HV²M: A Novel Approach to boost Inter-VM Network Performance for Xen-based HVMs

Kun Cheng^{a,b}, Yuebin Bai^{a,b}, Yongwang Zhao^a, Yao Ma^a, Duo Lu^c,
Yuanfeng Peng^d, Minxuan Zhou^e

^a*School of Computer Science and Engineering,
State Key Laboratory of Software Development Environment,
Beihang University, Beijing 100191, China*

^b*Science and Technology on Information Transmission and Dissemination
in Communication Networks Laboratory,
Shijiazhuang 050081, China*

^c*School of Computing, Informatics, Decision Systems Engineering
Arizona State University, USA*

^d*School of Engineering
University of Pennsylvania, USA*

^e*Department of Computer Science and Engineering
University of California, USA*

Abstract

Despite the rapid development and the wide use, virtualization confronts new challenges on improving network I/O performance. For virtual machines co-existed on a server, inter-VM network performance turns out to be worse than expected. Although many efforts have been dedicated to that issue on paravirtualized (PV) machines, HVMs (Hardware Virtual Machines) get less attention. However, co-resident HVMs are also suffering from low network I/O performance which troubles IaaS cloud infrastructure providers. Such problem is prominent as in the front-back network model several performance bottlenecks appear when inter-VM traffic is being handled.

In this paper, we propose a novel approach and implement a prototype called “HV²M” to optimize inter-HVM network performance, which mainly address the throughput issues. HV²M takes advantages of the existing mechanisms for HVMs provided by Xen and is totally hardware independent. Unlike the existing idea, HV²M pushes data to the receivers without using

Email address: zhaoyw@buaa.edu.cn (Yongwang Zhao)

Preprint submitted to Journal of Systems and Software

December 16, 2015

shared memory buffer to transfer the pending network packets. To evaluate our work, we test and compare network response time, network throughput and CPU utilization with other popular approaches such as SR-IOV. The result shows that the new communication model has great improvements in network delay and throughput with little pressure on Domain-0.

Keywords:

Virtualization, Xen, HVM, Network, I/O, Inter-VM communication

1. Introduction

Virtualization technology improves the utilization of the rich computing resources. After years of researching and developing, hypervisors (virtual machine monitors, also called “VMM”) can nearly make full use of the powerful CPUs and the large memory. However, for today’s most applications, network performance matters more than the computing ability. With more and more applications being integrated onto virtualised computing infrastructures, such problem becomes more urgent. Network bandwidth is an important performance index of virtualised platforms. However, for virtual machines running on a physical server, as the outbound bandwidth is limited by the physical NICs, there’s little room for improvement. But given the fact that those co-resident virtual machines often cooperate with each other to finish a more complex job, there is an urgent need to improve the inter-VM network performance.

Compared with PV machines, HVMs are generally considered to be less efficient but the performance gap is getting narrower. According to many test results, the gap between HVMs’ and PV machines’ scores are getting narrower, such as the result of a test with Amazon EC2 large instance type and Open-source Phoronix test suite [phoronix-test suite.com](http://phoronix-test-suite.com) (2015); openbenchmarking.org (2015) shows. And another advantage is that HVMs are easier to be maintained than PV machines. Given those above, it’s not surprising that HVM and PV-HVM hybrid (also known as “PVHVM”) are receiving more and more attentions. Besides, for a large number of Windows-based applications, HVMs are their only choice. Under such a circumstance, there is a need to boost the inter-HVM network performance as: 1) cloud infrastructures highly value it as most applications hosted in the cloud are I/O sensitive which have requirements on both latency and bandwidth; 2) most clouds are more prone to provide HVMs to users, however almost all

existing solutions (inter-VM network performance optimizations) are for PV machines. Thus, HVMs are more and more popular with many IaaS service providers, but in the meanwhile “how to overcome the inter-HVM network performance bottlenecks” also troubles them, of which low throughput is the first to be blamed.

The root cause of such performance issues lies in the virtualised network model. On Xen-based platforms, network traffic path of an HVM is shown in Figure 1. For an HVM, the path starts from its virtual NIC (net-front), then data is transmitted to net-back in Domain-0 by a complex way (which we will explain in Section 3). A net-back links to a software bridge through a virtual switch, and the bridge is attached to Domain-0’s NIC (eth0). That is an example of the classic split driver way to handle I/O operations in Xen. In this bridged network mode, routing is managed by xenbr0 (the software bridge) and net-fronts. For networking between two co-resident VMs, communication gets Domain-0 involved in, which extends the path and makes Domain-0 a performance bottleneck. Considering the instantiation of multiple co-resident VMs on a single server, the burden of switching frames among them cannot be ignored. As a matter of fact, lots of literature [Cherkasova and Gardner \(2005\)](#); [Menon et al. \(2005\)](#) have found out that the switching can greatly increase Domain-0’s CPU load. During the last decade, a lot of solutions were brought up for PV machines (as in a long time they are the best and only option), from which HVMs can’t benefit.

Figure 1: General HVM’s networking pattern on Xen

SR-IOV (Single Root I/O Virtualization) [pcisig.com \(2010\)](#) is another

popular option as it standardizes a way of virtualising and sharing PCIe devices among multiple VMs. It bypasses hypervisor so VMs can directly access physical devices through virtual functions (VFs). Each VF connects to the Virtual Ethernet Bridge (VEB) inside the device, which works as a ‘switch’. In that way SR-IOV moves the virtual switch out of the hypervisor but into the device. However, our experiments show that NIC’s internal switching promotes inter-VF network performance little. That is mainly because the spawned VFs’ ability are constrained by the hardware and the traffic path (dash lines) is no shorter than Xen’s default software-shared way (solid lines) as illustrated in Figure 2, which is also proved by our experiments.

Figure 2: SR-IOV and split-driver model

In this paper we design and implement a novel software approach called “High performance VM-VM coMmunication (HV²M)” to address the inter-HVM network performance issues. HV²M aims at the key points in the network pattern of Xen. As mentioned above, the existing bottleneck includes long path and Domain-0’s switching burden, etc. HV²M bypasses Domain-0 and reduces data copies compared with existing solutions, which allows two co-resident HVMs to transmit data directly. Unlike most of the existing solutions, HV²M only copy data once between two participating VMs during the transmission, which saves memory space (as fewer shared

memory is needed only for data transmission management) and cuts down data operations' cost. What's more, HV²M is implemented for both Linux and Windows HVMs. Although this paper focuses on the inter-HVM traffic, the basic idea can be migrated onto PV machines as the (similar) features it used are also provided by Xen's paravirtualization technology or KVM platforms.

