Virtualization I/O Optimization Based on Shared Memory

Fengfeng Ning
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Email: wikke@sjtu.edu.cn

Chuliang Weng
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Email: weng-cl@cs.sjtu.edu.cn

Yuan Luo
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Email: luoyuan@cs.sjtu.edu.cn

***Abstract*—With the development and popularization of cloud computing, more and more services and applications are migrated to cloud for the sake of low cost, high availability and excellent performance. As the foundation of cloud computing, virtualization technology integrates and reallocates the computing capability, storage and network resource fairly among virtual machines and provides a full-featured, isolated and reliable hardware environment for various operating systems. Owe to the virtualization technology, computing capability of virtual machines has achieved fantastic performance, some even achieve near native speed. However, low I/O performance is still a bottleneck, especially in I/O intensive applications. The leading causes include redundant data copy and frequent VM exits. Focusing on network I/O optimization, we design and implement virtsocket, a new network socket library in virtualization scenario which utilizes shared memory for data transmission. A ring buffer data structure stores I/O requests of virtual machine which is triggered to issue all requests with only one hypercall according to scheduler. Data referred in the I/O requests is read directly from virtual machine memory by host machine kernel module with interfaces provided by modified hypervisor. Experimental results show that throughput is improved by hundreds of times when compared with original virtualization scenario, and the latency also achieves a remarkable reduction. Both throughput and latency performance exceed existing para-virtualization solutions.**

***Keywords*-architecture virtualization, shared memory, I/O optimization**

I. Introduction

Architecture virtualization has already been widely deployed in data centres and enterprises, which efficiently reduces cost and improves productivity. With virtualization technology, different virtual machines with various operating systems can run on the same host machine simultaneously, sharing the computing capability, storage and network resource, etc. As a result, resource utilization ratio raises, hardware purchase cost reduces and system maintenance becomes flexible. Resource owned by host machine is multiplexed by privileged software referred to as hypervisor or virtual machine monitor(VMM), which provides an integrated hardware environment for virtual machines. Hypervisor guarantees the isolation property between virtual machines so that each virtual machine is unaware of others and cannot intervene them. Resource is distributed fairly according to scheduling strategy, like credit scheduler in Xen[1]. Besides, live migration promises that a virtual machine can be migrated to another host machine while keeping the services running continuously. All these features of virtualization technology ensure a secure, robust, reliable and fair environment for cloud applications.

In a technical perspective, architecture virtualization is categorized as full virtualization and para-virtualization. Full virtualization offers all the hardware requirements, including processors, memory and devices, for guest machine to run without any modifications. However, some privileged instructions, like page table update, are completed with assistants of the hypervisor, which is inefficient and costly. In para-virtualization, guest virtual machine is modified for a more efficient execution of privileged instructions. A specific module is installed in virtual machine to cooperate with hypervisor to avoid the expensive simulation cost.

KVM[2] is a full virtualization hypervisor product which needs hardware support, often referred to as hardware-assisted virtualization. By now, Intel and AMD both support for hardware-assisted virtualization with Intel-VT[3] and AMD-V[4] technology, which add two operation modes for x86 architecture, root operation and non-root operation. Virtual machine runs in non-root operation and hypervisor runs in root operation. 4 privilege rings are supported in each operation so virtual machine can run natively by switching. VM exit, usually caused by exceptions, will trigger the switch from non-root operation to root operation. Hardware loads the context of the hypervisor process automatically. Virtual machine is resumed with a VM entry initiated by hypervisor at a proper time. Each virtual machine is actually a normal process in host machine operating system and scheduled by the default Linux scheduler, which brings great convenience and flexibility for maintenance. All the management tools, scheduling algorithm and optimization strategy for Linux processes can be applied to virtual machine processes seamlessly. By the way, para-virtualization acceleration can be applied to KVM via VirtIO, which needs qemu-kvm[5] support and VirtIO drivers installed in guest operating system.

Poor I/O performance of virtualization results from many aspects. Take network I/O in virtualization scenario for example, each virtual machine is assigned with a virtual

978-1-4799-1293-3/13/$31.00 ©2013 IEEE

network device, which is apparent to virtual machine operating system and applications. The virtual network device connects with physical network device in host machine via a device file so that network packets can be multiplexed correctly in and out of virtual machines. However, in such implementation, packet data is copied redundantly during the transmission and VM exits occur frequently, which result in low throughput and long latency. We try to optimize I/O performance in virtualization scenario via shared memory strategy. Redundant data copy is avoided by shared memory transmission and frequent VM exits are reduced with a ring buffer data structure and effective scheduler. Experimental results show that throughput is improved significantly and latency is reduced. Through our efforts focus on network optimization, principles of our design and implementation can be applied broadly to other I/O scenarios in all virtualization fields, which is discussed in section 3.G.

