ACCOUNTING PER-VM RESOURCE USAGE FOR I/O ACTIVITIES IN VIRTUALIZED ENVIRONMENT

Youngjoo Woo

Computer Engineering

Graduate school of UNIST

Accounting Per-VM Resource Usage for I/O Activities in Virtualized Environment

A thesis submitted
to the Graduate School of UNIST
in partial fulfillment of the requirements
for the degree of Master of Science

Youngjoo Woo

7. 13. 2012 of submission
Approved by

Major Advisor
Young-ri Choi

Co-Advisor
Euiseong Seo

Accounting Per-VM Resource Usage for I/O Activities in Virtualized Environment

Youngjoo Woo

This certifies that the thesis of Youngjoo Woo is approved.

7. 13. 2012 of submission

Signature

Thesis Supervisor: Young-ri Choi

Signature

Euiseong Seo: Thesis committee Member #1

Signature

Beomseok Nam: Thesis committee Member #2

Abstract

Virtualization technology in Cloud computing environment offers efficient resource and power utilization through enabling multiple operating system instances to run concurrently within virtual machines on a single physical machine. Virtualization layers provide the billing system of clouds for business purpose. Current cloud computing systems focus processor utilization or number of virtual machines allocated by each user for billing. However, the billing system based on processor utilization may be unfair when several virtual machines that run on single physical machine have different workloads respectively. To improve fairness of billing and resource provisioning, the system has to consider about not only processer but also I/O resource utilization. The Xen virtual machine monitor follows a split device driver model to handle I/O requests from guest domains. Virtualized system has a driver domain, which performs I/O operations on behalf of guest domains and uses their native device to access I/O device directly. For this reason, the driver domain executes delegated instructions to processing I/O activities from guest domains. But the delegated instructions are not considered for scheduling domains or accounting. This paper presents the profiling technique of delegated processor usage to the driver domain for I/O operations per virtual machine in virtualized environment by Xen. And we introduce relationship between network utilization of guest domains and delegated processor utilization.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Cloud computing is to provide computing power through the network as a service. It facilitates to make efficient use of resources in data centers. So, the venders can provide services with less server equipment without decreasing quality of the service (Armbrust et al., 2009). Nowadays, data centers are regarded as a big problem in economic, and environmental aspects, because they waste huge amount of energy. But the number of required data centers is increasing (Kumar, 2007). For this reason, the cloud computing attracts attention of venders as one of the effective solutions which reduce total cost of ownership (Armbrust et al., 2010).

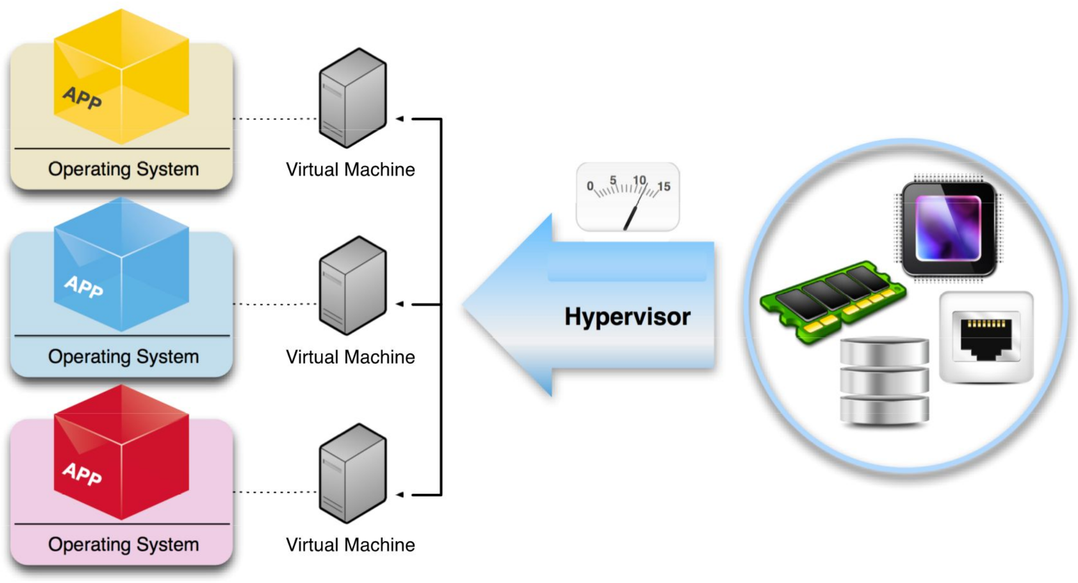


Figure 1.1: The Role of Virtualization

Virtualization is an important technology in cloud software stack, which takes on a great role for managing resources. The hypervisor running between hardware and guest operating systems enables that multiple virtual machines (VMs) run on a physical machine concurrently. And applications running on VMs are isolated from underlying hardware and other VMs by the hypervisor (Adams and Agesen, 2006, Barham et al., 2003, Nanda and Chiueh, 2005).

For example, when multiple server applications share a physical server without virtualization, then the server faces a number of problems like security, performance isolation, scalability, stability and so on. A defect of an application that is running on the server can affect entire system because those applications share a file system and hardware resources. The simplest way to avoid these problems is to assign a server machine per application. However, this method requires the great expense to maintain severs, and also it decreases usability.

Venders try to provide more services without degrading of stability or increase of the cost by using resources efficiently. Thus, the service providers may decide to assign each application to each VM like described in Figure 1.1. Although, the virtualization generates some overhead, but this solution makes the server more secure and safe with low additional cost.

Likewise, the virtualization technology intensifies stability of servers, flexibility of resource distribution, and scalability of providing services (Armbrust et al., 2009). Virtualization also allows adjusting amount of resources provided to users through the hypervisor (or virtual machine monitor). And additionally some hypervisors provide live migration (Kallahalla et al., 2004, Clark et al., 2005), which moves VMs into other physical servers on the fly depending of utilization physical servers. Therefore, it helps to observe SLA (service level agreement) (Buyya et al., 2009).

Cloud computing services can be classified as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) depending on depth of software stack that vendors provide for customers. Especially, the IaaS is the delivery of hardware and associated software. Amazon Elastic Compute Cloud (EC2)(Ostermann et al.) and Rackspace Hosting are the popular examples of IaaS and those services provide VMs to customers. VM instances have equivalent structure as physical machines, so users can organize software stack unconstrained from operating system to application level. Furthermore, users can purchase the VM instances and computation resources on a pay-as-you-go basis without any extra cost for maintaining their server.

Amazon EC2 is charging for VMs based on hours from time an instance is lunched until it is terminated. Whether the instance that consumes resources maximally or it is just on, the charge is equal for the same running hours. We saw this policy is not fair, considering the numerous factors that increase cost for providing the service.

The operating expenses of a data center consist of equipment fund and cost of maintenance. Spending on cooling and power consumption accounts for a great part of maintenance expenditures. By the way, the cost of maintenance varies with the characteristics of workloads. The cost of providing VM increases naturally with the amount of used resource by the VM, and it even depends on which resource is used (Kim et al., 2011).

However, it is not easy to measure resources used by each VM. Hardware cannot gauge the usage of a VM that is on the move with live migration or shares a host with other VMs. For this reason, profiling VM has to be performed by software (Kansal et al.). But also, virtualized environment has structural properties that make profiling real resource usage of a VM difficult.

Xen (Barham et al., 2003), one of the most commonly used hypervisors by cloud vendors, has adapted the safe hardware interface which allows the unmodified device drivers to be shared across isolated operating system instances. When Xen boots, it loads a privileged domain (or domain 0) that has responsibility for handling devices. So every I/O request from unprivileged domain (or domain U) is transferred to domain 0 (Fraser et al., 2004, Chisnall, 2008). Consequentially, the I/O activities from domain U consume CPU and memory resources on the domain 0; in addition, the usages are accounted as the share of domain 0. We see those resource usages have to be regarded as consumed by domain U that has produced the requests. But, current virtualization frameworks do not have any way to profile real resource usage of VMs, including I/O activities processed by domain 0.

Typically, a VM scheduler allocates CPU to VMs by time interval that depends of the scheduler algorithm and its quantum size. And the Credit Scheduler, default of Xen, targets a fair share CPU scheduler on a SMP host (Cherkasova et al., 2007). But the existence of domain 0 causes unfairness of CPU allocation, even if hypervisor uses a fair scheduler. The real CPU usage alters considerably following characteristic of workloads.

