Towards Scalable and High Performance I/O Virtualization – A Case Study

Jinpeng Wei[1], Jeffrey R. Jackson[2], and John A. Wiegert[2]

[1] Georgia Institute of Technology, 801 Atlantic Drive
30332 Atlanta, Georgia, USA
weijp@cc.gatech.edu
[2] Intel Corporation, 2111 NE 25th Ave
97124 Hillsboro, Oregon, USA
{jeff.jackson,john.a.wiegert}@intel.com

Abstract. I/O Virtualization provides a convenient way of device sharing among guest domains in a virtualized platform (e.g. Xen). However, with the ever-increasing number and variety of devices, the current model of a centralized driver domain is in question. For example, any optimization in the centralized driver domain for a particular kind of device may not satisfy the conflicting needs of other devices and their usage patterns. This paper has tried to use IO Virtual Machines (IOVMs) as a solution to this problem, specifically to deliver scalable network performance on a multi-core platform. Xen 3 has been extended to support IOVMs for networking and then optimized for a minimal driver domain. Performance comparisons show that by moving the network stack into a separate domain, and optimizing that domain, better efficiency is achieved. Further experiments on different configurations show the flexibility of scheduling across IOVMs and guests to achieve better performance. For example, multiple single-core IOVMs have shown promise as a scalable solution to network virtualization.

1 Introduction

I/O Virtualization provides a way of sharing I/O devices among multiple guest OSes in a virtualized environment (e.g., Xen [2]). Take network virtualization for example (see Fig. 1), here the platform has one Gigabits/second network link, but a guest OS may only need 100Mbps network bandwidth, so it would be very cost-efficient to share this Gigabit link among several guest OSes. In order to support this kind of device sharing, Xen has employed a split-driver design, where the I/O device driver is split into a *backend* and a *frontend*. The physical device is managed by a *driver domain* which acts as a proxy between the guest OSes and the real device. The driver domain creates a device backend, and a guest OS which needs to access the device creates a frontend. The frontend talks to the backend in the driver domain and creates an illusion of a physical device for the guest OS. Multiple guest OSes can share a physical device in this way through the backend in the driver domain.

R. Perrott et al. (Eds.): HPCC 2007, LNCS 4782, pp. 586–598, 2007.

The most common way of deploying driver domains in Xen is to have the service OS (e.g. domain 0) as the single driver domain (called *centralized* I/O Virtualization in this paper), as shown in Fig. 1. However, this centralized architecture is not scalable when the platform has many devices, as in a server consolidation environment. Specifically, as the number and variety of shared devices increase, the centralized I/O Virtualization has the following problems. First, the Service OS can be easily overloaded by I/O virtualization. Second, any optimization of the Service OS for better I/O virtualization performance needs to consider all other tasks in the service OS (e.g., service daemons and management tasks), so it may not be easy to do such optimizations. Third, different devices may have different needs for better performance. For example, a graphics device may be more sensitive to latency while a network device may be more interested in throughput, so it may be difficult to find an optimization that satisfies both graphic devices and network devices at the same time.

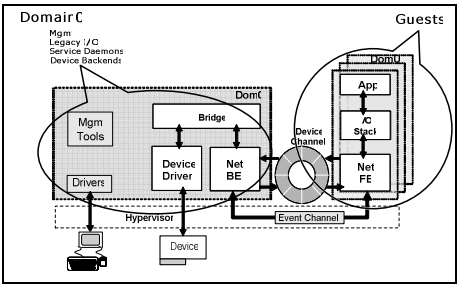

Fig. 1. Centralized I/O Virtualization Architecture

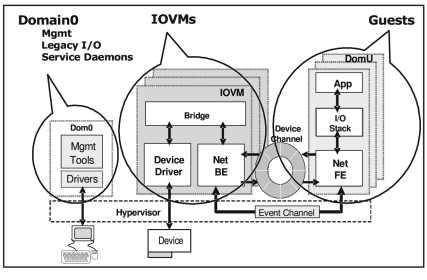

Fig. 2. IOVM Architecture

One possible solution to the scalability problems with *centralized* I/O Virtualization is *scale-up* (e.g., adding more resources to the Service OS). However, our experience with Xen (version 3.0.2) shows that allocating more computing resources (e.g., more CPUs) does not necessarily translate into better I/O virtualization performance. For example, we found that a Service OS domain (uniprocessor Linux) with only one CPU saturates before it can support 3 Gigabits/second NICs (Network Interface Cards) at full capacity. Adding one more CPU to this domain (SMP Linux) does not show much improvement - the receive throughput increases by 28%, but at the cost of reduced efficiency (by 27%); the transmit throughput even decreases by 6.6%, and the transmit efficiency drops by 88%. The definition of efficiency can be found in Section 4.1.

