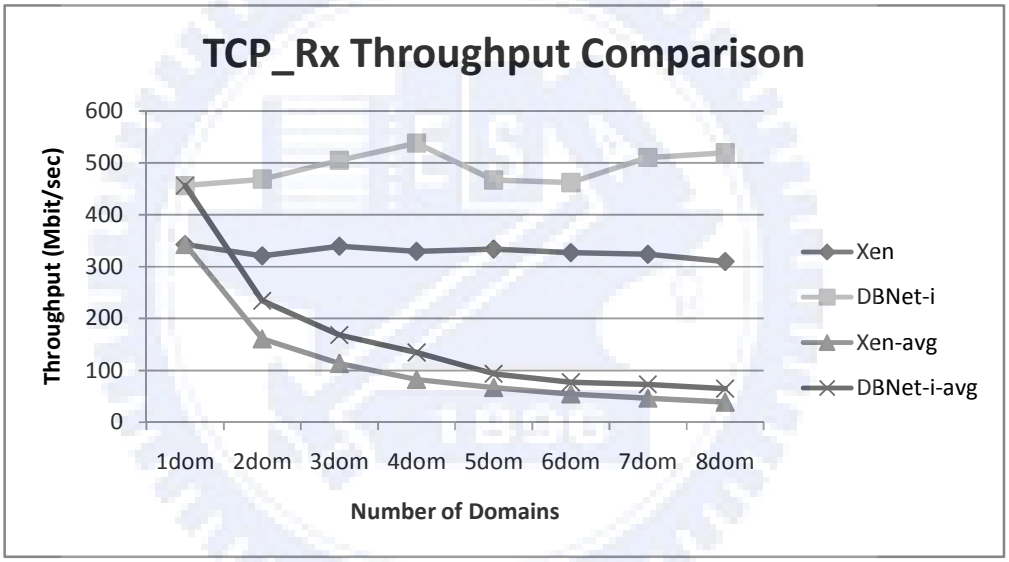


Figure 12. TCP_Rx throughput comparison between Xen and DBNet-i

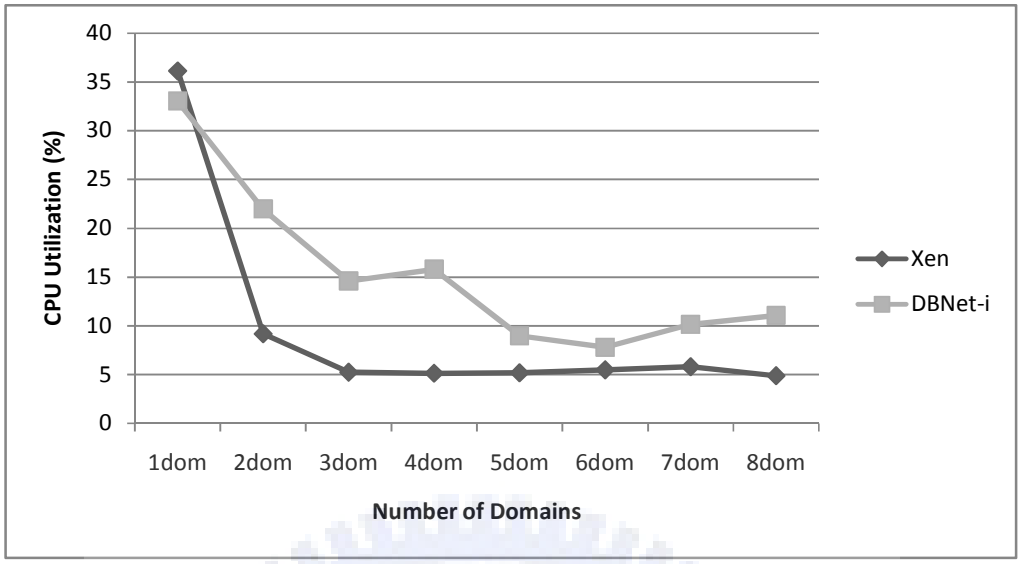


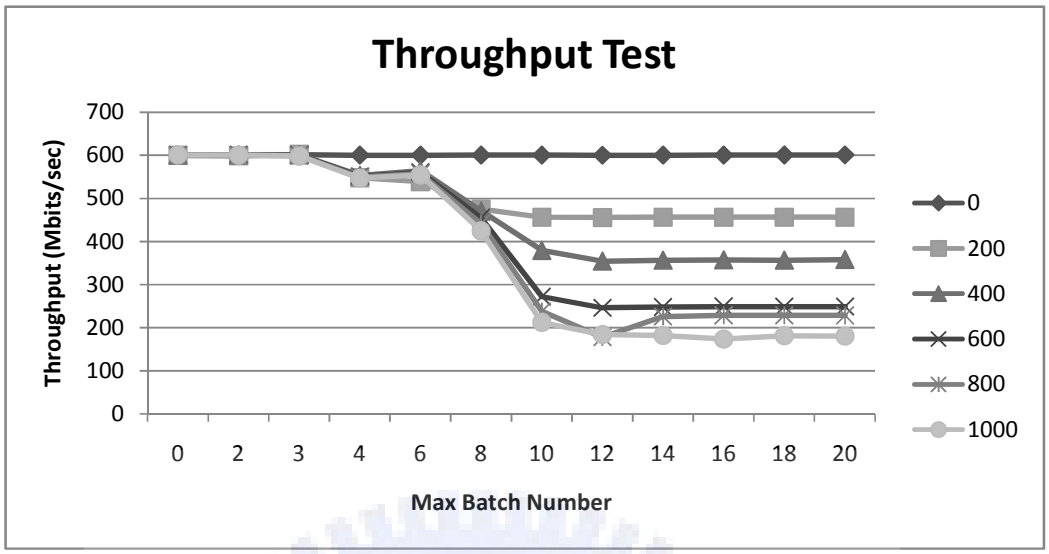
Figure 13. CPU utilization Comparison of TCP_Rx test between Xen and DBNet-i

4.3. Rx Batch Evaluation

In this section we adjust the batch amount of incoming packets to measure the performance in one-guest-domain environment on DBNet-i architecture. Xen uses the event channel mechanism which is similar to the signal mechanism to simulate virtual interrupts. However, event channel will cause lots of overhead and the VMMBE driver has to use it to notify a guest domain to receive an incoming packet. Therefore we have to avoid sending too many virtual interrupts through the event channel. Instead of notifying the guest domain per packet, we batch packets which belong to the same target and then notify the target guest domain to receive a batch of incoming packets. In order to accumulate the incoming packets, we set a max batch number. The VMMBE driver will not notify a guest domain unless the number of incoming packets reaches the max batch number. Besides, we set a max delay time to prevent incoming packets be batched too long. We adjust the max batch number and the max delay time to measure the performance in one-guest-domain environment on DBNet-i architecture. Figure 8 shows the CPU utilization and throughput of TCP_Tx test. The x-axis represents the max batch number and each line represents the max delay time (microsecond). From the figure, if we batch more packets and delay for a more long time, the throughput will

decrease substantially, which reduce the CPU utilization. This is because in TCP_Tx test, the receiver sends a packet periodically to notify the transmitter that it can continue to transmit packets. If we batch an incoming packet too long, the transmitter has to wait for receiving a packet which is sent from the receiver and thus decrease the throughput. However, if we do not batch any incoming packet, the throughput will reach 600 MBits/sec while use 45% CPU time. We found that when the pair of parameter (max delay time, max batch number) is (200, 3), we can also reach 600 MBits/sec throughput and only use 39% CPU. Therefore we choose this pair of parameter as our default value in all TCP_Tx tests to prove that DBNet-i architecture can reach almost the same throughput with native Linux in packet transmission while only use a little higher CPU time than native Linux.