The main contributions of this paper include:

- A detailed analysis about inter-HVM traffic bottlenecks on Xen.
- A hardware-independent approach to boost inter-HVM network performance which eliminates most of the existing bottlenecks.
- New data transferring approach which reduces data copies compared with shared-memory solutions.

The rest of the paper is organized as follows. Section 2 introduces the related work in recent years. In section 3 we review the Xen's networking model and the motivation of this work. The Design and implementation of HV²M are presented respectfully in Section 4 and Section 5. The following section gives experiments and results analysis. In the final section we conclude this paper.

2. Related Work

Virtualization systems' network performance optimization remains fascinating for over decades. Some researchers try to solve the problem by improving the scheduling framework or resource control. Zhibao Chang et al. [Chang et al. \(2012\)](#) study the impact of Xen's credit scheduling on network I/O and optimize it for I/O-intensive domains, but they don't solve the problem fundamentally. Because of lacking an access control mechanism, if there're two many I/O-intensive jobs/domains in a single run queue, the network performance of those domains could still be bad.

Based on the existing functions and features of Xen, most researchers use shared-memory, grant table and event channel, etc., to address the problem for PV machines. Among them XenSocket [Zhang et al. \(2007\)](#), IVC (Inter Virtual Machine Communication) [Liao et al. \(2008\)](#), XWay [Kim et al. \(2008\)](#), XenLoop [Wang et al. \(2008\)](#) and MMNet [Burtsev et al. \(2009\)](#) are outstanding. XenSocket provides a one-way communication bypassing the

protocol stack, which uses the I/O ring as the transmission media. In XenSocket, a sender needs to ask for memory pages and event channel, then shares the pages with the receiving domain after notifying the channel port to the latter. Until the sender finishes writing the data and updating the pointer, then the receiver can read the data after getting a notification through the event channel. XenSocket provides a prototype that requires manually passing parameters like port number and both domains' id, etc. The similar idea can be found in IVC and Xway, etc., which are all based on the shared-memory approach.

IVC works in a distributed way which allows VMs on the same platform to transmit data through a piece of shared memory. It provides APIs for applications, which are implemented by modifying MVAPICH2 [Menon et al. \(2006\)](#) to MVAPICH2-ivc. IVC uses shared memory and event channel provided by Xen. It consists of a set of APIs (user mode) and a driver (kernel mode). Applications using the API can create event channels after negotiating. Then the sender applies for memory pages, and the driver manages authorization and memory mapping. In order to use IVC, applications have to be reprogrammed and it only works under Linux domains.

XWay adds a new layer to the network stack, which analyzes TCP packets to find those whose destinations are on the same platform. It uses shared-memory, page grant access operations and event channel, which is similar to XenSocket and IVC. In addition, XWay implements a complete set of TCP operations. To use XWay the Linux kernel has to be modified, and the patched kernel only supports TCP protocol.

XenLoop works between the link layer and the network layer to provide a transparent direct 2-way communication for domains on the same physical machine without involving Domain-0. It provides kernel modules (which can be dynamically loaded) to cooperate only with PV guest kernels so no further kernel code modification is required. XenLoop uses two pre-mapped FIFO rings between a pair of participating VMs to send and receive data, which is essentially another form of using shared memory for transmission. XenLoop relies on netfilter to deal with different network protocols' packets and currently it only supports IPv4 protocol. In such a design, FIFO data buffer's size also has an impact on the transmission, e.g. packets have to wait if the ring is full or they have to be transferred through the original front-backend path when packet size exceeds the FIFO size, where the size could be vital to the optimization.

MMNet(or Fido) uses a more radical shared-memory approach as it map-

s the entire sender’s memory space to the receiver’s to reduce data copies, while others mentioned above use only a small piece of shared memory as data buffer or map only the FIFO data buffer pages. Upon such design, the receiver can easily access the pending data, which could solve throughput scaling issue caused by the small shared memory segments. MMNet provides a kernel module which contains a routing table for data forwarding and redirection. It uses Scatter-Gather Array-SG to convert the sender’s address pointer and data size to the receiver’s format so that the latter can use them to read data from its’ memory space, which is done by the module MMNet provided. MMNet shows advantages over XenLoop in reducing the latency, but it breaks the isolation (although just in a read-only way), which requires an administrator to judge if both domains can trust each other and transmit data.

For HVMs there’re fewer researches on VM-VM traffic optimization. Among those ideas IV²COMM [Bai et al. \(2013\)](#) is a representative. It (or to say most of existing inter-HVM solutions) borrows some idea from XenSocket, etc. IV²COMM also uses shared-memory queues to complete data transmission, which requires the data to be copied twice between participating domains.

Sugerman et al. study the virtualization and performance issues on VMware Workstation [Sugerman et al. \(2001\)](#). They think IRQ processing, world switch (between VMM and host) and inability of VM application and VMM to distinguish a hardware interrupt which produces an event for the VM is the key to optimize virtual NIC performance. Many later works are enlightened by their forecasted optimization ideas, such as modifying both the guest and the host OS to avoid using inefficient virtualization instructions and optimize network stack, guest driver protocol and how Linux kernel handles `sk_buff` and the most important thing is bypassing the host OS to reduce world switch and promote performance.

SR-IOV (Single Root I/O virtualization) [Dong et al. \(2010\)](#) allows packets to be sent from NIC to VM. It helps realize the network device virtualization so that a domain can use the “virtual function” itself without getting Domain0 involved. SR-IOV creates receive queues for externally received frames and eliminates the packet receiving interrupt for the receiver. However, inter-VM traffic on a single server is not specially considered by SR-IOV, which still uses a software switch to do the trick [intel.com \(2009\)](#).