Accordingly, accounting per-VM resource usage for I/O activities on domain 0 should proceed to develop a fair billing system and VM scheduling. In this paper, we introduce a method to profile CPU time for network activities on domain 0 and the implementation in Xen. And also, we evaluated with various workload sets to show unfairness of current system.

The rest of this paper is organized as follows: Chapter 2 introduces background and related work; it includes how Xen supports I/O requests and a motivational experiment. Chapter 3 introduces implementation of our accounting feature. Chapter 4 introduces some result of accounting with our implementation. Chapter 5 introduces some related work. Finally, we conclude with discussion in Chapter 6.

Chapter 2.   Background and Motivation

2.1   Background

Xen hypervisor is a layer of software running directly on a physical machine replacing the operating system thereby allowing the physical machine to run multiple guest operating systems concurrently. It was first created at Cambridge University; thereafter Xen was open sourced to improve the product. The latest version of Xen hypervisor supports x86, x86-64, Itanium, Power PC, ARM and other CPU architecture, and simultaneously it supports a wide range of guest operating systems. Thus, Xen powers most public cloud services and many hosting services.

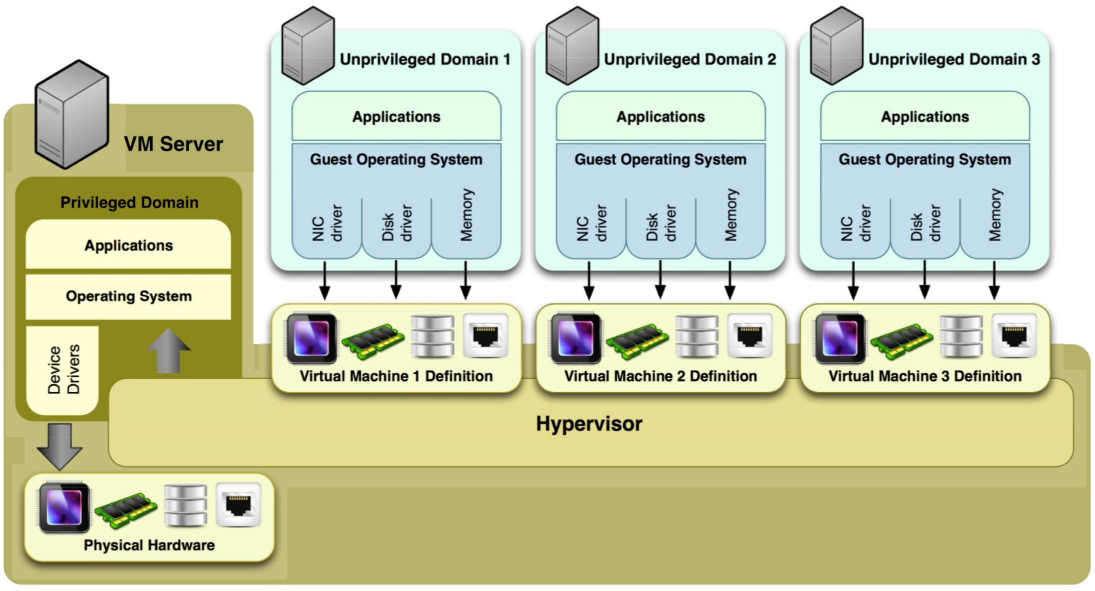


Figure 2.1: Virtual Machines on Xen Hypervisor

Figure 2.1 describes virtualized environment, namely a structure of a physical machine running multiple VMs with Xen. The term *domain* refers to a running virtual machine within which a guest OS executes. Domain 0 is the first guest to run and has charge of that handle devices and provides user interface. In contrast, other domains are referred to as domain U (or unprivileged domain).

To support the range of hardware available for commodity machine, Xen reuse native device drivers that already provided by the operating system in domain 0. Xen device drivers typically consist of the real driver, the back-end driver, the front-end driver, and the I/O ring. Figure 2.2 shows the path of a I/O data sent from a domain U and the composition of a split device driver (Chisnall, 2008).

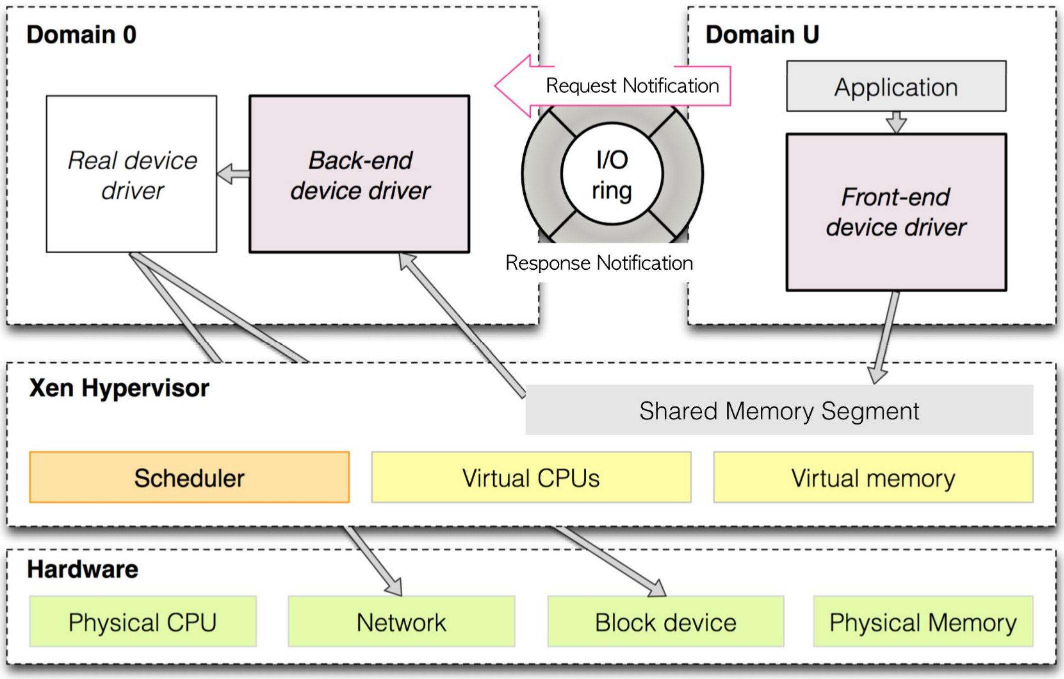


Figure 2.2: Xen I/O Architecture

The I/O ring provides a method for asynchronous communication between domains. It performs simple message-passing abstraction built on top of the shared memory mechanism. And the I/O ring is a kind of ring buffer that contains two types of data, requests and responses, updated by back-end and front-end driver. Each request and response is signaled to other domains through an event channel. Event channel is a notification mechanism between domains asynchronously (Barham et al., Fraser et al.).

Front-end driver running on domain U initializes a memory page with a ring data structure and exports it to domain 0. Back-end driver handles multiplexing to allow more than one VM to use the device and provides a generic interface for various operating systems. Actually, some features of a Linux kernel commonly used for domain 0 already provide service to multiplex access to devices. For example, access to hard disk is multiplexed using file system abstraction, network devices using a socket abstraction, and so on. For this reason, Xen driver use real drivers via those abstractions in existing operating system (Chisnall, 2008).

Watching the process of sending a packet from an application run on domain U to external node is a good example of how a Xen driver works. The packet travels through the TCP/IP stack normally, and the bottom of the stack, front-end driver, put it into shared memory. The back-end driver, running on the domain 0, reads the packet from the buffer and insert it into the exist networking stack of the operating system, which routes it as it would a packet coming from a real interface. Finally, it goes to the real device driver and the physical network device sends the packet.

2.2 Motivation

The network activities of domain U contain some process done by domain 0 guest. In other word, domain 0 uses processor and other resource for domain U guest. The structural characteristic may induce the resource distribution unfairness between domains.

If there are two domain U guests configured with same VM definition, which is running on same server. They have same value of priority for CPU scheduling, but different workload respectively. The Xen hypervisor employs Credit scheduler by default, so allocated

CPU time for each VM is proportionally equal. But one of them can use more resource except CPU depends on the workload.

