2008 International Conference on Computer Science and Software Engineering

The Optimization of Xen Network Virtualization

ZHANG Jian[1], LI Xiaoyong[2], GUAN Haibing[1*]
[1]Department of Computer Science and Engineering, [2] School of Information Security Engineering
Shanghai Jiao Tong University
Shanghai 200240, China
jianzh@sjtu.edu.cn, xiaoyongli@sjtu.edu.cn, hbguan@sjtu.edu.cn

Abstract—**Despite the benefits brought by virtualization technology, the network I/O performance degradation remains as a barrier for its wide usage. This paper presents the design and implementation of a new method to improve Xen network virtualization performance by optimizing the interrupt deliver route and shortening the network I/O path. With the above optimization techniques, network throughput of a HVM guest domain is improved by 50%, CPU utility of QEMU driver model is reduced by 70%, TLB miss and cache miss is improved by 40% to 80%. The rationale behind our optimization model can also be extended for other I/O device virtualization in HVM guest domain.**

Keywords-Virtual machine; Xen; Network Virtualization optimization, Ne2000

I. INTRODUCTION

The concept of virtual machine was introduced by IBM in 1960s, which is a virtual layer introduced between software layer and hardware layer. Virtualization technology can provide an isolated execution environment to applications, shield the dynamics and heterogeneous of hardware platform, and support share and reuse of hardware resources.

As there is a growing trends to use virtual machine for server consolidation and system security enhancement in distributed computing environment, virtualization technology has attracted much interest in recently days[2][3]. Although cost reduction and simplicity of management and administration are the prominent advantages of virtualization technology, the overhead is a big obstacle to its wider applications. It is well known that compared with computing intensive applications, I/O intensive applications, especially network intensive applications suffer more in virtual execution environment. For instance, Menon et al. [6]reported significantly lower network performance under a Linux 2.6.10 guest domain compared to native Linux performance: degraded by a factor of 2 to 3x when receive data, 5x when transmit data. Our work also shows that throughput reduces by a factor of 4x in a Windows XP Hardware Virtual Machine (HVM) guest domain.

Based on research of the Xen network virtualization architecture and our own observations, we think the major part of overhead in Xen's network performance degradation was caused by the long data transfer path as well as the repeated VM EXIT(see section 2.3) that leads to trap in Xen hypervisor in a HVM domain.

In this paper, the design and implementation of a new method is presented to improve the performance of Ne2000 virtual NIC network virtualization in a HVM guest domain. By shorten the data transfer path and optimize the interrupt deliver route, the network virtualization performance was improved. Besides, the proposed optimization method can also be applied for other I/O virtualization because most I/O device works the same way as NIC.

The key contributions of this paper are as follow:

First, the motivation and key issues involved in the optimization of virtual machine network virtualization are described.

Second, a new model to optimize Xen network virtualization is proposed and implemented which only needs little modification to Xen but no modification to Guest OS.

Third, a detail performance evaluation of the optimization model is given.

Table 1 summarizes the overall throughput performance of the optimization model. While using winscp to copy a 100MB file, the throughput improved 54%.

TABLE I. OVERALL THROUGHPUT PERFORMANCE

	Average Throughput
Native performance	2.13 Mbps
Ne2000 original	1.12 Mbps
Ne2000 optimized	1.72 Mbps

The remainder section of this paper is organized as follows: Section 2 presents the motivation towards virtualization optimization and how Xen works based on Intel-VT technology. Section 3 describes key aspects of our design and implementation. Section 4 introduces an evaluation method. The experiment results demonstrate the effectiveness of our optimization model. Section 5 reviews the related work and Section 6 draws conclusions and discusses about the future work.

II. BACKGROUND

This section presents the background information for our work: Xen and Intel VT technology. To understand the Xen network virtualization overheads better, we also present how Xen network virtualization works.