3. Motivation and Backgrounds

The default network approach implemented in Xen works in a complex way [Santos et al. \(2008\)](#). As shown in Figure 1, a net-front uses shared I/O rings to communicate with its corresponding net-back. In such case two I/O rings are required, as one ring only applies to sending or receiving data. During the communication, when a new message (a request or response) has been put in the ring, the producer (a net-front or net-back) has to trigger a virtual interrupt on the other domain to notify the new event with the help of Xen’s event notification mechanism. To help receive the data, Xen also provides a page-grant mechanism, which enables another domain (Domain-0 or another VM) to access I/O buffers in a guest domain (VM)’s memory. By leveraging such mechanisms, data transmission between a guest domain and Domain-0 can be described as follows. A VM first creates a page grant reference containing access to a page (containing data) and forwards the reference to the corresponding net-back in Domain-0. Domain-0 can then use the reference to access the guest memory pages by invoking a hypercall. For VM’s sending requests, Domain-0 first invokes a hypercall to map the VM’s pages into its address space before sending the request through the bridge to the physical device, then Domain-0 could directly access the pending data. After the physical device finishes sending, the net-front will be notified and then revokes the page grant. For receiving requests, net-front puts I/O ring page grant references to the receiving I/O ring. When the net-back receives a packet from the bridge, it retrieves a posted grant from the I/O ring and invokes a grant copy hypercall to copy the packet directly into the guest memory page. Finally, the net-back sends a response through the receiving I/O ring which notifies the guest that a packet is available. When two VMs are performing an inter-VM communication, both have to exchange data with Domain-0 first, then the data will be filtered and switched by the virtual bridge without sending them to the physical device. However, the whole process involves many page grant operations as both VMs have to exchange data with Domain-0, which adds additional overhead compared with non-virtualised I/O handling.

With the support of Intel VT technology, HVMs can run in non-root mode while Domain-0 runs in root mode. This is because Intel VT introduces new instruction set called Virtual Machine eXtensions (VMX). VMX consists of a set of operations, of which only hypervisor can use the root operations (root mode) but HVMs can only carry out the non-root ones (non-root mode). So

when a HVM needs to access the hardware resources (e.g. to complete I/O operations), it needs to trigger a VMX transition. The transition includes VM Entry (root to non-root) and VM Exit (non-root to root). In this scene, when a VM Exit occurs, the current running environment of the HVM is stored in a virtual machine control structure (VMCS). Then Domain-0 is scheduled in to handle the real request. After it's done, the HVM resumes running (VM Entry). Compared with actual I/O handling cost, the VMX transition expense is huge [Bai et al. \(2013\)](#). Specifically, When two VMs are performing an inter-VM communication, both VMs have to exchange data with Domain-0 first, then the data will be filtered and switched by the virtual bridge without sending them to the physical device. However, the whole process involves many page grant operations as both VMs have to exchange data with Domain-0, which frequently triggers VMX transition and adds additional overhead.

Now we can conclude here that in the existing inter-HVM traffic, the probable software optimization could happen in:

- Long path. Inter-VM traffic has to go into Domain-0 and through net-backs and software switch.
- Switching burden. Switching inter-VM frames in privileged domain will increase Domain-0's CPU load and cause bad performance when multiple HVMs are dealing with VM-VM traffic on the same server.
- Shared-memory approach which is adopted by existing optimizations. Coping data through shared-memory buffers sacrifices in both time and space. For a pair of sender and receiver, it requires two buffers to complete a 2-way communications. What's more, a single transmission requires data to be copied in and out of a buffer.

The basic idea of optimizing inter-VM traffic is letting two VMs transmit data directly through their net-fronts or other proposed modules. In which way there's no need to do frame switching and that frees Domain-0. In the meantime such approach can shorten the path as it bypasses both net-backs and the software switch. Compared with what we learn from [Figure 1](#), the optimized path looks like what's shown in [Figure 3](#). The optimized solutions usually use two queues for sending and receiving respectively to complete a 2-way communication. Either queue is a block of shared-memory which is managed as a circular queue and operates on the producer-consumer model.

Pending data is first copied into a queue and waiting for the other side to copy it out. Such design is based on the experiment and analysis that per-packet page flipping in Xen’s default network virtualization is a large source of inefficiency [Zhang et al. \(2007\)](#); [Kim et al. \(2008\)](#).

Figure 3: The existing inter-VM traffic optimization on Xen

Although such approach bypasses Domain-0 and software switch which eliminates some bottlenecks listed above, there come new issues. First, adding a new module to a guest domain (virtual machine) increases its complexity and memory use (fortunately not every solutions adds new modules), some solutions even bypass the entire network stack (transmission in a new protocol). In the meantime, managing new modules and net-fronts also introduces new burden, such as their cooperation issue, i.e. deciding when either should work. Second, additional memory is required to organize queues or buffers for data copying. Both the issues require further optimization.

This research is conducted to address those new issues and to boost the inter-HVMs network performance. At the beginning stage, it aims mainly at the network throughput, and currently neither QoS nor load balancing is considered. Besides, further expectations of the new inter-VM communication approach for HVMs on Xen include:

- Reducing known overheads. This includes a shorter path, no switching burden for Domain-0 and reducing data copying times.
- Transparency. Data handling should happen in the lower level which could retain the upper network stack and applications unchanged.

- Easy to use. It should require as few parameters as possible from the administrator of a Data Center.

4. The design of HV²M

Before the further discussion, the flexible trust model adopted by HV²M should be introduced. As in enterprise server virtualization, most software components (e.g. applications, VMs, etc.) are pre-tested and considered to be qualified. Therefore, the following assumptions can be accepted. First, the underlying VMM and its operations can be trusted, which indicates that invoking hypercalls to let VMM access a VM's data is acceptable. Second, for VMs collaborating with each other to complete a job, they're assumed to be generating/receiving non-malicious data, which means those network packet data is trusted. With such basic assumptions, VMM and the inter-HVM network data are supposed to be credible so the security risks are cut down to the bottom level.

The basic inter-VM data delivery model works in a "receiver-active" way. In such a solution, the sender tells the receiver where the data is, such as in a memory buffer (shared memory way) or somewhere in the sender's memory pages (mapping address space approaches, e.g. mmnet), then it's the receiver's responsibility to "pull" the data. As an analogy, the clerk (sender) of a supermarket tells the customer (receiver) where the goods are, and customers have to get what they want from shelves (receiving data). In a shared memory way, either side has to allocate a dedicated memory region (or hypervisor offers a small block of memory) to store the pending data. First senders copy data into the shared memory region, then notify the receivers to copy it out, which obviously requires more space and increases data operations. Shared memory regions are usually managed under producer-consumer model, any change can be informed through Xen's event mechanism, which hardly requires additional coordination between both sides. Compared with such approach, mapping the address space reduces data copy operations and requires no shared memory space, which leads to a relatively higher performance but introduces security concerns.