The rest of this paper is organized as follows. Section 2 elaborates the network I/O architecture and other related I/O optimization researches. Section 3 presents the design and implementation of virtsocket, a new network socket library in virtualization scenario which utilizes shared memory to transmit data. Section 4 lists the experiments and evaluation results. Section 5 concludes this paper and summaries our contributions and future work.

II. Background

With more and more I/O intensive applications deployed in cloud environment, low I/O performance has been the bottleneck in most cases. Many researches have been conducted to optimize in this situation, including buffer strategy and instructions optimization by software techniques. A classical instance goes that two co-resident virtual machines communicate with each other via a TCP connection, the cost is expensive, which will be elaborated in section 2.C. Another real industrial example goes that in a map-reduce application system, several co-resident virtual machines cooperate with each other to solve a complicated problem. The input file are located at host machine and accessed by virtual machine applications via network file system. In such scenario with great I/O workloads, I/O performance is playing a vital role. Different from such specific scenarios, we are aiming at a more common and wider adopted optimization on virtualization I/O. And we would like to start with the network I/O architecture in KVM environment.

A. KVM Network I/O

In KVM virtualization, a slight modified QEMU process in the host machine is responsible for the simulation of I/O requests issued by virtual machines. Various devices simulation is provides by QEMU including a virtual network interface controller(VNIC), which is bridge connected with the physical network device through a tap device file. Network packets in and out of virtual machine are transmitted via this path as depicted in figure 1.

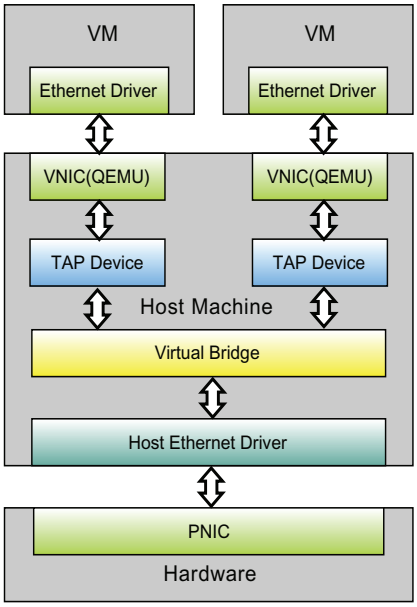

Figure 1. KVM Network I/O architecture

In the original virtualization scenario, network packets received by applications in virtual machines will arrive at PNIC firstly, then distributed to correct tap device via a virtual bridge. QEMU process sends the packets into right virtual machine and notifies this event by injecting an interruption. Virtual machine detects the interruption and processes with the packets. Sending data is in a similar but reverse way. Network I/O requests from virtual machines are transmitted to host machine kernel via the virtual network control controller(VNIC), which induces non-root operation and root operation switching. Since I/O operations are simulated in user space QEMU process, transmitted data need to be copied to host machine user space, which brings redundant data copy[6].

B. Existing Network I/O Optimization Solutions

1) VirtIO: VirtIO is designed to be a de-facto standard for virtual I/O devices, proposed by Rusty Tussell[7]. It's currently implemented in KVM and lguest and achieves excellent I/O performance. Like Xen, VirtIO is a para-virtualization solution that requires drivers to be installed in guest machines. However, VirtIO is aiming at a unified framework and interface for device drivers in different virtualization environment since the differences among hypervisors bring great difficulties when writing and maintaining device drivers. By now, VirtIO drivers for disk, network and PCI devices have been implemented for various operating systems. Performance is improved significantly when compared with original virtualization scenario. However, redundant data copy still exists.

The key data structure in VirtIO is virtqueue, which is a buffered ring structure that stores I/O requests and issue all at once. Thus, multiple VM exits are merged and submitted by a kick function, which reduces VM exits apparently and improves the performance. Great flexibly is achieved when multiple virtqueues are used for data transmission and the kick function is served as interface for user defined scheduler.