*Corresponding author address: Guan Haibing, professor, Department of Computer Science and Engineering, Shanghai Jiao Tong University, 800 DongChuan Road, Shanghai, 200240, China

A. Xen

Xen is an open source virtual machine monitor (VMM) brought by Cambridge university, it allows many (up to 100) different guest operating systems running on the same physical machine [4]. Generally, there are two kinds of technology to realize virtualization, full virtualization and para-virtualization. VMware and Virtual PC use full virtualization and thus they don't need any modification to OS. Xen and Denali [9]use para-virtualization technology which provides the virtual machine an abstraction that is different from the underlying hardware and needs patch for the kernel[3]. As guest OS must be ported to the hypervisor, para-virtualization performance was much better than full virtualization [6]. Xen support full virtualized virtual machine from version 3.0 with the assist of hardware virtualization technology such as Intel VT and AMD Pacifica. As Xen does not change the application binary interface (ABI), user level applications don not need any modification to run in Xen and thus keeps the application compatibility.

Fig.1 illustrates the architecture of Xen hypervisor [4]. Unlike hosted VMM model which hypervisor running on top of OS, Xen is running directly on hardware. In Xen, the terminology domain refers to virtual machine, all domains runs on Xen hypervisor. There is a privilege domain named domain0, which can creates, destroys other domains, it also controls the schedule parameter, memory allocation, disk access and other operations. Two mechanisms are designed for communication between Xen and guest domain: synchronous hypercall for domains to call Xen and asynchronous event-channel for Xen to deliver notifications to domains.

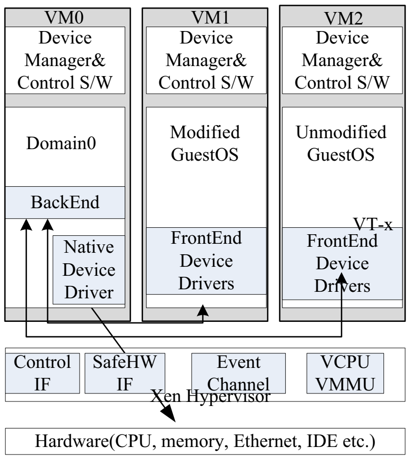

Figure 1. Xen hypervisor 3.0 architecture

B. VT technology in Xen

Our work adopts full virtualized guest virtual machines, which needs the support of hardware virtualization technology. In 2005, Intel released its hardware virtualization technology, a serial of processor technologies that can support unmodified OS running on Intel-VT enhanced VMMs[8].

As Fig.2 described, VT-x supplies two new CPU operation environments: virtual machine extensions (VMX) root operation and VMX non-root operation. VMM runs in VMX root operation while guest runs in VMX non-root operation. The transfer from VMX non-root operation to root operation is called VM Entry and the opposite direction is called VM Exit.

The virtual machine control structure (VMCS) is defined to manage VM Entry and VM Exit. It includes guest-state area and host-state area. When guest OS executes privilege instructions, it will cause VM Exit, Xen will save the processor state to guest-state area and load processor state from the host-state area. VM entry will do the opposite job. Xen can handle some of VM Exits directly; however, most of them must be handled by Domain0.

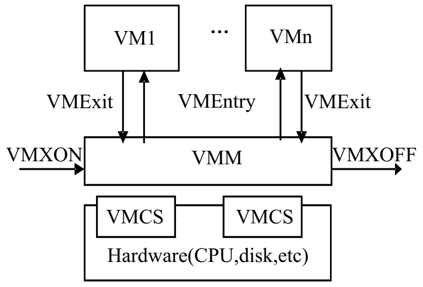

Figure 2. Architecture of Intel VT-x

III. XEN NETWORK I/O ARCHITECTURE

Xen uses a spit driver model, among all domains running on Xen, only domain0 has direct access to physical I/O devices. It performs I/O operations on behalf of the other domain. As shown in Fig.1, in order to access device, guest OS sends request to the Frontend and Frontend will transfer request to the corresponding Backend. In this way, each data transmit or receive operation must go through domain0, which makes the network I/O virtualization architecture in Xen a bottleneck for networking performance.