Another choice is to design a "data express". This approach borrows the idea of express delivery: Receivers tell the courier where to put the packages and the courier will do the delivery. In HV²M, every sender is a courier which is responsible for delivering the data to a given destination announced by a corresponding receiver. In another word, senders "push"

Figure 4: Data transmission. The upper half describes how data is “pulled” by a receiver, the lower half shows data is pushed to a receiver. Solid directed lines indicate data transmitted by sender, dashed ones indicate operations done by receiver.

data to their corresponding receivers. As shown in Figure 4, pushing data obviously reduces operations required by both sides compared with those shared memory approaches. To begin a data express, the receiver should first tell the sender where to put the data (destination) and grant the sender certain privileges to access the “destination”. Then the sender is able to deliver the data to the destination. However, those data is in disorder from the receiver’s view, so the sender has to describe the data (e.g. size, address) to the receiver. To achieve that, the receiver has to allocate pages before the transmission, and use Xen’s page grant operations to grant the sender the access privilege. However, memory mapping is not encouraged and should be reduced (or even not used) between both of HVMs as, first mapping pages is complex which requires additional jobs such as lock and synchronization, second it’s going to raise security concerns (the more mapping pages are used, the bigger the security concern will be). Hence, although the sender is allowed to copy data to the destination pages, it has to invoke hypercalls to complete data copy, which is actually done by the hypervisor. Once the receiver gets the data, it can restore each frame according to the description, which includes the start address (offset in the page) the size (length) of each data frame.

Description message is however delivered differently than physical data. They are transmitted in a shared memory way as, first a description message' size is so smaller than the data frame it described that invoking hypercalls to push them is not economical, and experiment part also shows pushing small-sized data is inefficient. Second, delivering the description message can tolerate small latency as receivers can accumulate the data frames and begin analysis a little later. Third, as they are short so only a small region of shared memory is needed. Also, Xen's event mechanism can be used to achieve an asynchronous communication in order to reduce the overhead introduced by a periodical polling. Once there is data to be sent, the sender will notify the receiver to be ready, which also reduce potential polling latency.

The complete procedure of pushing a data frame can be described as follows. First, the receiver allocates pages for data transmission and grants foreign access to them (by `grant_foreign_access_ref`). Those pages will be marked as "allocated but not used" in an FIFO queue, which has to be shared between both sides so that it can be accessed by the sender while managed and maintained well. Second, the sender chooses a page reference (head of the queue) and constructs a page grant copy operation to put data into the page, in which way the data is pushed to the receiver. Third, the sender puts the data offset, data length and page id (position in the queue), etc., in a description message and delivers it through shared-memory ring. If a packet (from IP layer) is too long then it has to be split into fragments, then the sender has to repeatedly invoke hypercalls to push each fragment and generate descriptions for them. Finally, the sender can notify the receiver "there is pending data to be received" through the event mechanism and the job on sending side is done. Then the receiver restores each packet according to its description (fragments has to be dealt repeatedly). After that it has to update the shared FIFO reference queue accordingly. If the queue is out of free pages, then the receiver allocates more as supplementary and also grants foreign access to them so they can be used in the following transmission. Finally the receiver delivers the packet to the upper network stack (IP layer). Such mechanism takes some hints from DMA (Direct Memory Access), in which there will be pre-allocated pages (buffer) for physical NIC driver to complete DMA transmission, and more pages will be allocated for further transmission.

With the above design, HV²M is able to push data from a guest domain to another. Therefore a two-way communication requires two "data express", whose only difference is the direction as receiving can be seen as sending from

the opposite side. Details of the prototype implementation is introduced in the following section.

5. The implementation of HV²M

5.1. The Structure of HV²M

The basic structure of HV²M takes general ideas of existing works, i.e. bypassing Domain-0 and letting two VMs handle network traffic directly. As shown in Figure 5, HV²M consists of frontends (in HVMs) and a backend (in Domain-0). When two co-resident VMs are going to communicate with each other, network traffic would be sent and received directly through their frontends. HV²M backend automatically gathers each domains' information after their frontends wake up. During the data transmission, backend helps two domains setup a message tunnel between both domains' frontends in order to notify each other about the description message (e.g. size) of the data. HV²M frontends are implemented in Xen's PV driver net-fronts, which won't conflict when both are working.

Figure 5: The framework of HV²M

After a new frontend wake up (it usually happens when a VM starts), it automatically registers itself to the netback. The registration sends the

HVM's domain ID and MAC address to the netback, which is started by the guest domain and the netback just waits for new comers. When two co-resident HVMs are going to launch an inter-VM communication, netback helps them establish an event channel and exchange page grant info, etc. Before two HVMs can transmit real traffic, both have to negotiate and prepare themselves. The sender notifies the receiver about the pending data's description message, and the latter will prepare data receiving such as allocate data structures. The event channel is constructed using a block of shared memory. As the description messages are short and the event channel also requires a small amount of memory, neither will introduce too much space overhead.

Currently HV²M provides a command tool for administrators to setup up traffic tunnel between two VMs. All the parameters required are only the two VM's domain IDs. The command alerts the backend to prepare tunnel establishment, which notifies both domains' frontends to begin the real process. It's obvious that HV²M works in a distributed way and the backend serves as an agent which listens administrators' commands and assigns tasks for the related frontends to accomplish.

5.2. Data Transferring Approach

There're three data path in HV²M. First, the I/O rings between each DomainU and Domain0, which are for management purpose and will be introduced in the later subsection. Second, data description path which is used to deliver description message of pending network data. Third, "data express" which is used to transfer network data with the help of descriptions. In this part the implementation of data description path and data express is going to be introduced.

HV²M only uses a small amount of memory between two VMs to pass description messages in order to assist data transmissions. In the prototype, "tunnel" is built to deliver the data descriptions and to share the (granted) page reference queue info, which requires a page from both sides. Then both sides map each other's page into their own memory, like we discussed in the design part (Section 4), transferring those messages in a shared-memory by page mapping is a balance of efficiency and overhead.