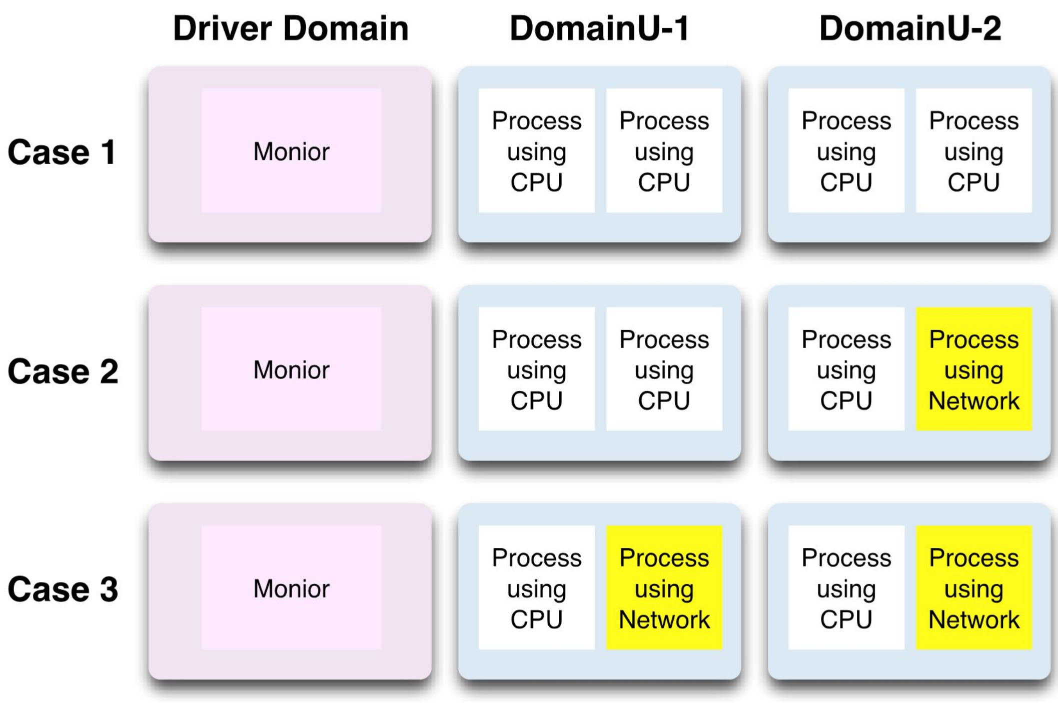


Figure 2.3: Various Workload sets used in the experiment

To make sure our supposition, we performed a simple experiment. We conducted a group of experiments on a Xen virtualized environment, which runs two domain U guests with exactly same definition and uses domain 0 to provide I/O features. Figure 2.3 describes three workload pairs for VMs. We measured average CPU time of domain 0 and two domain U guests for one second.

The CPU intensive processes were included in every workload pair and they performed floating-point operations without sleep. As a result, guests that run the process consumed allocated CPU time maximally. In the second case, CPU-Network workload, DomU(2) VM had Network intensive process. It sent fixed size of packets continuously. In the final case, both guest have network intensive process. In this experiment VMs could use only a portion of the network bandwidth because of the interference of other processes or other VM.

Figure 2.4 shows the result of this experiment. All unprivileged domains consumed same amount of CPU time, regardless of which workload was allocated. However, CPU usage of domain 0 was increasing depend on the number of VM that has network intensive process. In the second case, only DomU(2) used network. Thus, the excess CPU usage, comparing with the case that uses CPU only, had come from DomU(2). In conclusion, VMs that used network caused more CPU usage than those who did not.

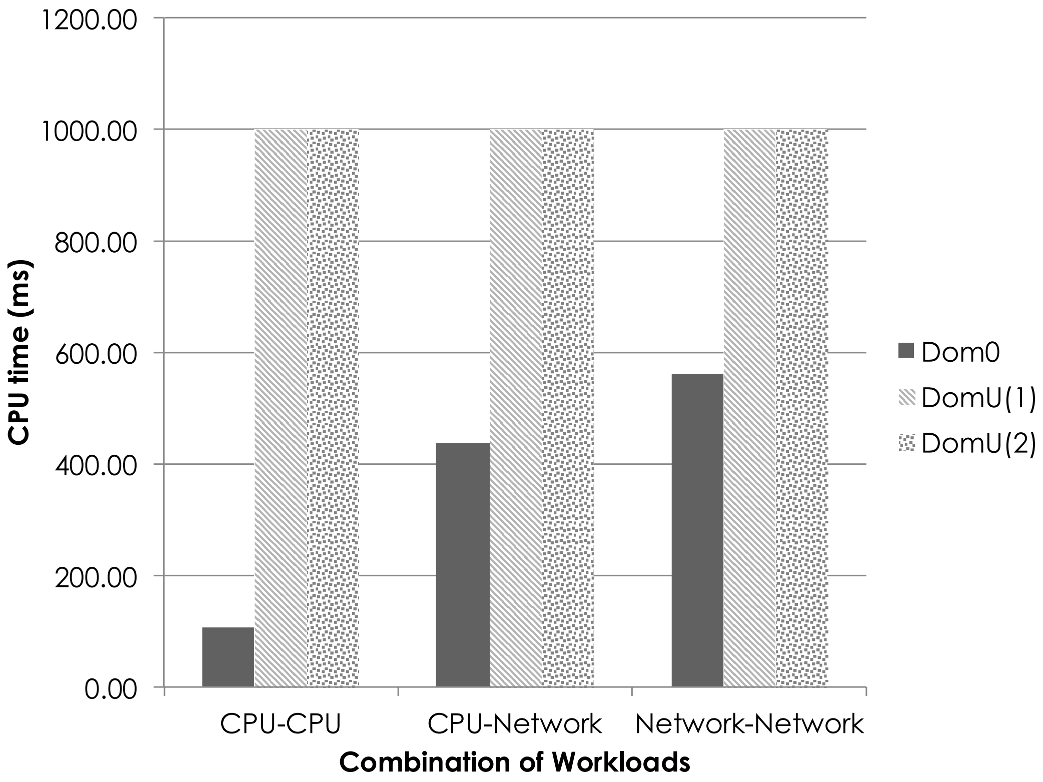


Figure 2.4: CPU Usages of Domains under Various Workload pairs

However, exist hypervisors do not consider about the delegated CPU usage for use of I/O device of domain U. There is no way to show the unfairness and adjust it. Besides, the unfair resource allocation of hypervisor can corrupt reliability of fair share scheduler, billing system of IaaS, and load balancing via live migration on VM server cluster. The feature to gauge delegated CPU usage per-VM for the I/O activity is necessary for the accurate resource accounting.

Chapter 3. Accounting per-VM processor usage

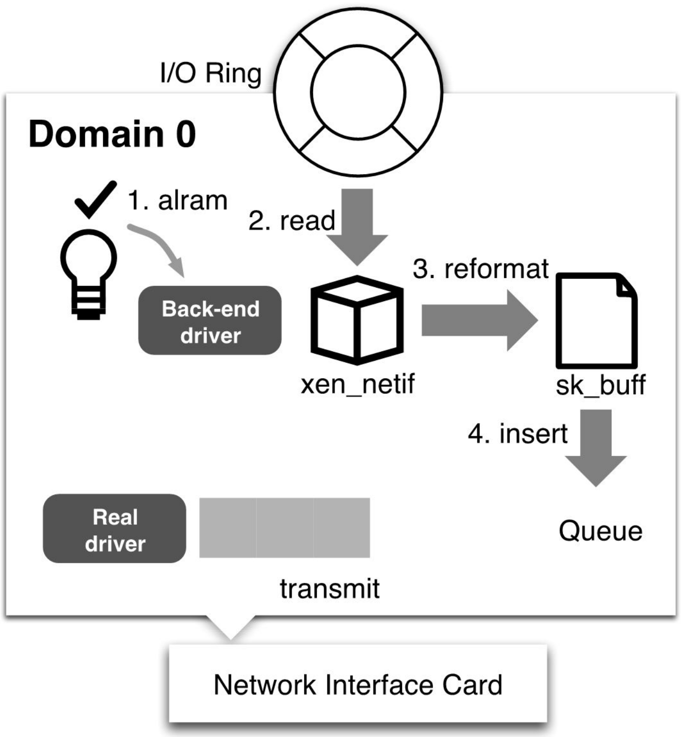
3.1 Accounting framework

To measure delegated CPU usage by domain U, all measurement should be performed at domain 0. The front-end driver is run on each domain U guest, and the CPU usage of the driver is included in the usage of guest that generate I/O request. On the other hand, CPU usage of back-end and real driver those are run on domain 0 is the delegated from unprivileged domains. We implemented the accounting of the delegated CPU usage on Linux 2.6.32, which is used for domain 0.