2) Vhost: Vhost[8] is a further improvement based on VirtIO which optimizes I/O performance by avoiding re-trap in kernel operation for host machine. Assuming that a kick function is invoked and a virtqueue of I/O requests are sent to host. Since KVM kernel module does not process I/O requests directly, instead it transfers them to QEMU process in user space. I/O requests are simulated by QEMU process with system calls invoked inevitably, which brings a kernel trap. Obviously, it is unwise to trap into kernel space again since it just comes from there. Vhost solves this problem by simulating the I/O requests in kernel space directly instead of transferring to QEMU in user space. At the time this paper is being written, community has implemented the Vhost-net for network optimization, disk and PCI device optimization are still in progress.

3) Software Techniques: Ole Agesen proposed a solution that utilizes software techniques to avoid hardware virtualization exits in VMware products[9]. Hypervisor inspects guest code dynamically to detect back-to-back pairs of instructions that both exit, which will save 50% of the cost. Furthermore, they generalize from pairs to clusters of instructions that include loops and other control flows. A binary translator is used to generate customized translations when handling exits instructions[9].

C. Inter-VM Network Optimization

Virtualization techniques ensure the isolation, security and fair resource sharing among virtual machines. However, great cost is paid in this specific scenario. For example, two co-resident virtual machines communicate with each other inefficiently, since they are unaware of the virtualized environment. Data is copied redundantly and vm exits occur frequently during communication. Many researches have been conducted to optimize the network performance in such scenario.

1) XenLoop: XenLoop[10] is a full transparent and high performance inter-VM network loopback channel implemented in Xen. Guest virtual machine can switch between the standard network path and XenLoop channel seamlessly. Xenloop intercepts network packets under the network layer. If co-resident communication is detected, packets would be sent to target VM through shared memory channel that bypasses the virtual network interface controller.

2) XenSocket: XenSocket[11] is high performance network channel designed for co-resident inter-VM communication with a static circular memory buffer shared between two virtual machines. It's implemented in Xen and only supports one-way connection. Transmitted data is written in shared memory firstly by the sender VM and read asynchronously by the reader VM. Notifications are transmitted through the event channel provided by Xen. A new socket family is created, which provides a standard socket API for applications.

3) XWay: XWay[12] provides optimizations for inter-VM TCP communications while keeps the user-level transparency. Network protocol stack is modified so that shared memory can be used to transmit data for TCP connections between co-resident virtual machines. Besides, XWay supports live migration. The implementation of XWay relies on grant tables and event channel provided by Xen.

III. Design and Implementation

We try to improve I/O performance in virtualization with shared memory. Our research mainly focuses on network I/O in KVM virtualization. The feasibility of applying the optimization principles to other I/O is demonstrated in 3.G. After trials and efforts we would like to introduce virtsocket: a new network socket library implemented in KVM virtualization environment which utilizes shared memory for data transmission. The shared memory optimization brings higher throughput and less latency when compared with original virtualization scenario. Take original network virtualization scenario for instance, redundant copy and frequent non-root operation and root operation switching brings too much overhead when data is transmitted via TCP or UDP channel, results in poor performance finally. With virtsocket, I/O requests from virtual machine are buffered in a ring list and issued with one hypercall according to scheduler. We would like to make virtsocket a general socket library in different architecture virtualization environment.

A. Data structure

In virtsocket, data to be transmitted is represented by a descriptor data structure, which keeps all the properties of a data chunk including the memory address, data length, offset and so on. Descriptors are organized in a ring buffer data structure, named descriptor list. A descriptor list of I/O requests are issued with one hypercall instead of one by one. Host machine maintains a descriptor list for each virtual machine and synchronizes with the descriptor list in corresponding virtual machine. When the virtual machine process switches from non-root operation to root operation, host KVM virtsocket module updates local descriptor list by reading the memory of the descriptor list in guest machine. After I/O requests are completed, latest descriptor list is written back to guest memory to keep update.

B. Socket API

Aiming at flexible and convenient I/O acceleration library for virtualization, a standard socket interface is provided

for applications in virtual machines. Little modifications are paid to apply applications to virtsocket. New socket address family(AF_VIRT) and protocol family(PF_VIRT) are registered in Linux kernel by virtsocket modules, which handle virtsocket operations in kernel.