The difficulties of *scale-up* lead us to consider another way of deploying driver domains: *scale-out,* e.g., using dedicated guest domains other than the service OS as driver domains. This idea was suggested by Fraser [5] and Kieffer [7], and we have also proposed IOVMs [1]. The IOVM approach can solve the three problems with centralized I/O Virtualization. The first problem is solved by moving the I/O Virtualization out of the service OS, so it will not be overloaded by I/O Virtualization. Second, any possible optimization is performed in a different domain from the service OS so it does not affect the existing tasks. Third, when different devices have different requirements for high performance, we can create different IOVMs and optimize each

one in a different way, according to the characteristics of the device and workload. Therefore the IOVM approach opens up opportunity for scalable I/O virtualization given that proper optimizations are done in the IOVMs and the addition of IOVMs to the platform does not consume excessive resources.

This paper presents our initial experience with IOVMs for *network virtualization* on a multi-core platform. The question we want to address is - Given a platform equipped with multi-core, how can its support for network virtualization *scale* as the number of network devices increases? By 'scale' we mean 'aggregated bandwidth increases linearly with the increase of computing resources (CPU cycles) on the platform'. We believe that the software architecture (including the network device driver and protocol stack) is a key to the solution. In this paper, we show how IOVMs enable the division of work load among cores in such a way that scalability can be achieved by incrementally starting IOVMs on new cores. Specifically, this paper makes the following contributions:

- It proposes a novel way of using IOVMs for scalable I/O Virtualization. Although dedicated driver domains were initially proposed for fault isolation [8][5], we observe and show experimentally in this paper that it is also beneficial to scalable I/O Virtualization.
- The second contribution of this paper is a comprehensive set of experimental results to evaluate the idea of IOVMs. It compares the performance of three different configurations of network IOVMs: Monolithic IOVM, Multiple Small IOVMs, and Hybrid IOVMs. And it concludes that the Hybrid IOVM configuration offers a promising balance between scalable throughput and efficient use of core resources.

The rest of this paper is organized as follows. Section 2 describes the IOVM architecture. Section 3 briefly outlines the optimizations that we have carried out in a Network IOVM. Section 4 presents a series of experimental results which evaluate the performance and scalability of the Network IOVM. Related work is discussed in Section 5 and we draw conclusions in Section 6.

2 IOVM Architecture

This section describes our IOVM architecture (See Fig. 2). The main idea is to move I/O Virtualization work out of the service OS (domain 0) and into dedicated driver domains.

In Xen an IOVM is a specialized guest operating system, so it has the basic features and structure of a modern operating system (e.g., memory management and process management). In addition, it has the following components. First, it contains the native device driver for the physical device(s) that it virtualizes. Second, it runs the backend drivers for guest domains interested in sharing the physical device. Finally, it runs any multiplexer/demultiplexer that glues the first two together (e.g., code for routing). A network IOVM essentially has the structure of a switch or router.

2.1 Different IOVM Configurations

Three different IOVM configurations can be used to virtualize devices: Monolithic, Multiple Small IOVMs, and Hybrid.

Monolithic IOVMs (Fig. 3)**:** All devices are assigned to a single IOVM. As a result, the platform only has one IOVM, but this IOVM can be very heavy-weight due to a large number of devices to be virtualized.

Multiple Small IOVMs (Fig. 4)**:** Each device is assigned to a dedicated IOVM. So the number of IOVMs is equal to the number of physical devices being shared. When there are many devices, this configuration can result in many IOVMs.

Hybrid IOVMs (Fig. 5)**:** This configuration is a compromise between the first two configurations. In this configuration, there are multiple IOVMs, and each IOVM is assigned a subset of the physical devices being shared. The IOVMs are medium-sized, so they are larger than those in the Multiple Small IOVMs configuration, but they are smaller than a Monolithic IOVM. A Hybrid IOVMs configuration results in a smaller number of IOVMs in the system compared with the Multiple Small IOVMs configuration, but a larger number of IOVMs compared with the Monolithic IOVM configuration.