Description path. A tunnel contains two I/O rings, either of which is organized as a circular queue and can only be used to complete a one-way message pass. Therefore two rings are required to accomplish a 2-way communication. When passing data descriptions through a ring, both sides rely

on Xen’s event channel notification mechanism to notify the opposite domain. As shown in Figure 6, a sender put the data descriptions into a receiver’s ring and notify the receiver to read them out through the event channel. The ring is managed according to the producer-consumer model. There’re four pointers, namely req_prod (request producer), req_cons (request consumer), res_prod (response producer) and res_cons (response consumer). A sender adds a new request into the queue (request producer), which updates the req_prod shared by both sides; a receiver picks up a pending request (request consumer) and updates the req_cons, then puts the response message into the queue (response producer) and updates res_prod shared by both sides; the sender, now as a response consumer, updates its own res_cons after handling a response message. Each description is 12 bytes which contains: type (1 byte), indicating it’s a description of pending data; flag (1 byte), telling if the described packet are fragmented or not; offset (2 byte), pointing out data’s start position in the given page; length (2 byte), indicating the length (how many bytes) of the data; id, which means the position of this described data in the data buffer; page reference, which is reserved for further “pull” method (receiver copies data actively, which is not used in this work).

Figure 6: Descriptions transferring model

Data buffer. The purpose of data buffer design is to simplify the operation and to avoid complex synchronization. Sender uses pre-allocated pages from receiver as data buffer and put data in it, then it tells the receiver which page is used and where the data is (by using a description message), after receiver gets all the data, it releases all the previously used pages or

allocate more if the buffer is full, where the whole process shares something in common with a DMA transmission.

In order to deliver network data correctly, both sides have to learn which pages on the opposite side is accessible, which is done by sharing the pre-allocated page references mentioned in the design part (Section 4). The page reference queue resides on the pre-mapped page with the description path from either side. At either side, the pre-allocated pages are organized in a linked list. The page grant info (references) of these pages is in an FIFO queue which is initialized when both sides' netfronts are establishing the tunnel (procedures described in the following subsection) with the help of the HV²M's netback in Domain-0. As mentioned above, HV²M's netback coordinates and manages all the frontends. When two guest domains are going to launch an inter-HVM communication, either's frontend will tell the netback about the page allocated for data transmission and their grant access info. Then netback tells those info to their opposite sides to complete the info exchange. With the page grant references, both sides are able to construct the FIFO reference queue which provided by the opposite side. Note that the actual page list on either side may be different because it depends on how either side allocates the pages. After tunnels establishment, as both sides have mapped the other's shared page, this reference queue can be accessed by both sides. If the receiver allocates more pages as a complementary to the existing data buffer, it can update the queue and the sender will know by checking the available references. In such a mechanism, the reference queue is implemented by using an array, the index serves as the id (used in the description message), and the initial value is the page's granted reference. After the sender pushes data to a certain page, it will put the page's id (index) in the description so receiver will know which page is used to store the data and whether the buffer is full or not. Then the receiver can decide if it should allocate more pages, grant their references to the sender and update the FIFO reference queue.

Data Transmission. As shown in Figure 7, data is “pushed” by the sender to the receiver. First, the receiver grants the sender access to copy data to its pre-allocated pages, which is stored in the FIFO reference queue. Then the sender chooses the head element of the queue to construct a page granted copy operation and writes the data description to the sending I/O ring, which includes the starting address and the amount of pending packets. In Linux domain we use `sk_buff` to present a packet, while in Windows we use `NDIS_PACKET`. Multiple ring units may be required because one unit

only describes one frame, and a packet may be fragmented due to its length exceeds an MTU's size. After that, the sender invokes a hypercall to complete the data copy to the receiver's pre-granted pages and then it will notify the receiver through the event channel. Once the receiver gets the notification, it immediately reads the descriptions out and initializes corresponding data structures. In that way the sender is able to "push" data to the receiver. After the above procedure, the receiver reassembles the packets from the frames in its pre-granted pages according to the descriptions and delivers them to the upper network protocol. Receiving a packet follows the same procedure, during which the packets are pushed by the receiver to the sender's memory. However, in order to avoid repeatedly invoking hypercalls, HV²M aggregates the data operations and always tries to push a page-sized data.

HV²M is implemented in the link layer so it can support all the protocols in the upper network stack. Network frames are first filtered in frontends according to their destinations' MAC addresses. To support efficient frames forwarding, HV²M use a MAC table. The MAC table is implemented as a hash table. Given the fact that VMs' MAC addresses are generated according to the physical NIC's MAC address, and the key difference between them lies in the last few bytes. For addresses stored in 6-byte format, we can use $(mac[3] \times 131 + mac[4]) \times 131 + mac[5]$ to calculate a unique key, the key's value is a reference pointed to another VM. If the frames are being transferred to another co-resident domain, then they will be delivered through HV²M, otherwise they will flow in Xen's default way.

When HV²M are transmitting data, it's the hypervisor who will do the real transmission (push data to the other side) as either side has to invoke hypercalls to complete network data copy. In that way, with given page access granted, senders can only access to the pre-allocated clean pages and the hypervisor can ensure the access cannot exceed the "boundary", which helps reduce potential security risks. That is also the biggest difference between HV²M and MMNet, as the former only allows senders to access certain pages and data transmission is done by hypervisor, but the latter maps the sender's memory space to the receiver's and let the receiver access the whole memory, which could lead to malicious unauthorized data copying. However, compared with MMNet or Xen's split-driver model, HV²M has to copy data once (data pushing) from the sender to the receiving VM by invoking a page grant copy hypercall. Although data copy operations introduced overheads, it could reduce security risks. Besides, compared with solutions using shared-memory buffers which copy data twice during the transmission, data copies

Figure 7: Data transferring model

have been reduced by half.

5.3. The Management of Inter-HVM Communications

As HV²M's control plane, the management of inter-HVM communication is to coordinate the setup between HVMs and make the necessary preparations. An established “tunnel” means two HVMs are allowed to transfer data with the help of HV²M. Tunnels are used to synchronize both sides' operations but not as transmission buffers for network data. Currently HV²M supports basic tunnel management, namely establishing a new channel, revoking or resetting an existing channel. Executing those functions requires relevant domains' MAC addresses, IDs, domain type and event channels' id, etc. The collection is done by HV²M's frontends and backends automatically and the information is stored in XenStore. XenStore is a storage space shared between domains which is something like a database.