C. Hypercall

Like system call in Linux kernel, hypercall acts similarly but invoked from guest virtual machine user space and responded from host machine kernel space. Hypercall is expensive because VM exit will be triggered. However, the quick response and high priority still make hypercall a good choice for message passing in specific scenarios.

Considering the timeliness, hypercall is used to send requests in virtsocket via standard socket interface like *connect*, *accept*, *bind*, *send* and *recv*. Though VM exit caused by hypercall is expensive, methods like *connect*, *accept* and *bind* are invoked with finite times in real applications, which causes negligible cost in benchmarks and real productions. However, methods like *send* and *recv* need be to optimized since they are invoked frequently. A ring buffer descriptor list is implemented as a buffer for *send* and *recv* requests in virtsocket. A simple but practical scheduling strategy is designed to trigger the hypercall to flush all the I/O requests in descriptor list.

Adding a hypercall to a virtual machine consists of 2 steps. First, a system call sys_virtsocket is implemented in virtual machine Linux operating system. In the implementation function of the system call, running environment is detected whether current operating system is a virtual machine via KVM APIs, since it makes no sense to add a hypercall in a non-virtual machine. Secondly, hypercall is performed by invoking the interfaces of KVM library, which changes the virtual machine process from non-root operation to root operation. We slightly modified the KVM module in host machine to add support for capturing virtsocket hypercalls. After hypercall is captured, I/O requests are transferred to a server virtsocket, which reads the parameters stored in registers, performs I/O operations and responses the results back.

D. Shared memory

In a global perspective, all co-resident virtual machines share the computing capability and storage resource of the host machine. Theoretically, communication between co-resident virtual machines can achieve similar performance with inter-process communication. I/O of the virtual machine can achieve similar performance with that in host machine. But the truth is that it's impractical and unrealistic, since the isolation, security and privacy need to be ensured for a stable and independent environment. Even though, hypervisor still has chances to get access to virtual machines, which provides possibilities for optimizations in specific scenarios, such as shared memory implementation between virtual machine and host machine in our research.

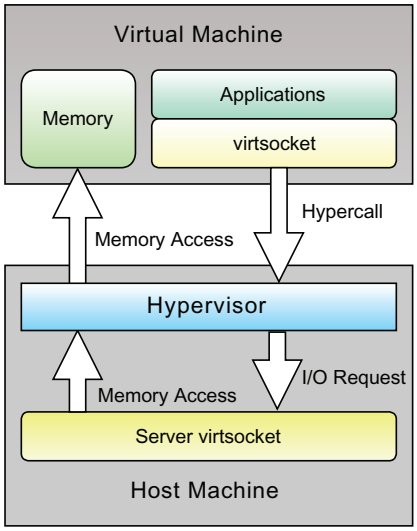

Figure 2. global diagram of virtsocket

Logically, hypervisor works between host machine and virtual machine and handles the management, scheduling and resource distribution for virtual machines. In KVM, a virtual machine instance is actually a process running in host machine, whose logical memory corresponds to the physical memory of the virtual machine. KVM module is responsible for the memory mapping between the virtual machine physical address and the logical address in the host machine, which is implemented by shadow page or extended page table(EPT)[15]. Guest virtual machine memory can be accessed by host kernel via interfaces of KVM modules, which is modified and used in our research. When a virtsocket hypercall is captured by KVM, the descriptor list and I/O data of virtual machine are transmitted to host machine via a hypercall. By the way, unpublished interface also support for VM logical and physical address translation.

E. Global workflow

Sections above describe some important points in the design and implementation of virtsocket, covering the application interface, hypercall implementation and shared memory strategy design. In this section, a global workflow of virtsocket in virtualization environment will be presented, which is depicted by a diagram in figure 2.

Guest virtual machine and host machine both need virtsocket module loaded in kernel which registers AF_VIRT address family and PF_VIRT protocol in Linux operating system. KVM kernel module in host machine are slightly modified to add support to recognize the virtsocket hypercall. Applications running in guest machine can enjoy the virtsocket optimization by including related header files and creating socket instance with virtsocket family address

AF_VIRT. Virtsocket kernel module in the guest virtual machine allocates memory for the descriptor list with static size. The reason why we used a static size descriptor list is that the dynamic allocation cost can be avoided since it brings great workload to allocate and free new descriptor memory which makes the system unstable. Then the virtsocket instance invokes *connect* method, a hypercall is issued to host machine with the physical memory address of the descriptor list as a parameter. Server virtsocket stores the address for synchronization later.