Figure 9 shows the CPU utilization and throughput of TCP_Rx test. From the figure, if we do not batch any incoming packet, the throughput can reach 540 Mbit/sec but use 80% CPU time which is caused by using event channel too frequently. When the max delay time is 500, incoming packets can be batched effectively, which decrease 20% CPU time and still maintain 540 Mbits/sec throughput. When the pair of parameter is (1000, 25), DBNet-i can reach 450 Mbits/sec throughput and only use 33% CPU time. In original Xen architecture, it can only reach 340 Mbits/sec throughput while use 36% CPU time. Therefore, we choose this pair of parameter as our default value in all TCP_Rx tests to prove that DBNet-i architecture can reach higher throughput in packet reception while use less CPU time than Xen.



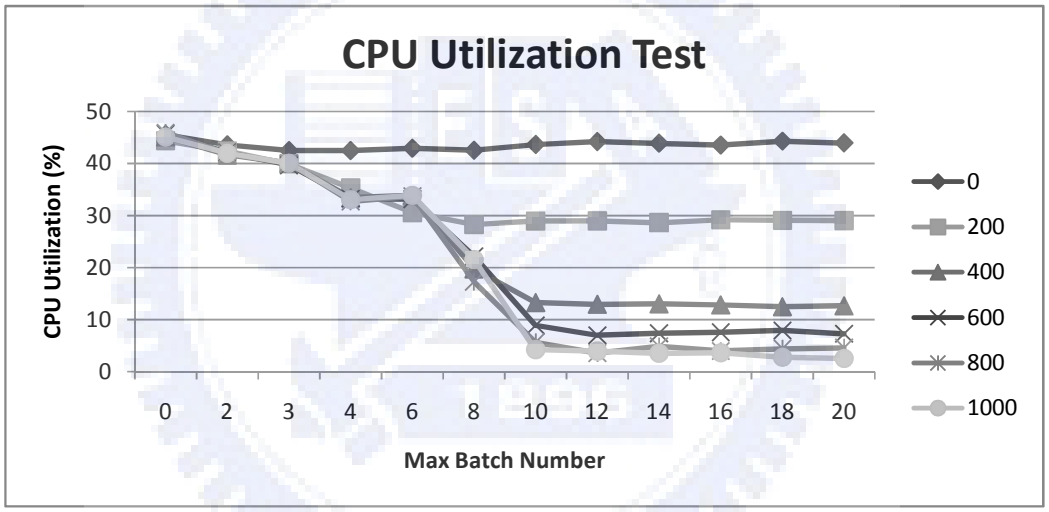
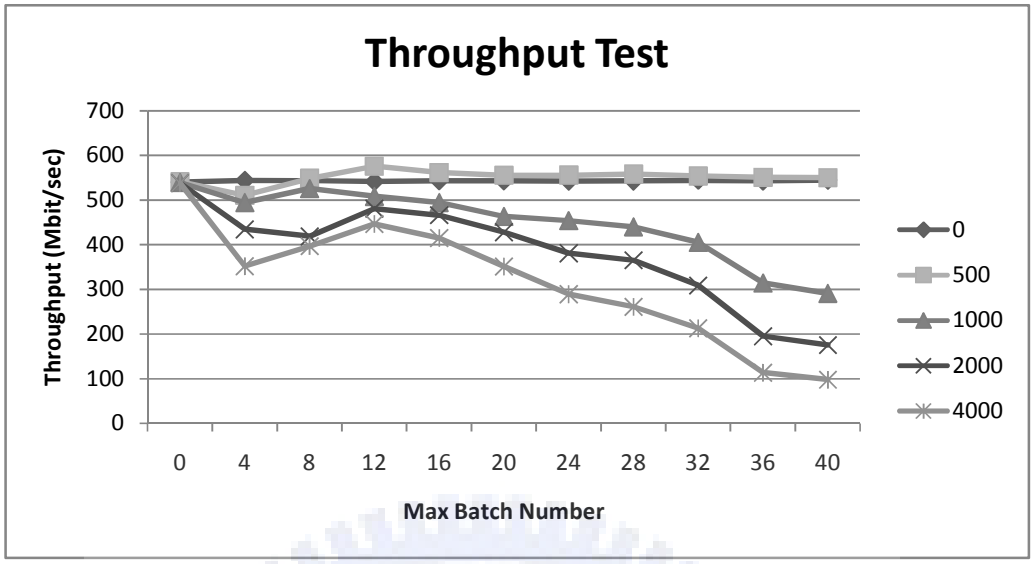