A. Data transfer in HVM domain

The performance is even worse for a HVM domain without Front driver like windows. Fig.3 shows the network send packet flow in a HVM domain:

1) When a guest domain, named as domainU, sends data packet, the In/Out instruction will trigger VM Exit, control will transmit from guest domain to Xen. A function will be called to handle VM Exit. If this function can handle it directly, the process will be finished immediately.

2) Xen writes the detail information of the In/Out instruction to a shared page between Domain0 and DomainU, and then notifies domain0 via event-channel. After this, Xen will block DomainU and schedule other domains to run.

3) Xen restores domain0 and transfers control to domain0.

4) The first function on domain0 to be called is the callback function *hypervisor_callback*, which will call *evtchn_do_upcall* to collects I/O request from domainU.

5) *evtchn_do_upcall* will trigger the select system call in switch and then call I/O request handle function *cpu_handle_ioreq*. The latter will call *cpu_get_ioreq* to get the I/O request information in the shared page.

6) DM figures out what kind device the request wants to access, and calls the corresponding callback function registered when the device was initialized to process the request.

7) Those call back function were executed to send/receive data through the physical driver in domain0.

8) When the data transfer finished, DM will notify Xen that the data transfer has been completed. Xen will unblock domainU when it gets the notification, and then domainU can run again.

B. *Overheads*

In the above flow, if a HVM guest domain wants to send a packet, it has to write the command register and other registers that are emulated by QEMU first. Thus there will be a switch from HVM guest domain to Xen hypervisor and Xen hypervisor to Domain0. Those switches need to save and load CPU state, memory information, and etc at high cost.

This model extends the data transfer path and leads to two more switch between domain0 and Xen hypervisor, and causes a high overheads. This is the reason for the significant overhead for network I/O performance as described in section 1 and also the motivation to optimize the Xen network virtualization.

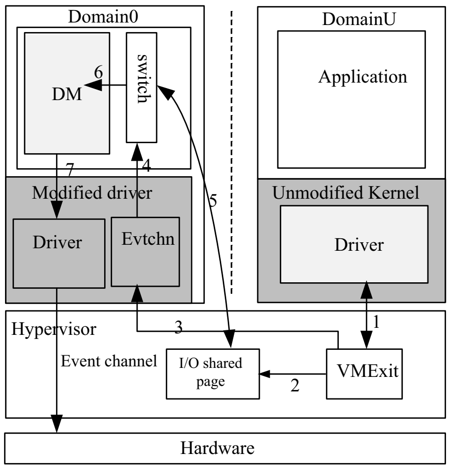

Figure 3. Xen network virtualization I/O process flow

IV. Design and Implementation

The optimization model was implemented for Xen 3.0.4 with Linux kernel 2.6.16.33. Design issues and how the optimization model works will be proposed in this section.

A. *Design issues*

Because how guest domain I/O request is sent to/from each physical NIC has great impact on the performance, one way to improve the network performance is to optimize the interrupt deliver route and shorten the data transfer path. Xen provides many virtual NICs to guest OS, such as Pcnet, Rtl8139 and Ne2000. We take Ne2000 for our optimization because its performance is quite poor and its structure is relatively simple. Our test shows that, in a Windows XP HVM guest domain, the native network throughput is 94.43Mbps while the Ne2000 throughput is only 18.84Mbps.

The optimization of the network virtualization performance of an HVM guest domain cannot make any modification to the guest operating system. Thus, in our design, we add a driver to the HVM guest domain to handle the requests and reroute the interrupt and requests.