After virtsocket connection is established, everything is ready for data transmission. Applications in virtual machine send data via the socket API *send*. Data is copied to kernel space firstly and then an empty descriptor is found and filled with information about this data chunk, such as the memory address, length and status, etc. The descriptor list is a ring buffer data structure, which means I/O requests are buffered and issued according to scheduler. A fair and efficient scheduling algorithm is important and should be designed carefully because an intensive issue ratio will cause frequent VM exits. On the contrary, loose issue ratio will result in long latency. In virtsocket, we implemented the scheduler simply: when the ring buffer descriptor list is full, all requests are issued at once via a hypercall. It's apparent that VM exits will decrease since the hypercall will result in only one VM exit instead of multiple VM exits for every I/O request. We are aiming at optimizing the I/O performance via shared memory and exploring the maximum optimization. So a simple and efficient scheduling algorithm would make things clean and direct.

When VM exit occurs, several parameters are stored in specified registers and captured by KVM module in host machine. Besides, KVM module in host machine provides interfaces to read and write memory of virtual machines, which facilitates the shared memory implementation in virtsocket. When the descriptor list satisfies the scheduling prerequisites, a hypercall is triggered, which passes parameters to the host machine, including the hypercall number and description list memory address. Server virtsocket receives these parameters and updates local descriptor list by reading the latest descriptor list content via access to virtual machine memory directly. After all I/O requests are completed, latest descriptor list in host machine is written to specific guest virtual machine memory for synchronization.

F. Compare with previous work

XenLoop, XenSocket and XWay optimize network I/O performance for co-resident inter-VM communication via different implementations. However, the key principle behind is the utilization of shared memory, which alters the data transmission path to bypass the virtual network interface. Other optimizations solutions utilize the shared memory strategy to achieve significant improvement with different emphasises. Fido[13] focuses on enterprise appliances and ZIVM[14] implements a delicate architecture to achieve zero copy for co-resident inter-VM communication.

Different from previous work, we are trying to improve the I/O performance in virtualization environment, especially for the network communication between virtual machine and host machine. The original intention of this paper is to solve the poor performance when virtual machine applications read or write files in host machine via network file system. Let's assume that several virtual machines are running on the same host machine, which cooperate with each other in a map-reduce application. It's a wise decision to put the input file at host machine, which can be accessed by virtual machine via network file system. Storage resource can be significantly saved since the input file is located at host machine instead of multiple duplications in every virtual machine, especially when the input file are huge in most cases. Thus, a high performance I/O channel between virtual machine and host machine is needed. Shared memory is demonstrated to be a good idea, as many previous work has demonstrated, and is rarely used for virtual machine I/O optimization in KVM environment. So we decide to do something to optimize I/O performance in such scenario. To summarize, our work differs from previous work mainly on two aspects:

1) We are aiming at improving virtualization I/O performance and our efforts mainly focus on network I/O. Feasibility will be demonstrated in next section that similar idea can be applied to other virtualization I/O with slightly modification of the interface and drivers.
2) Different from the specific co-resident inter-VM network communication scenario in XenLoop, Xensocket and so on, virtsocket is designed for a more general and common scenario in which the other endpoint of virtsocket connection can be located anywhere, not limited in co-resident VMs. According to current implementation, virtsocket connection established between virtual machine and host machine achieves significant performance improvement.
3) The shared memory implementations are different. In previous work, shared memory is a public memory zone used for data transmission between different VMs, and they can get access equally. However, in our work, shared memory is not what it indicates literally strictly. With the assistance of hypervisor, host machine can get access to virtual machine memory directly, in which way the data is transmitted and descriptor list is synchronized.

G. Feasibility in other I/O systems

We apply shared memory optimization to virtualization network I/O scenario, which can also be applied to other virtualization I/O systems easily. The key principle behind virtsocket is the utilization of shared memory for data transmission. So when it comes to other I/O scenarios, like

disk or peripheral devices, we need to alter the interfaces of drivers to support for applications. I/O path from virtual machine to host machine is similar to network I/O, which can be optimized with same way: requests are buffered in a ring data structure and issued according to scheduler. Most importantly, data is transmitted via shared memory that the corresponding kernel module in host machine can get access to virtual machine memory directly for right information in need, which is supported by current popular hypervisors. Here we just put forward a shared memory solution for I/O optimization and implement virtsocket for network I/O scenario in KVM. The feasibility for general I/O systems optimization in virtualization environment can be achieved with similar techniques.