Figure 14. The impact of rx batch on TCP_Tx test



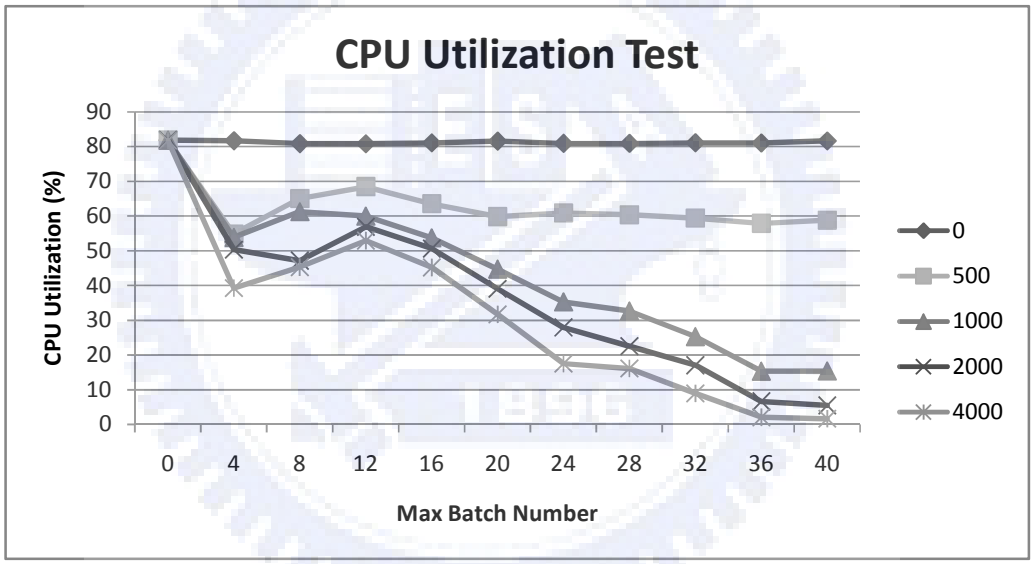


Figure 15. The impact of rx batch on TCP_Rx test

Chapter 5. Conclusion

In this thesis, we present a DBNet architecture, which allows a guest domain accessing the network without going through the driver domain. In order to avoid complicating the VMM, we only migrate the performance-critical part of the NIC driver from the driver domain into the VMM. Moreover, we implement a simplified bridge in the VMM to allow the VMM dispatching a packet correctly to a target guest domain or a remote machine and avoiding complicating the VMM. Finally, we put the VMMNIC driver into an independent driver segment and lower its privilege level to protect the VMM form crashing by driver faults.

Our performance measure shows that in the one-guest-domain environment, the DBNet architecture can achieve nearly the same performance with Linux in TCP transmission test, and can improve 24% network throughput while use less CPU utilization in TCP reception test. Beside, DBNet architecture can also achieve better performance for each guest virtual machine than the original Xen network architecture in the multi-guest-domain environment.

Reference

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt and A. Warfeld, "Xen and the Art of Virtualization", Proceedings of the ACM Symposium on Operating Systems Principles, pp. 164-177 Oct. 2003.

[2] C. Benvenuti, Understanding Linux Network Internals, O'Reilly Media, Dec. 2005

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic and W. K. Su, "Myrinet: A Gigabit-per-Second Local Area Network", IEEE Micro, pp. 29-36, Feb. 1995

[4] D. Bovet and M. Cesati, Understanding the Linux 核心, 3nd Edition, O'Reilly Media, Nov. 2005

[5] T. C. Chiueh, G. Venkitachalam and P. Pradhan, "Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions", Proceedings of the17th ACM Symposium on Operating Systems Principles, pp. 140-153, Dec. 1999.

[6] A. Chou, J. Yang, B. Chelf, S. Hallem and D. Engler, "An Empirical Study of Operating Systems Errors", Proceedings of the 18th ACM Symposium on Operating Systems Principles, pp. 73-88, Oct. 2001.