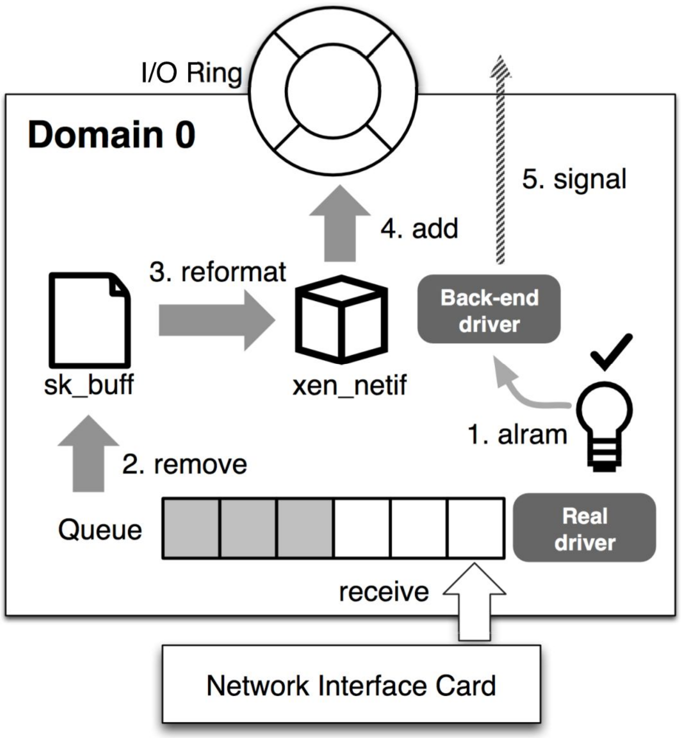
The implementation had required measurement of how much CPU time had been consumed by processing a packet in driver threads. The measurement should be conducted at entire software stack of network. And the stack can be roughly subdivided into a section for virtualization and the other of original operating system.

While the network back-end driver initialize, it allocates memory as the purpose of storing various device information and queue of buffered data. And also, Xen creates a virtual network device for each domain, which is attached to a bridge. *Tasklets* are registered for transmitting and receiving to kernel respectively and adds timers to wake up them.

A running *tasklet* for transmitting requests checks whether one or more events from network front-end driver was occurred or not. If there exist some works to do, it calls the *net_tx_action* function, which reads requests from I/O ring, fills socket structures with the data, and adds to socket buffer of operating system. The submitted requests to buffer are handled in the same way as requests from applications run over domain 0. This process is described in Figure 3.1 (a).



(a) Tx



(b) RX

Figure 3.1: Network drivers run on domain 0

A receiving *tasklet* sends data in the opposite direction of transmitting, like described at Figure 3.1 (b). Network component of operating system put received socket to a buffer. A *tasklet* check the buffer and when the buffer has more then one element, it generates I/O ring data from them and adds to I/O ring. After submitting all requests, back-end driver signals an event to domain U guests.

Fortunately, drivers are run in form of *tasklet* as mentioned above. *Tasklet* mechanism of Linux kernel offers a number of interesting features: a *tasklet* can register itself like timers, it can be scheduled to execute at normal or high priority, it may be run immediately and never later than the next timer tick, and it is strictly serialized with respect to itself. Namely, the same tasklet never runs simultaneously on more than one processor (Corbet et al., 2005). With understanding about these characteristics of *tasklet*, we decided that our implementation does not require consideration of synchronization or preemption of other threads.

The fact of that the device drivers needed no consideration of preemption made the implementation of measuring time laps simple. So, we just inserted time stamp to every step of handling network requests and got the remainder of the two time stamps. If a thread preempts a processor while it runs network driver, measured information is no more reliable because it include time laps for irrelevant jobs.

The function *native_sched_clock* used for time stamp returns current time in nano second units. Linux kernel process scheduler usually calls this function. A guest operating system scheduler uses virtual time to ensure correct sharing of its time slice between threads. Using this function, we got CPU time consumption without interference of other domains.

The VM scheduler allocates CPU to VMs by time slices calculated by scheduling algorithm. The CPU time information is measured using TSC (Time Stamp Counter) resister of a processor. And also, measured CPU time should include the time of that the processor was idle but occupied by a guest. Drivers applied the criterion of scheduler to measuring CPU consumption.

Accounted information has to be categorized according to domain that generation the request. Back-end driver is wakened up by timer and process every request in I/O ring at that time. Chiefly, back-end driver has a number of loops, in which remove an element form a queue and do something with the element repeatedly until the queue is empty. In this case, time stamps inserted inside of the loop. Figure 3.2 is the example of adding time stamps when requests from multiple domains are handled at once.

Xen manage domains with assigning unique number to identify each domain. Similarly, virtual network device of each domain has a device number locally, and the device has a name consisted domain ID and device number. Our purpose is to know delegated CPU usage from domain. So, Our implementation measures CPU time for each request and reads domain ID from processed data for categorizing. And the activity of transmitting and receiving is separated as different processes. Thus, it is possible to measure them respectively.

```
                        … …

    while ((skb = skb_dequeue(&netbk->rx_queue)) != NULL) {
            start = native_sched_clock();
            netif = netdev_priv(skb->dev);

                        … …

            end = native_sched_clock();
            rx_cycles[netif->domid] += end - start;
    }

                        … …
```

Figure 3.2 Example of CPU time measurement

The source code listed in Figure 3.2 is the part of the function of *net_rx_action*. The *native_sched_clock* gets current value of TSC and convert it as nano second. The structure of variable *netif* has domain ID data in it. Profiled data added to array *rx_cycles* indexed by domain ID. Time stamps inserted in the function measure CPU usage of *skb_dequeue* function.

Back-end and networking component of operating system use different data structure. Back-end driver read a request by I/O ring data structure, *xen_netif*. The structure has domain ID field, so it is possible to get domain information that had been sent the request by simply reading it.

However, other components do not handle any domain information directly. To get domain ID, Our implementation use device name. Xen initialize a virtual network device with a name that consist domain number and device number for that a device can have unique name in the system. When a packet is converted to I/O ring data, the back-end driver also uses device name to get domain ID of destination.

Drivers run some jobs those are not separated which domain had generated them. Especially, back-end driver deallocates some shared pages at what point they are not necessary any more. And besides deallocation there are common CPU usage for Network activities from all domain U guest and we measured them separately.

Our implementation used 64 bit integer arrays to profile delegated CPU usage of each domain U. Indices of these arrays mean domain ID directly. Back-end driver sets every element of the arrays by 0 when it is initialized. Values of profiled data are total delegated CPU usage since than.

To avoid synchronization problem through gathering all measurement, each driver has arrays for profiling. Network drivers run through completely separated context from each other, and data transfer is conducted via a number of buffers. In multiprocessor system, they may run at the same time. For this reason, it is dangerous to share an array between those drivers.

3.2 Interfaces and Monitor Application

Our implementation measured actually the CPU usage of domain 0 while domain U guest network. And code for measurement inserted to network drivers. All variables and arrays of drivers are in kernel memory space. That is, the measured information is stored in kernel memory space of domain 0.

It is possible to access profiled data from domain 0 operating system. But there is no way to access them directly from user application or hypervisor. Thus, interfaces to get the data are required for an accurate monitor application and a fair share scheduler with minimum overhead.

Applications can get data in kernel memory space using a system call. But the system call requires overhead of two mode switches. Drivers can also export the data as a file. However, the general file systems for storage are the most time consuming interface and driver must update the file periodically. Finally, to read and use these data on user space, we implemented an interface to export them using *sysfs*.

Sysfs is a mechanism for representing kernel objects, their attributes, and their relationships with each other. It provides two components: a kernel programming interface for exporting these items via in-memory file system, and a user interface to view and manipulate these items that maps back to the kernel objects which they represent. Table 3.1 shows the mapping between kernel constructs and their external *sysfs* mappings.

Table 3.1: Mapping between Kernel construct and *Sysfs*

Internal	External
Kernel Objects	Directories
Object Attributes	Regular Files
Object Relationship	Symbolic Links

A file in *sysfs* is usually formatted in ASCII and when a read operation occurs, the function registered as show attribute return as contents of the file. As a result, user programs can get the information with simple file read operations and the overhead to interface is required only when user read them.

Each device class contains subdirectories for each class object that has been allocated and registered with that device class. And each class object contains files for each attribute. We registered a class device named *netback_stat* using *class_register* function. The class has attributes of *rx_stat* and *tx_stat* to show delegated CPU usage since a VM is launched.