It is known that the manipulations to NIC are realized by operations to its registers, including control registers and state registers. Among those operations, only a few write operations to control registers will trigger NIC hardware operations. Other operations such as read/write operations to state registers and read operation to control registers will not trigger NIC hardware operations. Taking this into consideration, we design and implement a driver in the hypervisor to handle those operations that will not trigger NIC hardware action and operations that Xen can process directly. Since the proposed driver shortens the data transfer path and reduces the switch between Xen and guest domain, an improvement on the performance of the Ne2000 virtual NIC can be achieved.

B. *Architecture Overview*

Shared I/O page data structure in Ioreq.h is used to store the information of I/O requests and the process results of those requests. Both Xen and domain0 have direct access to this shared I/O page. The following optimizations were based on this feature:

First, move *Ne200State* data structure that describes the state information about the Ne2000 virtual NIC from QEMU to the shared I/O page. Hence both Xen hypervisor and Domain0 can access it directly.

Second, change the original way to access the Ne2000 virtual NIC and move the functions that read registers and write state registers to Xen hypervisor. However, those functions that handle operations to control registers haven't been changed since they need the help of domain0.

Third, add a simple switch in Xen hypervisor to judge whether our optimization driver can handle the requests from a HVM guest domain. If the driver cannot handle it, the request will be processed in the original way.

Fig.4 illustrates the Xen hypervisor architecture after optimization. The overall optimization model changes only a little part of Ne2000 virtual NIC and adds a small patch to the Xen hypervisor. It does not affect the reliability and performance of Xen hypervisor. By moving the state information of Ne2000 virtual NIC to I/O shared page, a lot of register read operations and state register I/O requests can be processed in Xen directly. This model reduces the switches between Xen and domain0, shortens the data transfer path, and improves the Ne2000 virtual NIC performance.

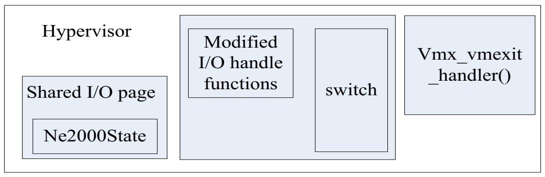

Figure 4. optimization architecture overview

C. Implementation

To implement the optimization model, first move the Ne2000State data structure, then redefine the initialization function and at last, redefine the I/O handle functions.

1) NE2000State: Move NE2000State data structure from Ne2000.c to *shared_iopage_t* data structure in ioreq.h. Thus both domain0 and Xen hypervisor have access to it. It means that we move the data structure in QEMU to Xen hypervisor. To add NE2000State as a member to vcpu_iodata is relatively simple:

```
struct vcpu_iodata {
    struct ioreq      vp_ioreq;    /* Event channel port */
    unsigned int   vp_eport;   /* vcpu uses it to notify DM */
    NE2000State ne_shared;  /*added to vcpu_iodata*/
};
```

2) NE2000 initialization: The initialization function *pci_Ne2000_init* is modified to get the Ne2000 state information contained in *vcpu_iodata*, in the following way:

```
NE2000State *share=&(shared_page->vcpu_iodata[send
_vcpu].ne_shared);
```

When the state information from the shared I/O page is obtained, we will call *pci_register_device* to register the virtual device and *pci_register_io_region* to register the corresponding read and write functions.

3) Switch and I/O operation functions: Now both domain0 and Xen hypervisor can manipulates Ne2000 NIC registers. The next step is to move those functions which can process the requests without domain0's assistance to Xen hypervisor. We mainly ported *Ne2000_ioport_write* and *Ne2000_ioport_write* to Xen hypervisor. As not all of the requests can be handled by our modified functions, a switch is needed to judge whether the request can be handled and those requests that cannot be handled will be sent to domain0 to process it as usual.

D. Data transfer flow

To further understand how the optimization model works and make a comparison between it and the original model, Fig.5 is referred. It describes the work flow of the optimization model. After moving those functions which don't need the help of domain0 to Xen hypervisor, the original data transmit path was shortened as Xen will handle it directly and did not need to send interrupt to domain0 and switch to it. This leads to performance improvement as the experiment results demonstrate in section 4.