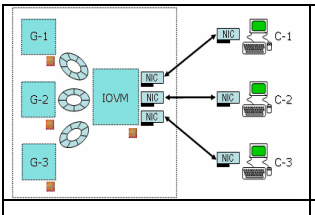

Fig. 3. Monolithic Configuration

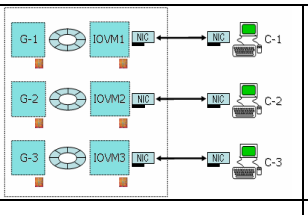

Fig. 4. Multiple Small IOVMs Configuration

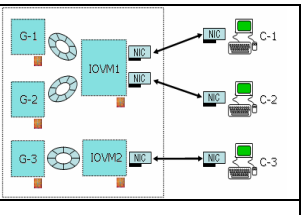

Fig. 5. Hybrid Configuration

3 An IOVM for Network Virtualization

Having a separate IOVM gives us much freedom in terms of constructing it. For example, we can start from a commodity OS (such as Linux) and specialize it; we can also build a custom OS from scratch. In this project we choose the first approach. Specifically, we customize the para-virtualized Linux for Xen for a network IOVM. The customizations (optimizations) that we perform fall into three categories: kernel, network protocol stack, and runtime.

Minimal kernel for a Network IOVM: We use the Linux configuration facility to remove irrelevant modules or functionalities from the kernel, e.g., most of the device drivers, IPv6, Cryptography, and Library Routines. The essential part of the kernel (e.g., memory and process management) is kept. Besides, we keep the list of functionalities shown in **Table 1**. The network interface card driver is configured to use polling for receive. Due to the inefficiency of SMP Linux on network virtualization, we turned off SMP support in the Network IOVM for the rest of the paper. By performing this step, we reduce the compiled size of the "stock" kernel by 44%.

Minimal network protocol stack: To make the network IOVM fast we use Ethernet Bridging in Linux kernel to forward packets between the guest OS and the NICs (Network Interface Cards). An Ethernet bridge processes packets at the data link layer

(e.g. only looking at the MAC addresses to make forwarding decisions). We do not use layer 3 or above forwarding, and we do not use iptables. As another special note, we found that bridged IP/ARP packets filtering is very CPU intensive: For example, although it happens at the data link layer, it computes IP checksum, looks up routing table, and replicates packets. In other words it adds significant processing to the critical path of every packet, even if no filtering rules are defined. So it is disabled in the network IOVM for better performance.

Minimal IOVM runtime: In this part we shutdown most of the irrelevant services in the network IOVM (e.g., *sendmail*) to save CPU cycles. As a result only *network*, *syslog* and possibly *sshd* are needed to support a network IOVM. We also start the network IOVM in a simple run-level (multiuser without NFS).

Table 1. Kernel Functionality Required for the Network IOVM

Basic TCP/IP networking support	802.1d Ethernet bridging
Packet socket	Unix domain socket
Xen Network-device backend driver	Ext2 , ext3, /proc file system support
Initial RAM disk (initrd) support	Network Interface Card driver (PCI, e1000)

4 Evaluation of the Network IOVM

In this section, we present several experiments which lead to a method for scalable and high performance network virtualization.

4.1 Experiment Settings

We test the idea of network IOVMs on an Intel platform with two Core Duo processors (4 cores total). This platform has several gigabit NICs. Each NIC is directly connected to a client machine, and is exclusively used by a guest OS to communicate with the client machine (Figures 3-5). The maximum bandwidth through each NIC is measured by the iperf benchmark [6]. There are 4 TCP connections between each guest OS and the corresponding client machine, which start from the guest OS, traverse through the IOVM and the NIC, and reach at the client machine. The connections are all transmitting 1024 byte buffers. The combined bandwidth of the NICs is considered the aggregated bandwidth (throughput) supported by the platform. Both Xen 3.0.2 and Xen-unstable (a version before Xen 3.0.3) are used as the virtual machine manager, and a para-virtualized Linux (kernel 2.6.16) is used as the guest OS. The guest OSes and IOVMs have 256MB memory each.

We me asured two metrics: **throughput** and **efficiency**. Throughput is measured using the microbenchmark iperf. Efficiency is calculated by dividing the aggregate throughput transferred by the IOVM(s) by the number of processor cycles used by the IOVM(s) during the workload run. The processor cycles used by the IOVM(s) are measured using xentop, a resource measurement tool included in Xen. This results in a value measured in bits transmitted per processor cycle utilized or bits/Hz for short.

Network bandwidth through each NIC is measured in two directions: (1) from the client to the guest OS (denoted as Rx, for receive), (2) from the guest OS to the client (denoted as Tx, for transmit).