When a user application read an attribute file, *sysfs* calls a function registered to show contents of the file. The show function for *rx_stat* and *tx_stat* return a string generated with total delegated CPU usage of all drivers and blanks as delimiter between values of each domain. Therefore, user applications can get profiled data of every domain at one read operation.

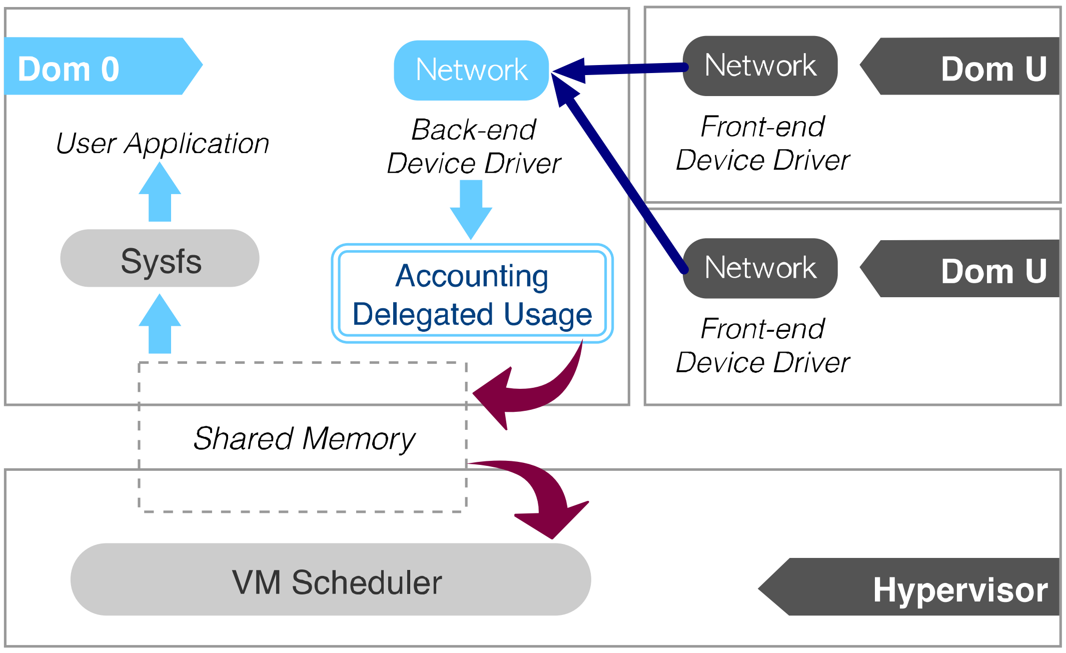


Figure 3.3: Interface to userspace on domain 0 and hypervisor

There is no other way to access kernel memory space of domain 0 guest from hypervisor except shared memory mechanism. Usually, the operating system can access hypervisor through hypercall. If domain 0 sends the profiled data to hypervisor using hypercall, it generates high overhead. And also, hypervisor does not support general file system as a kernel. However, using shared memory mechanism support simple communication method. And hypervisor can get latest profiled data in real time.

When back-end driver is initialized, it gets a page to share for each array using the function *__get_free_page*. The function returns pseudo-physical memory addresses of domain 0 for allocated page. And using the function *virt_to_mfn*, convert returned address to machine frame number and map to a pointer of hypervisor using function *map_domain_page*. Finally, the pointer value is sent via an implemented hypercall.

Figure 3.3 describes all interfaces implemented our accounting framework. Using the interface between user application and kernel memory space, a user application get profiled data by simple file read operation. And using the interface between hypervisor and domain 0 guest, hypervisor can read the data from shared memory directly. These interfaces do not decrease the overall system hardly at all.

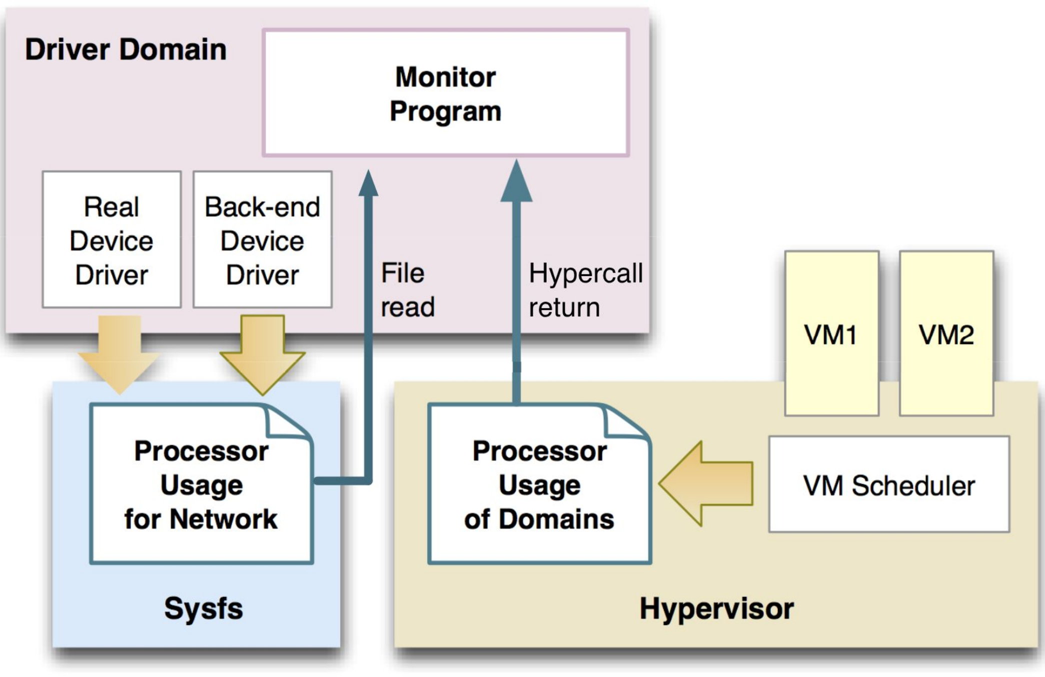


Figure 3.4: Monitor application using accounting per-VM processor usage

We implemented a monitor program run on domain 0 guest using our accounting framework and the interface to userspace. It can collect not only delegated CPU usage for network activities, but also network and CPU usage of unprivileged domains. The monitor aggregates all profiled data every one second.

It read the file whose path is */sys/class/netback_stat* to collect delegated CPU usage. The returned string consists of profiled data and blank, so the monitor has to pars it. Collected data is managed by domain id like the accounting framework.

VM scheduler profiles the CPU usage of VMs, so the data can be accessed by hyper call. For this reason, we added new hypercall to get accurate time information directly. This hypercall returns CPU time as nanosecond unit and the number of instructions when the argument is domain id of running VM.

The network usage of VMs, like number of transferred packets and byte, are obtained from files in *procfs*. Xen hypervisor resisters a virtual network card as a device that is connected to bridge. Thus, the network usage is profiled by operating system of domain 0. The monitor is also able to read by file operation.

The collected information is profiled since the server had been booted or the domain had been launched. The monitor save last data to memory, and get difference between current data. As a result, the printed value is usage for one second.

Using the monitor, users can compare and analyze scheduled CPU usage and delegated for Networking. We used this monitor program in our all evaluation.

Chapter 4. Evaluation

All the experiments were performed on a VM server with Intel core i7 processor and 8G of RAM memory. For these measurements, we used the Linux 2.6.32 and Xen hypervisor 4.1.0. Each VM has one virtual CPU and a 512M memory and the guest operating system is Ubuntu 10.04 server.

4.1 Experiments while VMs running various workload sets

The first group of experiments is same as introduce above at 2.2 . In the experiments, hypervisor assigned same time slice to both domain U guests. And only one of the domains used network additionally. From previous experiments, we knew that every domain U guest consumed same amount of CPU time. However, domain 0 guest consumed more CPU time while one or both domain used network. And now, we address where these usage comes from.

Using our monitor program introduced above, we measured some resource usage while two domains run workload pairs respectively. The monitor read profiled information every one-second, but each operation may be executed at once. For example, the monitor read delegated CPU usage, after than back-end device driver woke up and process some network request. After finishing them, kernel scheduler reruns the monitor, which gets network usage. In this situation, there is no coherence between delegated usage and network usage. To hide this kind of error, we measured 500 times continuously, and used average of them for comparison.

Table 4.1 summarizes the measurements collected during these experiments. ``Common CPU time" is time for networking of unprivileged domains but unable to categorize. ``Packets means" the number of packets each domain sent or received. ``CPU time" means CPU usage that an unprivileged domain delegates to network.