Taking data transfer as an example, the optimization model works in the following way:

1) When the HVM guest domain sends data packet to other machine, it will write the control register or state register of Ne2000 virtual NIC. IN/OUT instruction will cause VM Exit and traps in Xen.

2) Xen gets control of CPU and calls function *vmx_vmexit_handle*, which will read reasons of VM Exit from VMCS structure. The hypervisor will get the detail information about the I/O operation, such as I/O type, I/O port address, data length, etc. *vmx_vmexit_handle* will call *vmx_io_instruction* to handle I/O operations, which will invoke *send_pio_req* function later.

3) Send_pio_req will analyze the I/O request information and call our switch to judge whether this request can be handled by our model.

4) Described as the dashed line in Fig.5, if the request can be handled in Xen, the request will be sent to the modified Ne2000 I/O functions immediately, and the results will be written to the shared I/O page. As soon as this process finished, it will notify Xen hypervisor. It is observed in Fig.5 that the data will flow in the dashed line. Hence the data transfer path is greatly shortened compared with the original model.

5) If this request cannot be handled by Xen alone, the switch places the I/O request in the shared I/O page and notifies domain0 to handle it by the original way shown in Fig.3.

Compared Fig.3 with Fig.5, our optimization model greatly shortens the data transfer path by intercepting I/O requests and handling it immediately. In this way we reduced the switch between the Xen hypervisor and domain0, which is the major reason for the high performance degradation, and thus improved the performance of Xen network virtualization.

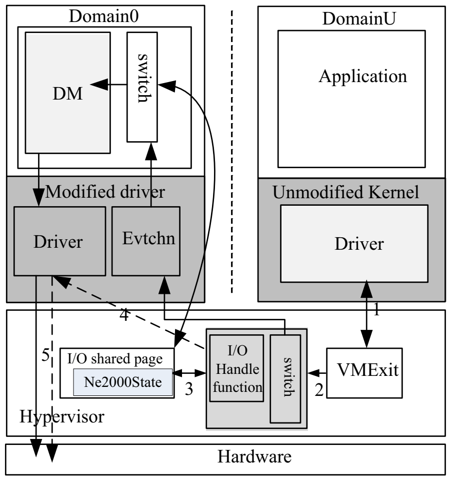

Figure 5. Data Transfer flow in the optimized model

TABLE II. XENOPROF ITLB, DTLB, AND CACHE MISS RESULTS

Module	Function	Original Ne2000 Virtual NIC			Optimized Ne2000 Virtual NIC		
		ITLB miss %	*DTLB miss %*	*L1 cache miss %*	*ITLB miss %*	*DTLB miss %*	*L1 cache miss %*
qemu-dm	cpu_handle_ioreq	0.5474	1.5288	1.6616	0.9943	0.8062	1.3487
vmlinux-syms -2.6.16.33-xen	evtchn_do_upcall	0.6316	0.3745	0.6155	0.1420	0.6155	0.6316
qemu-dm	Ne2000_ioport_write	0.3789	2.0072	0.2256	0.1420	0.0258	0.2117
qemu-dm	main_loop_wait	0.2947	3.9097	1.0683	0.1420	2.638	1.4154
qemu-dm	Ne2000_ioport_read	2.9830	0.1231	-	0.5474	0.1369	-

V. EXPERIMENTAL RESULTS AND EVALUATION

The optimization was implemented on Xen 3.0.4, running a Linux kernel 2.6.16.33. We use two testing machines, machine A with Intel Core^{TM}2 Duo 1.86GHz processor with VT-x support, 4MB L2 cache, 1GB memory; machine B with Intel Pentium D 3.00GHz processor, 2MB L2 cache, 1GB memory. We configured two guest VMs on machine A with 512MB memory which installed Windows XP sp2 operating system. VM1 uses the original Ne2000 virtual NIC and VM2 uses our optimized virtual NIC.