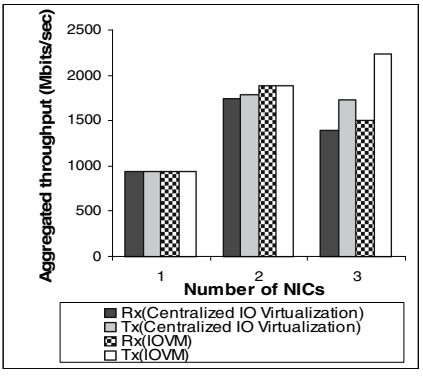

Fig. 6. Comparison of Throughput

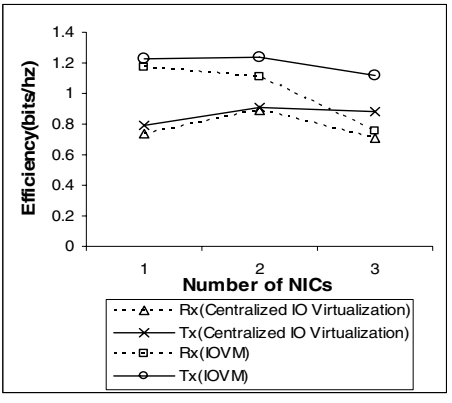

Fig. 7. Comparison of Efficiency

4.2 Making a Case for Network IOVMs

The first experiment compares the performance of centralized network virtualization versus a network IOVM. Here we use a Monolithic configuration (Fig. 3) with 3 NICs. Fig. 6 shows the throughput results for different number of NICs and different virtualization architectures (Rx means receive and Tx means transmit in this paper). From Fig. 6 we can see that the throughputs are the same for centralized network virtualization and IOVM network virtualization when only one NIC is used by iperf. It is so because at this load the CPU is not a bottleneck so both architectures can support nearly line rate (e.g. 940Mbits/sec). When we move on to 2 NICs, we can see that the IOVM network virtualization continues to deliver nearly line rate (1880 Mbits/sec), but the centralized network virtualization can not (1760Mbits/sec). This result shows that at 2 NICs, the Service OS in centralized network virtualization starts to be saturated. The IOVM is doing better because it is optimized for networking as described in Section 3. When we move on to 3 NICs, it becomes more apparent that IOVM network virtualization can support higher throughput. E.g., 2230Mbits/sec versus 1700Mbits/sec for transmit (denoted as Tx), and 1500Mbits/sec versus 1400Mbits/sec for receive (denoted as Rx). Comparatively the benefit of IOVM is not as big for receive as it is for transmit. The reason is that receive is more CPU intensive in current implementation (polling is used). As a result the optimization in the network IOVM is still not enough to meet the increase in CPU demand when there are 3 iperf loads. By carrying out more aggressive optimization in the network IOVM we may be able to support higher throughput. But the point of Fig. 6 is that using a network IOVM has advantage in terms of throughput.

Fig. 7 shows the result for efficiency (Section 4.1), which is more revealing about the benefit of a network IOVM. For example, when there is only one NIC used by iperf, the efficiency is about 1.2 bits/Hz for IOVM and about 0.8 bits/Hz for centralized network virtualization, meaning that network IOVM is 50% more efficient than centralized network virtualization. In other words, although the two architectures appear to support the same level of throughput (940Mbits/sec in Fig. 6) at one NIC level, the network IOVM is actually using much less CPU cycles to support that

throughput. So a network IOVM is more efficient in terms of CPU usage. This claim is also true for the 2 NICs case in Fig. 7. When there are 3 NICs, the efficiency of IOVM is still much higher than that of centralized network virtualization in terms of transmit, but the efficiency difference in terms of receive becomes small. This is because in both IOVM and centralized network virtualization the CPU is saturated, but the throughput is nearly the same.

Table 2. The Static Core Assignment Schemes

co-locate	Each guest and its corresponding IOVM are on the same core.
separate	Each guest and its corresponding IOVM are on different cores, but a guest shares a core with a different IOVM.
IOVM-affinity	All IOVMs are on the same core, and the guests are each on one of the remaining cores.

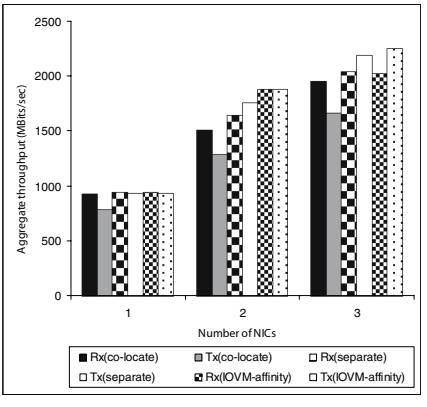

Fig. 8. Throughput Results for Different Static Core Assignment Schemes Using the Multiple Small IOVMs Configuration