At first experiment, the CPU-CPU workload set, all delegated CPU time include common is close to 0. The reason that the value is not perfectly zero is operating system send or receives keep its networking state connected. But it can be ignored.

Table 4.1: Resource usage while VMs run various workload pairs

		Common	DomU(1)		DomU(2)	
			Packets	CPU time (ms)	Packets	CPU time (ms)
CPU-CPU	Rx	0.130	0.051	0.000	3.413	0.008
	Tx	0.027	32.810	0.037	46.703	0.064
CPU-NET	Rx	24.099	0.020	0.000	19835.535	29.012
	Tx	21.151	13.608	0.015	39688.393	35.945
NET-NET	Rx	32.545	11057.220	16.832	10807.201	17.152
	Tx	32.888	22139.772	19.818	21637.495	21.188

We analyzed the results of the experiments on the workload pairs including network in two aspects. The first aspect is the relationship between network usages as the number of transferred packet and delegated CPU usage classified into domain that generated network requests. The other aspect is the relationship between network usage and common delegated

CPU usage for network. For more definite comparison, Figure 4.1 presents the delegated CPU usage under given workload pair.

In the experiment of CPU-Network workload pair a domain, DomU(2), had networked alone, so it got much higher throughput. And two domains got definitely lower throughput in third experiment of Network-Network pair. However, the total throughput of all unprivileged domains was higher then the second experiment. They could not consume network bandwidth fully. Because of the interference other process of CPU intensive workload, the networking process cannot make enough packets to use the network bandwidth fully. But when two domains network concurrently, that two guests produce requests and domain 0 consumes at once can be more efficient.

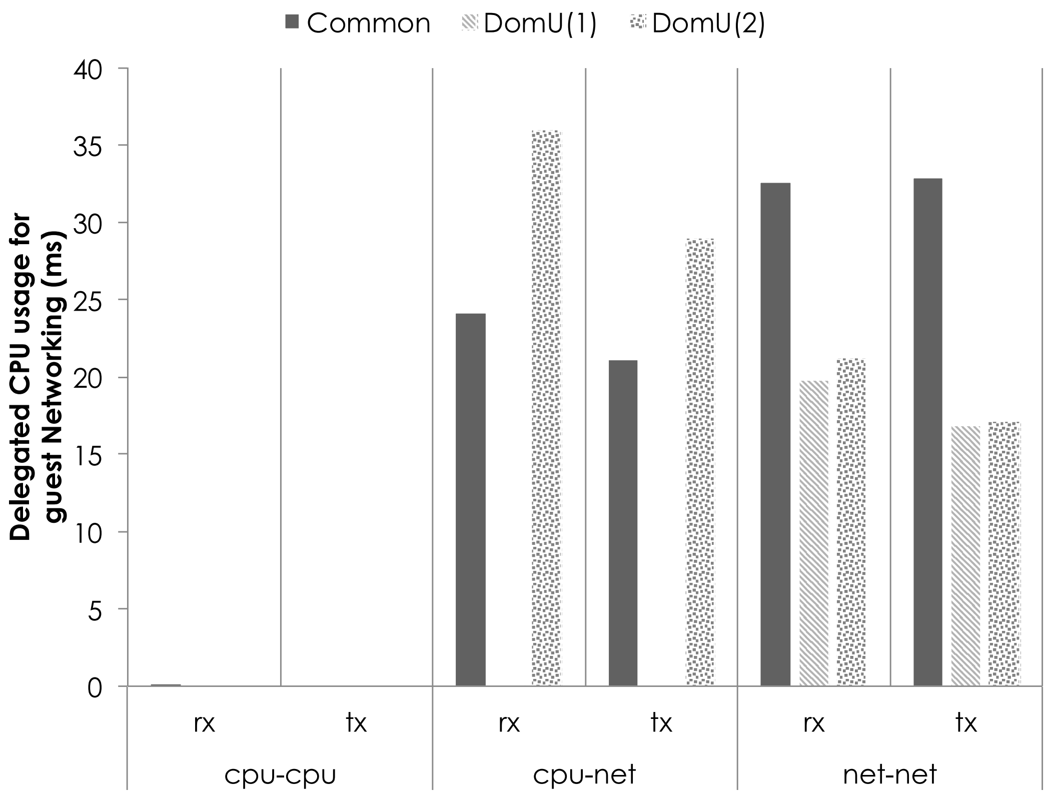


Figure 4.1: Delegated CPU usage while VM runs various workload pairs

We analyzed how much CPU usage unprivileged domains delegated for networking. Figure 4.1 shows that CPU usage that each domain delegates to network have a similar result to the number of packets. DomU(2) delegated most CPU time in second experiment, DomU(1) and DomU(2) in third experiment delegated slightly more than half of them. Comparing with both domain of only third experiment, although a domain that networked more delegates less CPU usage but the values are kind of similar. With consideration about measurement error, we can say a domain delegates more CPU usage when they network more.

Domain 0 consumed more common CPU usage when both domains use network. Form this group of experiments; it is hard to ensure what is the reason of increase of common usage. There are two possibilities in our experiments to increase common usage. At first, total network usage in the VM server was increased. Second, more domain were using network concurrently. Or both of them can affect in combination. To ensure its reason of the increase, we performed another experiment with various number of domains that is networking.

4.2 Experiments on various number of networking domains

The purpose of this experiment is to show effects of number of unprivileged domains that networking concurrently. Thus, every domain was created with same VM definition, installed same operating system, and run same applications. The VM server runs 4 VMs always, and we changed the number of networking domain by executing the application that uses network or not.

Unlike the experiment that introduced previously, domains did not run CPU intensive process. As a result, domains could not consume assigned time slice fully, and the networking domains required more CPU time then others not networking. According to result of measurement described in Table 4.2, domains transmitted or received about double amount of networking usage compared with previous experiments. And total network usage is more when number of networking domains on a VM server was smaller. But as in previous experiment, unprivileged domains shared limited network bandwidth.

Table 4.2: Average of transferred packets while multiple domains network

Number of Networking guests		Guest (1)	Guest (2)	Guest (3)	Guest (4)
4	Tx	11398	9320	9239	9708
	Rx	22786	18636	18472	19409
3	Tx	14793	12719	12146	0
	Rx	29581	25439	24293	0
2	Tx	23339	16420	0	0
	Rx	46668	32847	1	1
1	Tx	39790	0	0	0
	Rx	79520	0	0	0

Figure 4.2 is a graph for comparison of delegated CPU usage of all domains. The total sum of delegated CPU usage by each domain except common usage has always a similar value no matter how many VMs are using network. When VMs share network hardware, networking domains delegates a portion of total usage at a rate of occupied network bandwidth.

The common CPU usages for network activities of domain U spend a great part of entire CPU usage of domain 0. And it made a difference of total delegated CPU usage while various numbers of VMs using network. That the common usage is large means that total delegated usage is also large. Domain U guests transferred fewer packets while more VM sent and received packets and produced more common usage. So, we had believed the number of networking domain affects common CPU usage.