A. *Evaluation methodology*

To evaluate the performance of the optimization, the experiment compares the throughput, TLB and cache miss, CPU overheads and VM Exit between the original network model and the optimized model.

For the testing benchmarks, three different benchmarks are employed, CHARIOT to measure the throughput and response time, Xenoprof to measure the TLB and cache misses, Xentrace to measure the number of VM Exit, and Linux top is adopted to analyze CPU overheads.

B. *Throughput Performance and latency*

In the experiment, CHARIOT endpoints run on VM1, VM2 and machine B separately. VM1 and VM2 sends 2000 records with different record size to VM3 at the speed of 50 records per second. In such way, the throughput and latency is measured.

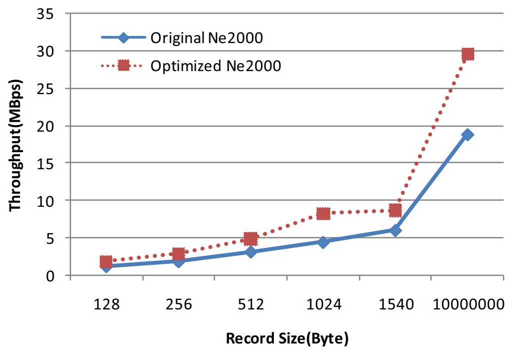

Figure 6. Throughput comparison results

Fig.6 shows the throughput results while sending records at different size between VMs and Machine B. As demonstrated in this figure, the optimization model improved the throughput by a factor range from 1.43 to 1.86 with the average of 1.56. The experiment also shows that when sending small records, the latency of the optimized virtual NIC is 0.001s, which is improved by 100% compared to the original 0.002s. When the size of the record increases to 10000000Bytes, the latency of our optimized virtual NIC is 2.713s, which is improved by a factor of 1.56 compared to the original 4.244s.

C. *TLB and Cache Miss*

We use Xenoprof[6]workload to evaluate the TLB and cache miss results of the optimization model. In the experiment, Xenoprof benchmark measures the TLB and cache miss when sending 10^9 data with Chariot from VM1 and VM2. The experiment shows that the TLB misses does not concentrate on several functions but scatters in domain0. This is quite similar to the conclusion in Menon's discovery [6]. In our configuration, we set the time counter to 10000, thus Xenoprof will sample every 10000 clock cycle.

In the experiment, in the original model, 1116 ITLB miss, 135661 DTLB miss and 45679 L1 cache miss happened; while in the optimized model, 704 ITLB miss, 25506 DTLB miss and 25506 L1 cache miss happened. These results show that the ITLB miss improves by 40% and the DTLB miss improves by 81%.

Table 2 shows the detail ITLB, DTLB and cache miss comparison (in percent) between the original Ne2000 virtual NIC and the optimized Ne2000 virtual NIC on most important function. It shows that the function call of *Ne2000_ioport_write* is dramatically reduced. In addition, as in the optimization Virtual NIC, most requests need not to deliver request to QEMU, the *evtchn_do_upcall*, which switch from Xen hypervisor to domain0, was reduced too.

D. *CPU Overheads and VM Exit*

Due to the increase of throughput, the data transfer time is greatly reduced, and the number of VM Exits during sending 10^9 Bytes record is also reduced. Experiment results show that in the original Ne2000 virtual NIC 30427338 VM Exits happened, while in the optimized Ne2000 virtual NIC only 26613669 VM Exits happened. The total number of VM Exits reduces 12.5%.

In order to evaluate the impact of our optimized model to CPU overheads, we use Linux Top to measure the CPU overheads of QEMU-dm. In the original Ne2000 virtual NIC, the CPU utility of qemu-dm is 68.7887% when sending the 10^9 Bytes records. In the optimized Ne2000 virtual NIC, the CPU utility is 18.459%. The CPU overheads reduced by 73%. The reasons of CPU overheads improvements lie in the shortening of I/O transfer path and the reduction of cache miss.