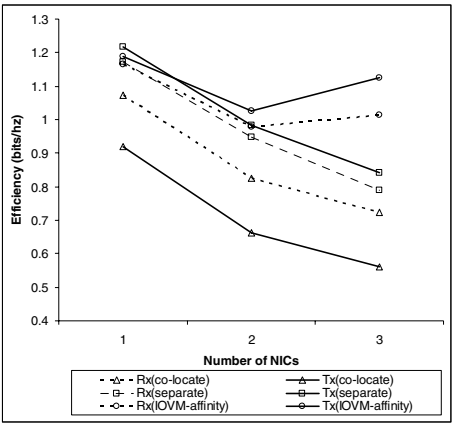

Fig. 9. Efficiency Results for Different Static Core Assignment Schemes Using the Multiple Small IOVMs Configuration

4.3 Multiple Small IOVMs and Static Core Assignment

This section evaluates the performance of using Multiple Small IOVMs (Fig. 4) for network virtualization, e.g., assigning each NIC to a dedicated IOVM. One concern about this configuration is the overhead (e.g., memory and scheduling overhead) of having a large number of simple IOVMs (domains). For example, the platform that we used only has 4 cores, but we have 7 domains in total (3 guests, 3 IOVMs and domain 0) when there are 3 test loads. Obviously some of the domains must share a physical CPU core. Xen 3.0.2 allows the assignment of a domain to a physical core, and this section shows that the way that the IOVMs and the guest domains get assigned is very important to the virtualization performance.

Specifically, we have tried 3 different assignment schemes as shown in **Table 2**. Fig. 8 and Fig. 9 show the evaluation results.

Fig.8 compares the 3 different assignment schemes in terms of throughput. We can see that given a certain number of NICs (test loads), no matter if it is transmit or receive, **co-locate** has the lowest throughput, **IOVM-affinity** has the highest throughput, and **separate** has a throughput in between the other two schemes.

That **co-locate** performs worse than **separate** is somewhat surprising: one would expect the other way around because if a guest and its corresponding IOVM are on the same physical core, there will be no need for InterProcessor Interrupts (IPIs) when packets are sent from the guest to the IOVM. However, the benefit of eliminating such IPIs is offset by a more important factor: the elimination of opportunities for parallelism between the guest and the corresponding IOVM. Specifically, there are pipelines of packets between the guest and the IOVM, so when they are assigned on different cores (as in the **separate** assignment scheme), they can run concurrently so that while the IOVM is processing the n^{th} packet the guest can start sending the $n+1^{th}$ packet. However, when these two domains are assigned on the same core, they lose such opportunities. As a result, there is no pipeline for **co-locate** and the throughput is lower.

But why is the throughput of **IOVM-affinity** the best? We found that a guest needs more CPU cycles than the IOVM to drive the same test load because the guest needs to run the network stack as well as iperf, which does not exist in an IOVM. Thus the bottleneck for CPU occurs in the guest. By assigning the IOVMs on the same core we reserve the most cores possible for the guests (each guest is running on a dedicated core in this case). As a result, the guests are able to drive more network traffic and we get the highest throughput. But this assignment can not be pushed too far, because if too many IOVMs are assigned to a same core, eventually they will run out of CPU cycles so there will be no further increase in throughput. In such cases, the bottleneck will move to the IOVMs.

Fig. 9 compares the 3 different assignment schemes in terms of efficiency. The overall result is the same: **Co-locate** has the worst efficiency, **IOVM-affinity** has the best efficiency, and **separate** has efficiency in between the other two schemes. **Co-locate** is the least efficient because of too much context switching overhead: for example, whenever the guest sends out one packet, the IOVM is immediately woken up to receive it. There are 2 context switches for each packet sent or received. One may argue that **separate** should have the same context switching overhead, but this is not the case, because after the guest sends out one packet, the IOVM on a *different* core may be woken up, but the guest can continue to send another packet without being suspended. The Xen hypervisor is able to deliver packets in batches to the IOVM; running the guest and IOVM on different cores allows multiple packets to be received by the IOVM within one context switch.

Finally, the workload of an IOVM is different from that of a guest, and a guest is more CPU intensive, so mixing them together on the same core (as in **co-locate** and **separate**) may result in more complicated, thus negative, interferences (e.g., cache and TLB misses due to context switches) to an IOVM than putting the homogeneous workloads of the IOVMs on the same core. Therefore, IOVM-affinity scheme is the most efficient.

One more note about Fig. 9 is that the efficiency drops with the increase of the number of NICs for **co-locate** and **separate**, which indicates that they are not scalable schemes. The reason is that each IOVM uses almost the same amount of CPU cycles under these schemes, so that when the number of NICs increases from 1 to 2 for example, the CPU cycles used by the corresponding IOVMs are nearly doubled, but the throughput is much less than doubled (Fig. 8). As a result, the efficiency drops from 1 NIC to 2 NICs. Similarly, efficiency drops off when moving from 2 NICs to 3 NICs. On the other hand, the efficiency for **IOVM-affinity** remains fairly constant as the number of NICS increases and is therefore much more scalable.