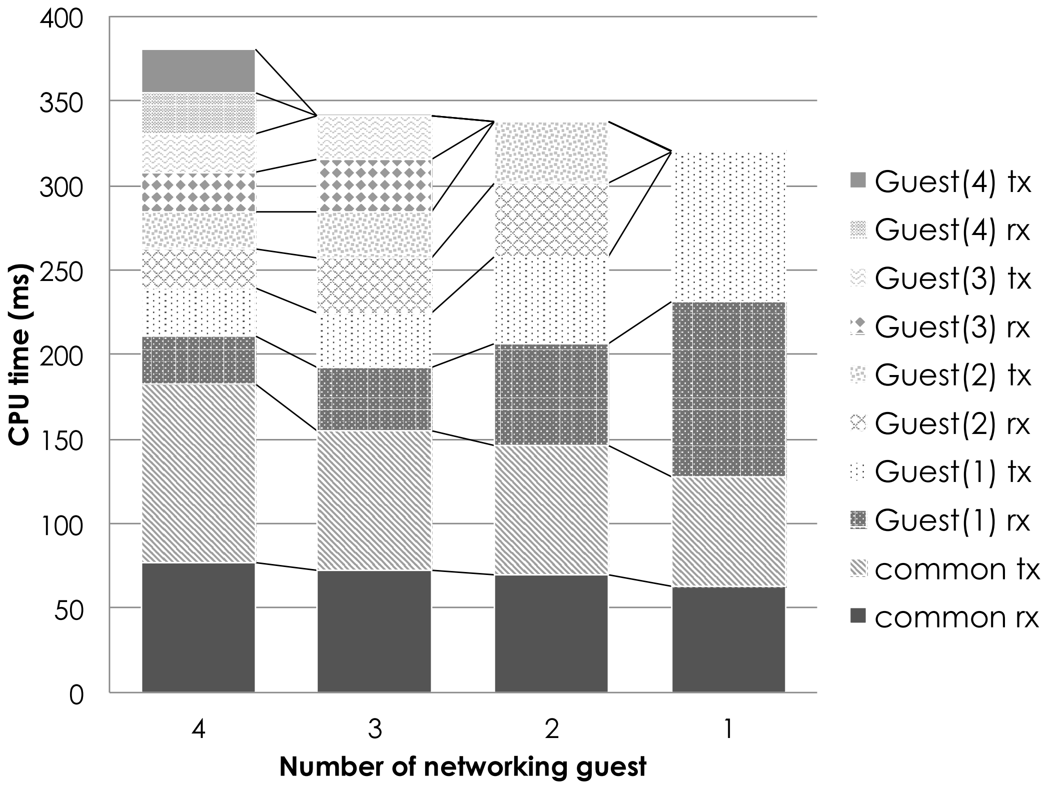
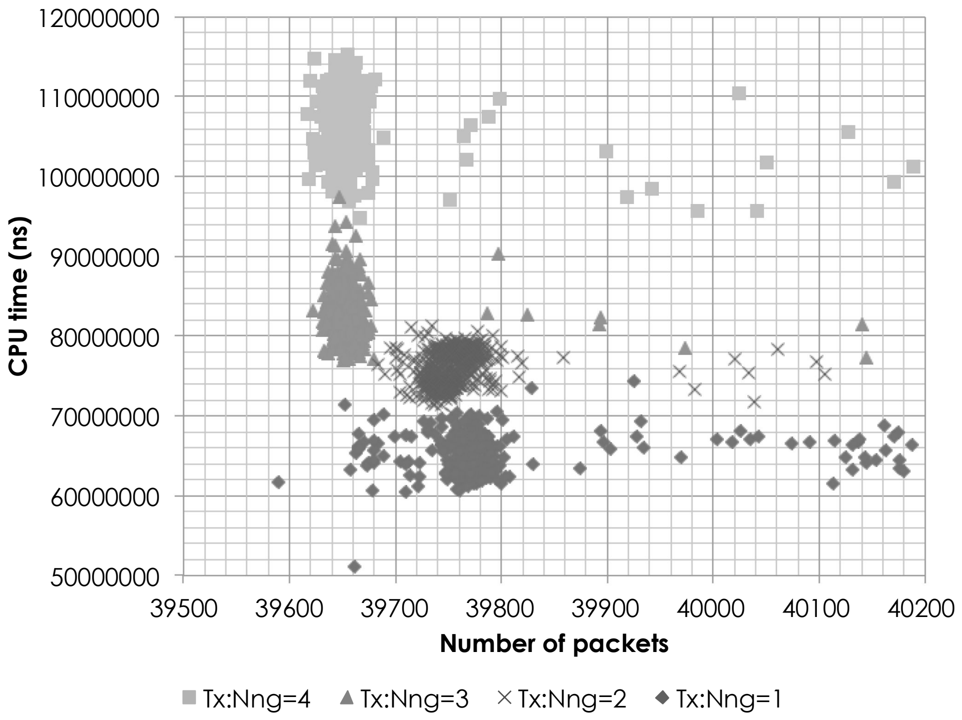
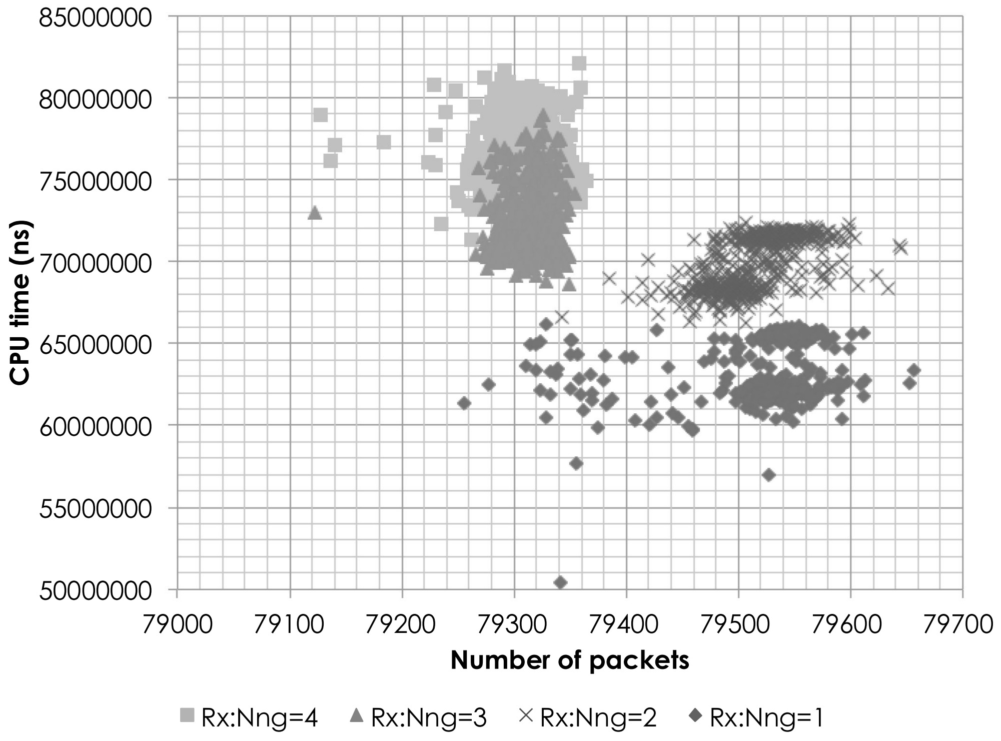


Figure 4.2: Delegated CPU usage while a server runs multiple networking domains

Figure 4.3 shows clearly that common usage is not related with network usage. The graph presents common usage of total some of packets number. This graph shows that there is no relationship between the number of packets and common CPU time. But it is clear that the increase of the number of networking domain leads to increase of common CPU usage for network activities.



(a) Tx



(b) Rx

Figure 4.3: Common CPU usage for total networked packets

The categorized usage was larger when the domain sends or receives more packets although it was not always true. Data that was used experiment above was average. It made hard to get correlation of delegated CPU usage and network usage. To see the relationship between delegation and network usage, Figure 4.4 shows categorized CPU usage that delegates from domain U for number of sent or received packets of them.

We ignored which domain produce the usage, how many VMs connected and transferred data through network. Figure 4.4 shows only correlation of CPU usage and packet number, which is directly proportional to each other. According to our measurements, the job to send a packet consumes more CPU time than receive a packet. The slope of trend line of "Tx" is greater then "Rx".