To create a tunnel, first we use **setup_tunnel** command to notify the both domains to prepare for data transmission. After they finish allocating the necessary resources, they write the outputs to their HV²M/domain0 keys in XenStore (including establishment state and process). Once Domain0 gets those info from XenStore, including grant-table items about their commu-

Figure 8: Procedure of establishing a tunnel

nication resources and the port number of their event channel, it will write them to either domain's HV²M/domain0 key and notify both VMs that establishment is allowed to be carried on. Then both VMs create the event channel and I/O rings (each I/O ring requires a page from DomainU). After that they notify Domain-0 again. Domain-0 will send the notification the 3rd time after both sides confirm the establishment. Finally after both VMs get the final notification, they can start to transfer data. The whole procedure looks like a three-way handshake as shown in Figure 8. The two VMs here, one is an active side which is active in creating the event channel, and the other is a passive side which is notified the channel port and cooperates with the active side to create the channel, and a control plane is mainly a shared I/O ring we mentioned above. Note that here memory mapping is only used to create shared memory for I/O rings, so both domains can negotiate with each other.

Terminating an existing tunnel is much similar to creating one. First we use *destroy_tunnel* command to notify both the sender and receiver. After that, both VMs stop any operations in the tunnel and notify Domain-0 through event channel. Then Domain-0 alerts them again to terminate communication. After receiving that notification both sides revoke the memory mapping (I/O ring), terminate event channel and notify Domain-0. Next

Figure 9: Procedure of destroying a tunnel

Domain-0 will notify both sides the final time to destroy the tunnel. At last, after receiving the final notification, either side will release the resources, including the control plane and event channel. The whole procedure is shown in Figure 9.

6. Evaluation

We implement and test HV²M for both Linux and Windows domains. In this section first the test environment will be introduced, then the experiment and result analysis will be presented.

6.1. Experiment Setup

The hardware of the tests is shown in Table 1. We used as many domains as we could in the test so the comparison result could show the bottleneck and the improvement. The test content and the tools we used are listed in Table 2. CPU utilization can be monitored by using the *xm top* command provided by Xen. Netperf is a network performance benchmark, and Nttcp is used to measure network performance in Windows. During the experiment, we use Xen 4.1.2 and Debian 6.0.5-amd64 (kernel version 2.6.32) as Domain-0, which is allocated with 4 VCPUs and 4GB memory. Each HVM domain uses

Table 1: Hardware

Item	Type	Quantity
CPU	Intel Xeon E5620	2
Mother Board	Intel Server Board S5500HCV	1
Adapter	Intel I350-T2 (100/1000 Mbps)	1
Memory	DDR3 1333	16 GB

Table 2: Test tools

Item	Content	Tools
1	RTT	ping
2	CPU utilization of Domain0	xm
3	Network throughput	netperf(Linux)
		nttcp(Windows)

512MB memory and 1 VCPU. We use Debian 5.0.8-amd64 (kernel version 2.6.32) for Linux guest domains and Windows XP for Windows domains.

During the test between Linux domains, we setup three scenarios. First we use two HVMs (a sender and a receiver), then we increase the number to 4 (2 pairs of senders and receivers) and finally to 6 (3 pairs of senders and receivers) in order to test the performance while the load is heavy. In those tests, we compare HV²M with PV-on-HVM (Xen’s pvdrivers wiki.xen.org (2014, 2015)) and SR-IOV enabled Intel I350-T2 network adapter. Each VM is assigned with a VF in both 100Mb and 1000Mb LANs for SR-IOV performance tests.

6.2. Measurement with Linux Domains

First we test the RTT improvement using only two HVMs. The RTT result can be achieved by using ping command and calculate the average of echo request/reply time comes from multiple tests, which is shown in Figure 10. The test results show that HV²M reduces the round-trip time beyond 77.5% compared with PV-on-HVM and 31.3% compared with SR-IOV. Figure 10 also shows SR-IOV achieves a steady performance, HV²M does worse and PV-on-HVM does the worst.

Then we use Netperf to test the network throughput. Here we use 8 different message sizes as listed in Table 3, the test length and message size can be set with -l and -m options (using default values for other options) and the test case which Netperf uses is TCP.STREAM. In each scenario (e.g. a

Figure 10: RTT test result

pair of HVMs, etc.) the test lasts for 60 seconds and the message size ranges from 64 bytes to 16K bytes. We use default values for the receiving socket size (87380 bytes) and sending socket size (16384 bytes). For each message size, it is tested 5 times and the average throughput will be used as the final result.

In each test case, we use the total throughput which all senders achieve to represents each solutions' performance. For example, In figure 12 we add the two senders' score as the final throughput, such as HV²M achieve 13899.9

Table 3: Message sizes of test cases

Item	Test length (secs)	Send message size (bytes)
1	60	64
2	60	256
3	60	512
4	60	1024
5	60	2048
6	60	4096
7	60	8192
8	60	16384

Mbps when transmitting a 4KB message while PV-on-HVM achieves 8945.76 Mbps.

Figure 11: Total TCP throughput (2 HVMs)

Figure 11 shows the test result of comparing the throughput between HV²M, PV-on-HVM and SR-IOV when a pair of domains are used. In this situation PV-on-HVM shows advantages over HV²M in most cases. The biggest average throughput promotion is 37% in 64B message size, the worst case is in 8192B size with 10.42%, on average the HV²M is 17% beyond PV-on-HVM. SR-IOV shows a steady performance as the throughput is around 94 Mbps (under 100Mbps mode) and 940 Mbps (under 1000Mbps). The results will be analyzed later.

When 2 pairs of senders and receivers are used in the test, HV²M shows outstanding performance in TCP throughput. Figure 12 shows that HV²M outperform PV-on-HVM over 28% on average.

Figure 13 shows the test result and the throughput comparison when 3 pairs of domains are used. In this situation HV²M is in ahead of PV-on-HVM. The biggest throughput promotion is 135.3% in 4KB message size, the worst case is in 64B size with -31.5%. In general HV²M's throughput is in 79.8% ahead of PV-on-HVM, let alone the advantage over inter-VF communication.

At the same time, we use *xm top* (Xentop) command to monitor the

Figure 12: Total TCP throughput (4 HVMs)

Figure 13: Total TCP throughput (6 HVMs)

CPU utilization of Domain-0 while testing the throughput. As shown in Figure 14, which is collected when two domains are used for test. While HV²M is running, it hardly consumes Domain-0's CPU resources and keeps

Figure 14: CPU utilization of Domain-0

a low utilization ranges from 1.20% - 1.60%. SR-IOV takes 1.5% - 3.1% and PV-on-HVM takes about 99% - 101%. Note that in Xentop the utilization spectrum is not 0% - 100%. According to the number of a domain's VCPUs, it ranges from 0% to $n \times 100\%$, where n is the VCPU quantity (in our test Domain-0's upper limit is 400%). During the whole test, Domain-0's CPU utilization increase 50% when a new domain start to handling network I/O with PV-on-HVM. However, when HV²M is being tested, the CPU load never exceeds 3%, while SR-IOV occasionally shows a burst increase to about 80%.