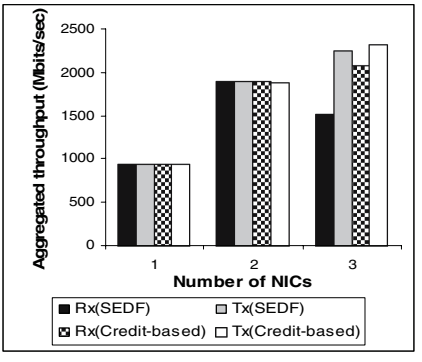

Fig. 10. Throughput Comparison between SEDF and the Credit-based Scheduler

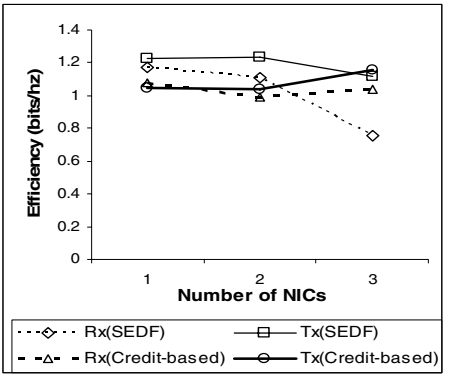

Fig. 11. Efficiency Comparison between SEDF and the Credit-based Scheduler

4.4 Credit-Based Scheduling and IOVM Configuration

Core Scheduling and IOVM Performance

In the evaluation so far we have been using static core schedulers - static in the sense that the CPU core that a domain runs on is fixed during its lifetime. The advantage of such schedulers is simplicity of implementation and reduced overhead of domain migration across cores. However, the drawback is that workload is not balanced across multiple cores, so that the platform resources are not efficiently utilized to achieve better overall performance (e.g. throughput). For example, while the core that an IOVM is running on is saturated, another core where a guest is running on may be idle for 30% of the time. Obviously the IOVM becomes the bottleneck in this case and thus the throughput can not improve, but there are free cycles on the platform that are not used. So if we could give the 30% free cycles to the IOVM, we can mitigate the bottleneck and have higher overall throughput as a result. This is the idea of the credit-based scheduler [13]. In a nutshell, a credit-based scheduler allows a domain to change the core where it runs on dynamically. The credit-based scheduler supports load balancing across the cores and is helpful for better overall performance.

Fig. 10 and Fig. 11 show the throughput and efficiency comparison of a static scheduler (SEDF, or Simple Earliest Deadline First) and the credit-based scheduler, on a Monolithic configuration (Fig. 3). Fig. 10 shows that using credit-based scheduler can achieve equal or higher throughput than using a static scheduler. Especially,

when there are 3 test loads, the overall 'receive' throughput is 2070Mbits/sec for the credit-based scheduler, which is significantly higher than 1500Mbits/sec where the static scheduler is used. The credit-based scheduler does especially well for receive because receive is more CPU-intensive.

Fig. 11 shows that the credit-based scheduler is not as efficient as a static scheduler when there are one or two test loads. This is understandable because moving the domains around incurs overhead, e.g., a moving domain needs to transfer its context to the new core and warm up the cache at the new core. When the CPU is not the bottleneck, this overhead makes the network IOVM less efficient. However, when the network IOVM is saturated (3 NICs), the credit-based scheduler results in better efficiency than the static scheduler, especially for receive.

Credit-based Scheduling and IOVM Configuration

Credit-based scheduling provides a solution to scalable network virtualization. This is because it virtualizes the core resources in a transparent way. In this sub-section we try to find out how to make the best use of this scheduler. We use the 3 configurations mentioned in Section 2.1 as the controlled variable.

In this experiment, 3 test loads are used. We run the experiment using 3 different configurations: Monolithic, Multiple Small IOVMs, and a Hybrid configuration with 2 IOVMs, where the first IOVM is assigned 2 NICs, and the second IOVM is assigned 1 NIC. Fig. 12 and Fig. 13 show the throughput and efficiency for the 3 configurations, respectively.