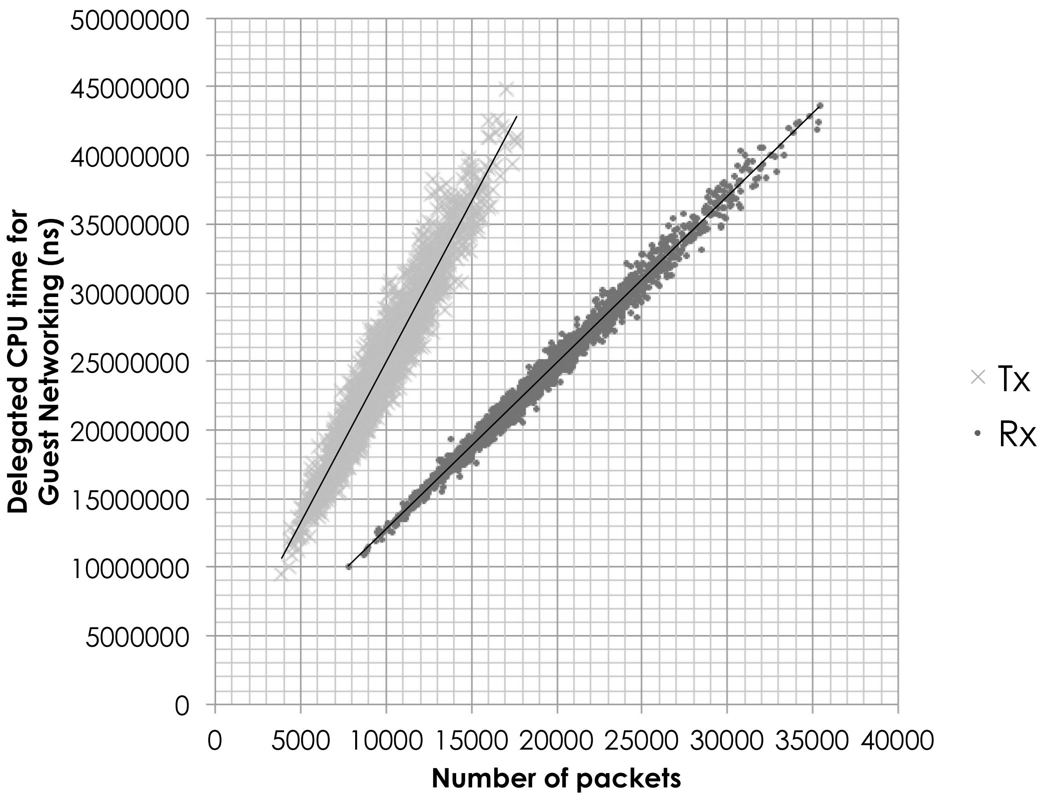


Figure 4.4: Delegated CPU usage of domain U guests for the number of packets

Chapter 5. Related work

Pu et al. introduced performance interference among different VMs running on the same hardware platform with the focus on network I/O processing. According to their study, network intensive workloads can lead to high overheads due to extensive context switches and events. But when the server runs CPU intensive and network intensive workloads in conjunction incurs the least resource contention, delivering higher aggregate performance. (Pu et al.).

Cherkasova and Gardner introduced a light weight monitoring system for measuring the CPU usage of different VMs including the CPU overhead in the privileged domain caused by I/O processing oh behalf of a particular VM. They measured the number of memory page exchanges between domain 0 and domain u performs over arbitrary time interval when domain 0 is in execution state. They derived the CPU cost through dividing the time interval by the number of memory page exchanges. They used the calculated cost to ``charge'' the corresponding VM that caused the I/O activities (Cherkasova and Gardner).

The use of driver domains to host device drivers has become popular for the reason of reliability and extensibility. Xen had started to use driver domain since the version 2.0, and VMware workstation (Sugerman et al.) also used it. But the existence of driver domain causes a performance penalty for device access. The problems introduced in Sugernam et al. (Sugerman et al.) and in Menon et al. (Menon et al.)

Xenoprof (Menon et al.) is a system-wide statistical profiling toolkit and developed for Xen hypervisor environment. Also, the toolkit provides capabilities similar to Oprofile that using performance monitoring hardware to collect periodic samples of performance data.

It is not a system embedded profiler, and users have to instruct the start and end point. Thus, it is suitable for performance debugging or measure the overhead.

Menon et al. addressed a number of techniques for optimizing network performance with retaining the basic Xen architecture of locating device driver in a privileged domain and providing network access to unprivileged domain through virtualized network interface. This optimization improved network throughput and reduce execution overhead by I/O channel optimization.

Joule meter (Kansal et al.) is a software approach to measure the energy consumption of virtual machines in a consolidated server environment. Joule meter estimates the amount energy that each virtual machine consumes by monitoring its resource usage dynamically. This research supposed accounted resource usage is accurate but is not. Our accounting framework can be applied for software base power metering.

Chapter 6. Conclusion

When a VM server runs multiple VMs concurrently that have different workload characteristics, current Xen hypervisors cannot distribute hardware resource to VM fair. One reason for this, virtualized environment have a separated privileged domain to perform I/O activities multiplexed from all VMs run on same host. We experimented to see that when hypervisor divides CPU time fair, some VMs required using network cause extra usage in privileged domain.

We considered that the CPU usage to handle I/O request of unprivileged domains is delegated from the domain that generates the request. In this paper, we introduced the implementation of accounting delegated CPU usage per-VM for network activities. The implementation includes accounting framework, user interface, and a monitor program.

We measured delegated CPU usage by separating CPU time that is able to categorize by unprivileged domains that requests and common usage. And also, the measurement was performed at each process of sending and receiving separately. Through two group of experiment, we showed the proportional relationship between number of networked packets and delegated CPU usage per-VM except common usage. The common delegated CPU usage was increased according to the number of VMs that network concurrently on a shared machine.

In result of our experiment, a domain U guest had stolen about 110ms of CPU time of domain 0 when one VM use network and another never uses network. It caused over resource of 11\% as compared with resource usage of other VM run at the same time.

The features introduced in this paper can show how much CPU resource is really consumed by VMs. Using the resource accounting adjusted our implementation, VM providers can allocate hardware resource fair. And on a cloud service, it is possible to build a fair billing system based on real resource usage of VMs.

References

1. Adams, K. & Agesen, O. 2006. A comparison of software and hardware techniques for x86 virtualization. Proceedings of the 12th international conference on Architectural support for programming languages and operating systems. New York, NY, USA: ACM.

2. Amazon. Amazon Elastic Compute Cloud [Online]. Available: http://aws.amazon.com/ec2/.

3. Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. & Zaharia, M. 2010. A view of cloud computing. Commun. ACM, 53, 50--58.

4. Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A. & Zaharia, M. 2009. Above the Clouds: A Berkeley View of Cloud Computing. UC Berkeley.

5. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. & Warfield, A. 2003. Xen and the art of virtualization. Proceedings of the nineteenth ACM symposium on Operating systems principles. New York, NY, USA: ACM.

6. Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J. & Brandic, I. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst., 25, 599--616.

7. Cherkasova, L. & Gardner, R. 2005. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. Proceedings of the annual conference on USENIX Annual Technical Conference. Berkeley, CA, USA: USENIX Association.

8. Cherkasova, L., Gupta, D. & Vahdat, A. 2007. Comparison of the three CPU schedulers in Xen. SIGMETRICS Perform. Eval. Rev., 35, 42--51.

9. Chisnall, D. 2008. Definitive Guide to the Xen Hypervisor, Prentice Hall.

10. Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I. & Warfield, A. 2005. Live migration of virtual machines. Proceedings of the 2nd

conference on Symposium on Networked Systems Design \& Implementation - Volume 2. Berkeley, CA, USA: USENIX Association.

11. Corbet, J., Kroah-Hartman, G. & Rubini, A. 2005. Linux Device Drivers, O'Reilly Media, Inc.

12. Fraser, K., H, S., Neugebauer, R., Pratt, I., Warfield, A. & Williamson, M. 2004. Safe hardware access with the Xen virtual machine monitor. Proceedings of the first Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS).

13. Kallahalla, M., Uysal, M., Swaminathan, R., Lowell, D. E., Wray, M., Christian, T., Edwards, N., Dalton, C. I. & Gittler, F. 2004. SoftUDC: A Software-Based Data Center for Utility Computing. Computer, 37, 38--46.

14. Kansal, A., Zhao, F., Liu, J., Kothari, N. & Bhattacharya, A. A. 2010. Virtual machine power metering and provisioning. Proceedings of the 1st ACM symposium on Cloud computing. New York, NY, USA: ACM.

15. Kim, N., Cho, J. & Seo, E. 2011. Energy-Based Accounting and Scheduling of Virtual Machines in a Cloud System. Proceedings of the 2011 IEEE/ACM International Conference on Green Computing and Communications.

16. Kumar, R. 2007. U.S. Data Centers: The Calm Before the Storm. Gartner White Paper.

17. Menon, A., Santos, J. R., Turner, Y., Janakiraman, G. J. & Zwaenepoel, W. 2005. Diagnosing performance overheads in the xen virtual machine environment. Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments. New York, NY, USA: ACM.

18. Nanda, S. & Chiueh, T.-C. 2005. A survey of virtualization technologies.

19. Oprofile. Oprofile [Online]. Available: http://oprofile.sourceforge.net/.

20. Ostermann, S., Iosup, R., Yigitbasi, N. & Fahringer, T. 2009. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. In ICST International Conference on Cloud Computing.

21. Pu, X., Liu, L., Mei, Y., Sivathanu, S., Koh, Y. & Pu, C. 2010. Understanding Performance Interference of I/O Workload in Virtualized Cloud Environments. Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on.

22. Rackspace. Rackspace [Online]. Available: http://www.rackspace.com/.

23. Sugerman, J., Venkitachalam, G. & Lim, B.-H. 2001. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. Proceedings of the General Track: 2002 USENIX Annual Technical Conference. Berkeley, CA, USA: USENIX Association.