6.3. Result Analysis

In situations where fewer VMs or smaller message size is used (e.g. the RTT results, 64-byte messages or throughput between 2 HVMs), HV²M performs worse than PV-on-HVM. That's mainly because HV²M invokes more hypercalls in order to push data and each participating guest domain has to bear data transferring burden which is previously done by Domain-0. PV-on-HVM works like we discussed in Section 3, it copies data after memory space is mapped, from this aspect the hypercall overhead of HV²M and it should be even. What's more, in PV-on-HVM, Domain-0 is only responsible for transferring data, while both sides are dealing with network stack such as invoking socket API. However, HV²M needs both sides to send data de-

scriptions, prepare necessary data structures and they have to transfer data by invoking hypercalls on their own (not Domain-0). Given that both the approaches have to invoke hypercalls to create shared I/O rings which causes VMX transitions, preparation and data transferring will cost more CPU resources of guest domain when HV²M is working, which are expensive when a small amount of data is being transmitted. What's more, compared with PV-on-HVM, HV²M grants/revokes page access more frequently, which also results in a worse performance during such tests. Besides, in two HVMs' test Domain-0 can still bear the switching burden in PV-on-HVM. But as more domains are involved and large messages are being transmitted, the overhead of switching and long path becomes PV-on-HVM's bottleneck.

When larger messages or more VMs are used, as shown in Figure 12 and 13, HV²M shows an advantage over PV-on-HVM. Although both of them transfer data directly in memory by CPU, HV²M distributes the CPU burden to every participant instead of making only Domain-0 responsible for doing the data transmission work. In those cases, every sender is granted to directly push data to its corresponding receiver, which releases Domain-0 from delivering data for each VM. In addition, inter-VM traffic won't be switched by the virtual switch in Domain-0, which shortens traffic path and is also helpful to boost the network performance.

Figure 14 gives a visual analysis on the CPU utilization of Domain-0. Compared with those using PV-on-HVM, using HV²M to communicate with other co-resident VMs hardly gets Domain-0 involved. That explains the low utilization of Domain-0's CPU. But using invoking page grant copy introduces additional cost. That leads to more CPU time has to be distributed to the network stack as each domain has to deal with data transferring in the low levels of the stack, which is handled by Domain-0 in those cases using PV-on-HVM.

During all tests, SR-IOV achieves a steady performance under both 100 Mbps and 1000 Mbps mode. Total throughput reaches almost the hardware's limit. As SR-IOV relies on DMA (Direct Memory Access) to copy data from the sender VM to NIC FIFO, then from NIC to the receiver VM's memory, both operations have to go through PCIe bus, which makes it slower than direct memory copy. What's more, the physical device's ability also has restricted the VFs'. As VFs share their host physical device's resources, their performance can never exceed the hardware's limit. Those are the main reasons that cause SR-IOV a lower throughput. That also means no matter how many VFs are used, the total throughput will never exceed the

Figure 15: Throughput comparison between HV²M and PV-on-HVM using Windows domains

PCIe device’s ability. What’s more, internal switching by VEB also has an influence on inter-HVM data traffic. From that aspect, data traffic’s path is no shorter than PV-on-HVM way. The only difference is who, hypervisor or device, shall do the switching. In addition, PCI-SIG has no standard now about how the internal switching shall be implemented, each vendor has its own solution which makes it implementation specific. That may also has an impact on achieving native device’s performance.

Due to the short path, HV²M achieves the smallest RTT results as shown in Figure 10. VMs can directly communicate with each other in HV²M which bypasses Domain-0 and virtual switch. However, the test also shows software approaches lacks adequate stability compared with SR-IOV. As both of HV²M and PV-on-HVM transmit data in memory by CPU, which will be affected by other issues like VCPU scheduling and interrupts. The VCPU’s execution has a direct impact on the data operation and in turn affects the RTT test. On the contrary, SR-IOV use DMA to handle data, which hardly involves CPU and leads to a more stable result. But compared with HV²M, the longer data path makes it 31.3% slower than HV²M on the average.

Although scheduling is not this paper’s major concern, but it does has an impact on the inter-VM data transmission. During the test which multiple

VMs are used, we notice that some VMs fall behind others a short time later after they simultaneously begin to run the same test scripts. A reasonable explanation is that those VMs' VCPUs fail in the competition for CPU resources and that in turn causes VMs' execution delayed, which leads to the bad performance of those domains.

Finally, Figure 15 gives a complementary test when Windows domains are used for testing. Each domain runs with Windows XP sp3 and the throughput is measured by using Nttcp. During those tests neither message size nor buffer size is specified, and only 2 posted send overlapped buffers are used (with -a option). Each domain is allocated to 1 VCPU and 1GB memory (so it can run smoothly). As SR-IOV's performance is too stable to be predicted, we give it up in the Windows tests. From the result we could figure out that HV²M is still better than PV-on-HVM but the lead is smaller, about 8% when using two pairs of Domains and 5% when using three pairs of domains. Except for the implementation differences between operating systems and their TCP/IP stacks, we think our lacking the experience in windows driver coding also affects the test result. Anyhow, HV²M is better and can be further optimized.

7. Conclusion

In this paper, we give a detailed analysis about inter-HVM network performance issues. In order to address those problems we propose a new approach to optimize inter-HVM network data delivery – HV²M, which is different from existing solutions in two main aspects. First it uses no shared memory as transmission media or buffers, so data copies are reduced. Second, it use a more gentle method to carefully transfer data which cuts down the security risks as it let the hypervisor to do the real data copy. HV²M is developed to boost the network I/O performance for HVMs in Xen. Test results show it can reduce the known overhead and achieve a better performance in network throughput. However, the idea and the prototype can still be further improved, for example, data descriptions could be attached to each frame's tail or head but that requires a clear boundary and we believe if that can be solved the whole mechanism can be further simplified. Also, to avoid repeatedly invoking hypercalls, data operations should be further aggregated than what we've done now, which perhaps require adding a new hypercall or somehow to complete the goal. What's more, our idea can also be easily transplanted to other platforms such as KVM.