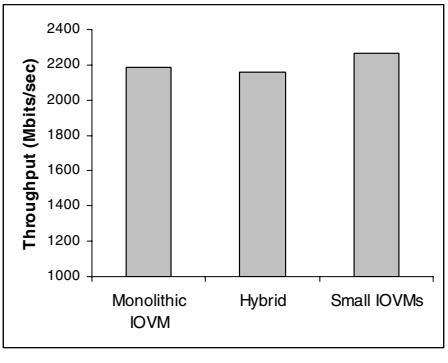

Fig. 12. Throughput of Different IOVM Configurations under the Credit-based Scheduling

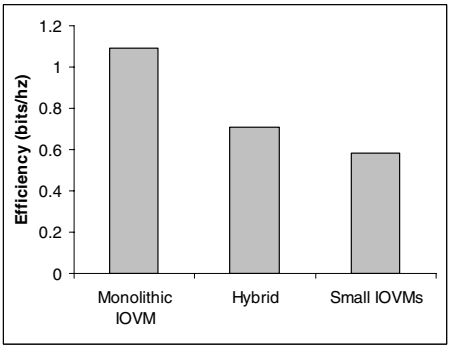

Fig. 13. Efficiency of Different IOVM Configurations under the Credit-based Scheduling

From Fig. 12 and Fig. 13 we can see that the Monolithic IOVM configuration is the most efficient, the Multiple Small IOVMs configuration has the highest throughput but is the least efficient, and the Hybrid configuration has good enough throughput and is more efficient than the Multiple Small IOVMs configuration.

The better efficiency of the Monolithic and Hybrid configurations is due to larger packet batch size in their IOVMs. It turns out that Xen has optimized the network implementation such that the network backend exchanges packets with the frontend in batches to reduce the number of hypervisor calls. The larger the batch size, the more savings in terms of hypervisor calls, and thus the more efficient. At runtime, the batch

size is influenced by workload, or the number of NICs in our case. For example, as Fig. 14 shows, an IOVM with 2 NICs has larger batch size than an IOVM with only 1 NIC, so an IOVM with 2 NICs is more efficient. In our experiment, the Monolithic IOVM has the largest number of NICs (3), each of the Multiple Small IOVMs has the smallest number of NICs (1), and each of the IOVMs in the Hybrid configuration has 1.5 NICs on average, so we get the efficiency result as shown in Fig. 13.

Although the Monolithic IOVM is the most efficient, obviously it can not scale because it only has one core (it uses a uniprocessor kernel, see Section 3), so it can not support higher throughput beyond the core limit. On the other hand, the Multiple Small IOVMs configuration is a scalable solution because it can utilize more cores, but it is the least efficient. The Hybrid configuration is also scalable for reasons similar to the Multiple Small IOVMs configuration, but it is more efficient. So the Hybrid configuration is the best configuration in terms of scalability and efficiency.

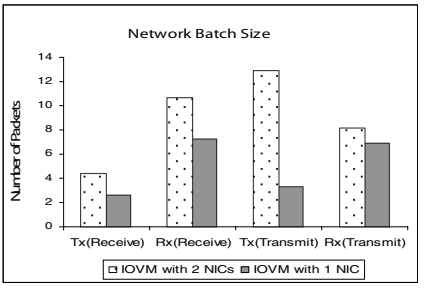

Fig. 14. Network Batch Size

Table 3. Combining Static Core Assignment and Credit-based Scheduler Yields Better Performance

	Throughput (Mbits/sec)	Efficiency (bits/Hz)
Default	2,106	0.72
Core Pinning	2,271	0.82

4.5 Further Improve the Efficiency of the Hybrid IOVM Configuration

The previous section shows that a Hybrid IOVM configuration combined with Credit-based scheduling can give us a scalable network virtualization solution. However, as can be seen from Fig. 13, the efficiency of a Hybrid configuration (0.7bits/Hz) is still not as good as that of a Monolithic configuration (1.1 bits/Hz). This section tries to address this problem. Specifically, we found that combining static core assignment and Credit-based scheduling can give a Hybrid configuration better efficiency.

In this experiment, we have 4 NICs and we use a Hybrid configuration where 2 network IOVMs are assigned 2 NICs each. We first run 4 test loads under the Credit-based scheduling and measure the throughput and efficiency. We call this test result "Default" in **Table 3**. Then we change the core scheduling a little bit: instead of letting the Credit-based scheduler schedule all the domains on the 4 cores, we manually pin the first IOVM to core 3 and the second IOVM to core 2, and let the Credit-based scheduler schedule the other domains on the remaining cores. We do the 4 guest experiment again and put the result in **Table 3** denoted as "Core Pinning". As we compare the two sets of results, we can see that using limited core pinning (or static assignment) results in improved efficiency as well as higher throughput (a pleasant side effect). This result suggests that it is not efficient to move the IOVMs across different cores. So it is more efficient to combine static core assignment (for the IOVMs) and Credit-based scheduling when using a Hybrid configuration for scalable network virtualization.