[7] P. Chubb, "Get More Device Drivers out of the 核心!", Ottawa Linux Symposium, vol.1:149-161, Jul. 2004.

[8] A. Forin, D. Golub and B. Bershad, "An I/O System for Mach 3.0", Proceedings of theUSENIX Mach Symposium, pp. 163-176, Apr. 1991.

[9] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield and M. Williamson, "Safe Hardware Access with the Xen Virtual Machine Monitor", Proceedings of the 1th Workshop on Operating System and Architectural Support for the on Demand IT Infrastructure, Oct. 2004.

[10] B. Hausauer, "iWARP Ethernet: Eliminating Overhead In Data Center Designs" Apr.

2006

[11] G. C. Hunt, "Creating user-mode device drivers with a proxy", Proceedings of the 1st USENIX Windows NT Workshop, pp. 55-59, Aug. 1997.

[12] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber and B. Zill. "An Overview of the Singularity Project", Microsoft Research Technical Report MSR-TR-2005-135, Microsoft Corporation, Redmond, WA, Oct.2005.

[13] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide

[14] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone and G. Heiser, "User-level Device Drivers: Achieved Performance", Journal of Computer Science and Technology, 20(5):654-664, Sep. 2005

[15] J. LeVasseur, V. Uhlig, J. Stoess and S. Gotz, "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines", Proceedings of the 6th Symposium on Operating Systems Design and Implementation, pp. 17-30, Dec. 2004.

[16] J. Liu, W. Huang, B. Abali and D. K. Panda, "High Performance VMM-Bypass I/O in Virtual Machines" Proceedings of the USENIX 2006 Annual Technical Conference, pp. 15-28, May. 2006.

[17] K. V. Maren, "The Fluke Device Driver Framework", University of Utah, Master's Thesis, Dec. 1999.

[18] A. Menon, A. Cox and W. Zwaenepoel "Optimizing Network Virtualization in Xen", Proceedings of the USENIX 2006 Annual Technical Conference, pp. 15-28, May. 2006.

[19] A. Menon, J. Santos, Y. Turner, G. Janakiraman and W. Zwenepoel, "Diagnosing Performance Overheads in the Xen Virtual Machine Environment, In Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, pp.13-23, Jun. 2005.

[20] Microsoft Corporation, "Introduction to the WDF User-Mode Driver Framework", Apr. 2005.

[21] A. Rubini and J. Corbet, "Linux Device Drivers", 3nd Edition, O'reilly Media, Feb. 2005.

[22] P. Shivam, P. Wyckoff and D. Panda, "EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing", Supercomputing, ACM/IEEE Conference, pp. 49-49, Nov. 2001

[23] R. Short, “Vice President of Windows Core Technology”, Microsoft Corp. Private Communication, Apr. 2003

[24] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt and S. Levi, "Solving the Starting Problem: Device Drivers as Self-Describing Artifacts", Proceedings of the EuroSys conference, pp. 45-57, April. 2006

[25] M. M. Swift, B. N. Bershad and H. M. Levy, "Improving the Reliability of Commodity Operating Systems", Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 207-222, Oct. 2003.

[26] M. M. Swift, S. Martin, H. M. Leyand and S.J. Eggers, "Nooks: An Architecture for Reliable Device Drivers", Proceedings of the 10th ACM SIGOPS European Workshop, Sep. 2002.

[27] H. Vemuri, D. Gupta and R. Moona, "Userdev: A Framework for User Level Device Drivers in Linux", Proceedings of the 5th NordU/USENIX Conference, Feb. 2003.

[28] Virtual PC: http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx

[29] VMware: http://www.vmware.com/

[30] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox and W. Zwaenepoel, “Concurrent Direct Network Access for Virtual Machine Monitors”, High Performance Computer Architecture, HPCA, IEEE 13th International Symposium on, pp. 306-317, Feb. 2007

[31] XenSource: http://www.xensource.com/

[32] V. Ganapathy, A. Balakrishnan, M. M. Swift and S. Jha, “Microdrivers: A New Architecture for Device Drivers”, Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS XI), May 2007

[33] Windriver: http://www.windriver.com/