As discussed in Section 2 and 3, VMX transition cost and scheduling also have an impact on the performance issue. When multiple VMs are handling network traffic, if a VM cannot be scheduled in time then it may not get adequate CPU time to deal with packets. VMX transition is caused by the nature of hardware-assisted virtualization, which currently relies more on hardware and specific instruction set, the overhead of such transition should be key to further optimization. However, neither is the emphasis of this work and we assume the existing frameworks Xen adopted can meet the requirements.

Acknowledgment

The authors would like to first express their sincere appreciation to Associate Professor Xuxian Jiang (chief scientist of Qihoo 360 Technology Co., Ltd, formally in North Carolina State University) as he offered lots of advice and suggestions to this work. The authors also would like to express their gratitude to anonymous reviewers for their professional opinions on this paper.

This work is supported by the National Science Foundation of China under Grant No.61572062, 61340031, and 61202425, Ph.D. Programs Foundation of Ministry of Education of China under Grant No.20121102110018, Science and Technology on Information Transmission and Dissemination in Communication Networks Laboratory Foundation and State Key Laboratory of Software Development Environment under Grant No. SKLSDE-2014ZX-18. The authors would like to appreciate their support.

Bai, Y., Ma, Y., Luo, C., Lv, D., Peng, Y., 2013. A high performance inter-domain communication approach for virtual machines. *Journal of Systems and Software* 86 (2), 367 – 376.

URL <http://www.sciencedirect.com/science/article/pii/S0164121212002476>

Burtsev, A., Srinivasan, K., Radhakrishnan, P., Bairavasundaram, L. N., Voruganti, K., Goodson, G. R., 2009. Fido: Fast inter-virtual-machine communication for enterprise appliances. In: *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*. USENIX'09. USENIX Association, Berkeley, CA, USA, pp. 25–25.

URL <http://dl.acm.org/citation.cfm?id=1855807.1855832>

- Chang, Z., Li, J., Ma, R., Huang, Z., Guan, H., Sept 2012. Adjustable credit scheduling for high performance network virtualization. In: Cluster Computing (CLUSTER), 2012 IEEE International Conference on. pp. 337–345.
- Cherkasova, L., Gardner, R., 2005. Measuring cpu overhead for i/o processing in the xen virtual machine monitor. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. ATEC '05. USENIX Association, Berkeley, CA, USA, pp. 24–24.
URL <http://dl.acm.org/citation.cfm?id=1247360.1247384>
- Dong, Y., Yang, X., Li, X., Li, J., Tian, K., Guan, H., Jan 2010. High performance network virtualization with sr-iov. In: High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on. pp. 1–10.
- intel.com, 2009. Intel 82576 sr-iov driver: Companion guide.
<http://www.intel.com/content/dam/doc/design-guide/82576-sr-iov-driver-companion-guide.pdf>.
- Kim, K., Kim, C., Jung, S.-I., Shin, H.-S., Kim, J.-S., 2008. Inter-domain socket communications supporting high performance and full binary compatibility on xen. In: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. VEE '08. ACM, New York, NY, USA, pp. 11–20.
URL <http://doi.acm.org/10.1145/1346256.1346259>
- Liao, G., Guo, D., Bhuyan, L., King, S. R., 2008. Software techniques to improve virtualized i/o performance on multi-core systems. In: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems. ANCS '08. ACM, New York, NY, USA, pp. 161–170.
URL <http://doi.acm.org/10.1145/1477942.1477971>
- Menon, A., Cox, A. L., Zwaenepoel, W., 2006. Optimizing network virtualization in xen. In: Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference. ATEC '06. USENIX Association, Berkeley, CA, USA, pp. 2–2.
URL <http://dl.acm.org/citation.cfm?id=1267359.1267361>

- Menon, A., Santos, J. R., Turner, Y., Janakiraman, G. J., Zwaenepoel, W., 2005. Diagnosing performance overheads in the xen virtual machine environment. In: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments. VEE '05. ACM, New York, NY, USA, pp. 13–23.
URL <http://doi.acm.org/10.1145/1064979.1064984>
- openbenchmarking.org, 2015. Ubuntu 14.04 lts amazon ec2 pv vs. hvm. <http://openbenchmarking.org/prospect/1404275-KH-UBUNTUXEN97/928ab7a268452c2b0d1247657f7e8f53ef633c5c>.
- pcisig.com, 2010. Single root i/o virtualization and sharing specification. <https://members.pcisig.com/wg/PCI-SIG/document/download/8238>.
- phoronix-test suite.com, 2015. Phoronix test suite. <http://www.phoronix-test-suite.com>.
- Santos, J. R., Turner, Y., Janakiraman, G., Pratt, I., 2008. Bridging the gap between software and hardware techniques for i/o virtualization. In: USENIX 2008 Annual Technical Conference on Annual Technical Conference. ATC'08. USENIX Association, Berkeley, CA, USA, pp. 29–42.
URL <http://dl.acm.org/citation.cfm?id=1404014.1404017>
- Sugerman, J., Venkitachalam, G., Lim, B.-H., 2001. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In: Proceedings of the General Track: 2001 USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, USA, pp. 1–14.
URL <http://dl.acm.org/citation.cfm?id=647055.715774>
- Wang, J., Wright, K.-L., Gopalan, K., 2008. Xenloop: A transparent high performance inter-vm network loopback. In: Proceedings of the 17th International Symposium on High Performance Distributed Computing. HPDC '08. ACM, New York, NY, USA, pp. 109–118.
URL <http://doi.acm.org/10.1145/1383422.1383437>
- wiki.xen.org, 2014. Xen linux pv on hvm drivers. http://wiki.xen.org/wiki/Xen_Linux_PV_on_HVM_drivers.
- wiki.xen.org, 2015. Xen windows gplpv. http://wiki.xen.org/wiki/Xen_Windows_GplPv.

Zhang, X., McIntosh, S., Rohatgi, P., Griffin, J. L., 2007. Xensocket: A high-throughput interdomain transport for virtual machines. In: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware. Middleware '07. Springer-Verlag New York, Inc., New York, NY, USA, pp. 184–203.
URL <http://dl.acm.org/citation.cfm?id=1516124.1516138>