5 Related Work

I/O virtualization can be carried out in the VMM [3][4], or the host OS [12], in addition to dedicated guest domains [1][5][8][10]. VMM-based I/O virtualization requires nontrivial engineering effort to develop device drivers in the VMM, provides inadequate fault-isolation, and is not flexible for driver optimization. HostOS-based I/O virtualization takes a further step by reusing existing device drivers, but does not support fault isolation and is still inflexible in terms of driver optimization. The IOVM approach supports driver reuse, fault isolation, and flexible driver optimization at the same time.

There has been some related work in improving the performance of I/O virtualization. For example, VMM-bypass IO [9] achieves high performance I/O virtualization by removing the hypervisor and the driver domain from the normal I/O critical path. But this approach heavily relies on the intelligent support provided by the device hardware to ensure isolation and safety, and it does not address the scalability issue. The IOVM approach that we proposed does not rely on such hardware support, and we have put much emphasis on scalability. In another work, Menon [11] proposes three optimizations for high performance network virtualization in Xen: high-level network offload, data copying instead of page remapping, and advanced virtual memory features in the guest OS. These optimizations are orthogonal to what we are doing in this paper, since our main concern is scalability, and incorporating such optimizations into our implementation may further improve the efficiency. Wiegert [14] has explored the scale-up solution by increasing an IOVMs compute resources to improve scalability.

Utility Computing has been a hot research area in recent years. From a service provider point of view, one of the goals is to achieve optimal overall resource utilization. Our work addresses the problem of optimizing the utilization of core (CPU) resources. Concerns about other resources (such as memory and disks) have not been a problem for us, but they can be added into our future work.

6 Conclusions and Future Work

Scalable I/O virtualization is very important in a server consolidation environment. This paper proposes IOVMs as a software solution to this problem. An IOVM is a guest OS dedicated to and optimized for the virtualization of a certain device. IOVMs are good for scalability and flexible for high performance.

The first contribution of this paper is a novel way of using a hybrid configuration of IOVMs to achieve scalable and high-performance I/O virtualization. It makes the scalability and efficiency tradeoff: the scalability is achieved by spawning more IOVMs to utilize more core resources, and the efficiency is achieved by making full use of the core resource within each IOVM.

The second contribution of this paper is a comprehensive set of experiments to evaluate the performance of network IOVMs. They show that IOVMs result in higher throughput and better efficiency compared to the centralized IO virtualization architecture, that a combination of static assignment and credit-based scheduling offers

better efficiency, and that a hybrid configuration of IOVMs is a choice for scalable network virtualization.

Future work: The physical constraint of the multi-core platform nowadays limits the scope of our experiments. For example, if we could have a platform with 32 cores, we can gather more data points to do a more comprehensive analysis. Second, we have studied network virtualization only as a starting point; applying IOVM architecture to other kinds of devices (e.g., storage devices) may further test the validity of IOVM approach. Finally, we plan to use other kinds of work load besides iperf to further evaluate the network IOVMs.

References

1. Abramson, D., Jackson, J., et al.: Intel Virtualization Technology for Directed I/O. Intel Technology Journal 10(03) (2006)
2. Barham, P., et al.: Xen and the Art of Virtualization. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 164–177. ACM Press, New York (2003)
3. Borden, T., Hennessy, J.P., Rymarczyk, J.W.: Multiple operating systems on one processor complex. IBM Systems Journal 28, 104–123 (1989)
4. Bugnion, E., et al.: Disco: Running Commodity Operating Systems on Scalable Multiprocessors. ACM Transactions on Computer Systems 15(4) (1997)
5. Fraser, K., et al.: Safe Hardware Access with the Xen Virtual Machine Monitor. In: OASIS. Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure, Boston, MA (2004)
6. Iperf, http://dast.nlanr.net/Projects/Iperf/
7. Kieffer, M.: Windows Virtualization Architecture, http://download.microsoft.com/download/9/8/f/98f3fe47-dfc3-4e74-92a3-088782200fe7/TWAR05013_WinHEC05.ppt
8. LeVasseur, J., et al.: Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In: OSDI (2004)
9. Liu, J., et al.: High Performance VMM-Bypass I/O in Virtual Machines. In: Proceedings of the USENIX Annual Technical Conference (2006)
10. McAuley, D., Neugebauer, R.: A case for Virtual Channel Processors. In: Proceedings of the ACM SIGCOMM, ACM Press, New York (2003)
11. Menon, A., et al.: Optimizing Network Virtualization in Xen. In: Proceedings of the USENIX'06 Annual Technical Conference (2006)
12. Sugerman, J., et al.: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In: Proceedings of the USENIX Annual Technical Conference (2001)
13. http://wiki.xensource.com/xenwiki/CreditScheduler
14. Wiegert, J., et al.: Challenges for Scalable Networking in a Virtualized Server. In: 16th International Conference on Computer Communications and Networks (2007)