VI. Related Work

As performance becomes increasingly important to Virtual Machine Monitor (VMM), several previous studies work on the performance of Xen hypervisor to discover which part should accounts for the performance degradation. Menon et al, introduce a system-wide profiling tool Xenoprof for Xen which was ported from Oprofile [6]As described in section 1, Menon reported significant performance degradation of Xen network virtualization.

As is reported by [11], the Xen hypervisor and domain0 consume as much as 70% of the execution time during network transfers in Xen network virtualization architecture. Many studies have been carried out to address this performance problem. Aravind Menon et al [7]investigate three techniques to optimize network virtualization in Xen. They define a new network interface, optimize the implementation of I/O channel between domain0 and guest domain and provide support for the use of superpages in guest OS. It is reported that they improved transmit performance by a factor of 4.4 and receive performance by 35%. Unlike our research, their work was designed for para-virtualization guest domains. Paul Willmann et al [11]present hardware and software mechanism to enable concurrent direct network access by guest OS. With their architecture, they reduced CPU overheads, improved the transmit performance by a factor of 2.1 and receive performance by a factor of 3.3.

VII. Conclusion and future work

This paper presents a new implementation to optimize Xen HVM guest network virtualization by shortening the data transfer path with detail throughput, TLB and cache miss, CPU overheads experiment results. From the experimental results, it is demonstrated that the optimized model can improve the HVM guest network throughput by a factor of 1.56. The TLB and cache miss and CPU overheads also improved a lot in the new network virtualization model.

As I/O devices in Xen are all emulated by QEMU, the data transfer path and process mechanism is similar to Ne2000 virtual NIC. Based on this similarity, we think that the principle behind our optimization model applies to other virtual device as well.

The optimization model mainly reduces switch between domain0 and Xen hypervisor. The switch between domainU and Xen remains as a future work to be solved.

Acknowledgment

This work is supported by 863 Program of China under Grant No. 2006AA01Z169 and NSFC under grant No. 60503013. We are also grateful for other contributors from INTEL Corporation.

References

[1] Creasy R.J., the Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development,* l981, 25(5):483-490.

[2] R.P. Goldberg, "Survey of Virtual Machine Research," *Computer*, June 1974, pp. 34-45.

[3] Mendel Rosenblum, Tal Garfinkel, Virtual Machine Monitors: Current Technology and Future Trends, *Computer*, May 2005, pp. 34-42.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A.Warfield. Xen and the art of virtualization. In *19th ACMSymposium on Operating Systems Principles*, Oct 2003.

[5] Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Malick. Xen 3.0 and the art of virtualization. *In Proc. of the 2005 Ottawa Linux Symposium,* Ottawa, Canada, July 2005.

[6] Menon, J. R. Santos, Y. Turner, G. J.Janakiraman, and W. Zwaenepoel. *Diagnosing Performance Overheads in the Xen Virtual Machine Environment*. In First ACM/USENIX Conference on Virtual Execution Environments (VEE'05), June 2005.

[7] Aravind Menon, Alan L. Cox, Willy Zwaenepoel. Optimizing Network Virtualization in Xen. I*n Proceedings of the 2006 USENIX Annual Technical Conference*, Boston, MA, May 2006.

[8] R.Uhlig, G.Negier, and D.Rodgers, Intel virtualization technology*, IEEE Computer Volume 38, Issue5, pp.48-56, May 2005.*

[9] Whitaker, A., Shaw, M., Gribble, S. D. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*,2002, 195-209.

[10] K. Fraser, S. Hand, R. Neugebauer, I. Pratt,A. War_eld, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure* (OASIS), Oct 2004.

[11] Paul Willmann, Jeffrey Shafer, David Carr, et al. Concurrent Direct Network Access for Virtual Machine Monitors. *Proc of High Performance Computer Architecture*, 2007.