IV. Performance evaluation

Virtsocket transmits data via shared memory and deduces VM exits by a ring buffer data descriptor and scheduling strategy, which is expected to achieve big improvements in performance. Like any other network evaluations, we choose throughput and latency as the criteria to judge the network performance. We conducted the experiments with the latest qemu-kvm 1.2.0 version in Linux operating system with 3.2.0 kernel version for both host machine and guest virtual machine. Hardware environments includes Intel core i3 2.1.3GHz CPU, supporting Intel-VT technology, 3GB memory for host machine and 718M memory allocated for virtual machines. We collected the experimental results carefully, each unit test is performed at least 3 times and the average value is token as the result. Besides, we made a performance comparison between different network scenarios listed as below.

1) original virtualization environment: virtual machine is running without any modifications, external drivers or any accelerations.
2) virtualization with VirtIO: VirtIO network drivers are installed in virtual machines.
3) Unix domain socket: a Unix domain socket connection is established with both endpoints in the same host machine.
4) virtsocket: virtsocket is loaded in virtual machine and used for network communication.

A. Throughput

We evaluated the throughput performance of all network scenarios in the same environment that a connection between the virtual machine and host machine is established to transmitting large volumes of data. To ensure the reliability and objectivity, BigdataBench[16] is brought in our research, which is a big data benchmark suite, providing tools to generate big data from seed files by analysis and expansion, while keeping the semantic close to real big data. Our test program reads the data generated by BigdataBench tools and sends to another end of the connection, which simulates

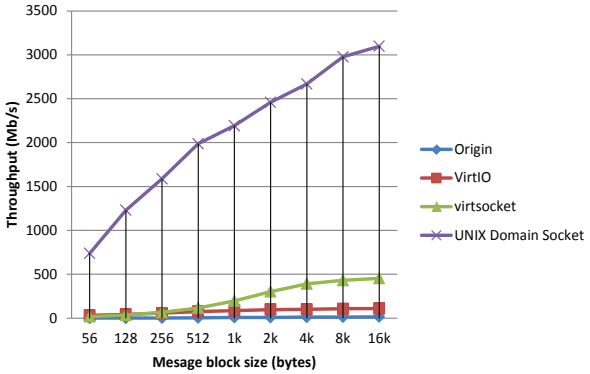

Figure 3. throughput in different network scenarios

a prototype of real applications, transmitting semantic and meaningful data, instead of random, regular even the same data every time.

Besides, we care about how the performance goes with different socket message size. For example, 1M data are to sent with each message 1k bytes. 1024 requests are needed to finish the task. However, if each message contains 4k bytes, only 256 requests are sufficient. In a word, bigger message size increases the duration of request but decreases the request quantities. We tested the throughput performance of each network scenario in different message size ranging from 56 bytes to 16k bytes. The results are listed in table 1 and depicted in figure 3.

According to experimental results, Unix domain socket always exceeds others in every message block size, since data packets are transmitted through a high throughput channel between endpoints, bypassing the protocol stack. Original virtualization environment preforms worst always because the network packets are transmitted with heavy cost and redundant copy. Virtualization with VirtIO drivers improves when compared with original virtualization, owing to a virtqueue data structure, which stores I/O requests and is triggered by a kick function interface. However, redundant data copy still remains. Virtsocket improves I/O throughput even more when compared with VirtIO scenario. As we can see, throughput of virtsocket performs better than VirtIO when message block size is bigger than a certain size between 128 bytes and 256 bytes, and even better when message block size increases. However, the increasing ratio decreases when message block size comes to 4k bytes, because the descriptor list is an array with fixed length and each descriptor can only describe a finite size of data. So when message block size exceeds 4k bytes, data will be divided into chunks to fill other descriptors, resulting in slower increasing ratio.

B. Latency

Latency is an important factor to reflect system performance, especially in I/O sensitive systems. We evaluated

Table I
THROUGHPUT IN DIFFERENT NETWORK SCENARIOS

	56	128	256	512	1k	2k	4k	8k	16k
origin	0.41	0.94	1.91	3.87	8.68	7.66	11.81	12.11	14.70
VirtIO	31.20	41.28	57.08	74.74	86.24	97.09	100.38	106.64	108.67
virtsocket	16.36	35.14	65.73	116.2	196.58	301.81	390.42	430.42	453.40
Unix Domain socket	737.02	1229.13	1587.10	1987.32	2192.88	2458.15	2668.43	2977.87	3097.89

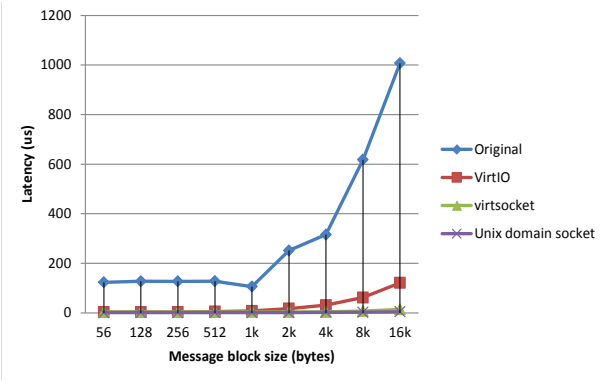

Figure 4. latency in different network scenarios

the latency in different network scenarios by carefully designed microbenchmarks. One byte data are sent via network connection with tremendous times. Cost spent on sending one byte of data is negligible, however, other costs, like the system switching, kernel trap and so on, impact the latency directly. The average period of a request can be calculated as the criterion of latency. We compared the latency in different network scenarios with message block size ranging from 56 bytes to 16k bytes. Results are listed in table 2 and depicted in figure 4.

As experiments show, latency of the original virtualization increases dramatically along with the message block size, surpassing all other scenarios. Virtualization with VirtIO drivers in guest virtual machine improves dozens of times when compared with original scenario, and appears to be linear relation with message block size. Virtsocket and Unix domain socket both have excellent performance with short latency. For Unix domain socket, kernel network stack is bypassed resulting in a short latency. I/O requests sent via virtsocket connection is triggered by a hypercall, which responses instantly when the request is completed because of the high priority of hypercall.

C. Evaluation summary

As evaluation results show, virtsocket achieves significant improvements in both throughput and latency, exceeds the original virtualization scenario for far. When compared with VirtIO, throughput of virtsocket performs better when message block size is bigger than certain size between 128 bytes and 256 bytes, and even better when message block size increases. Latency is also optimized with help of hypercall and a simple scheduling algorithm. Different from other 3 network scenarios, Unix domain socket bypasses the kernel network stack which reduces the cost and achieve highest performance among all.

The performance of original virtualization scenario appears to be extremely poor. The most critical reasons include redundant data copy and frequent VM exits cost. Virtsocket solves the redundant copy problem by shared memory data transmission path. Host server virtsocket can get access to virtual machine memory directly via interfaces of KVM module, which turns out to be a significant improvement. Besides, frequent VM exits problem is alleviated by aggregation of VM exits in a ring buffer descriptor list, and all I/O requests are issued with one hypercall according to scheduler.

Shared memory strategy is an efficient way to improve I/O performance in virtualization. However, some underlying risks also exist. Since server virtsocket can get access to all descriptors in a descriptor list, vicious applications might get access to unauthorised memory address, which brings potential risks including intentional deceive, vicious attacks and so on. Assumption can be made that virtsocket is a part of trusted computing base, but it solves nothing in real applications. A more elaborate grant strategy is expected which ensures server virtsocket can only get access to authorized memory address, which is what we are working on.

V. Conclusion

With the popularization of the cloud computing, architecture virtualization has been a key factor to ensure security, efficiency and stability. Though the computing capability of virtual machine running on the latest virtualization product is excellent, some even achieve near native speed, I/O performance is always the bottleneck, especially in I/O intensive scenarios. We explored the possibility to improve I/O performance in virtualization via shared memory strategy and introduced virtsocket, a new network socket library in virtualization scenario which utilizes shared memory for data transmission. Experiments were conducted to demonstrate the significant improvement in both throughput and latency compared with original virtualization scenario and existing para-virtualization acceleration such as VirtIO.

To sum up, our contributions include:

1) Improve I/O performance with shared memory strategy. Aiming at network I/O optimization in virtual-

Table II
LATENCY IN DIFFERENT NETWORK SCENARIOS

	56	128	256	512	1k	2k	4k	8k	16k
origin	123.44	127.39	127.01	127.75	105.97	251.62	315.75	617.72	1007.27
VirtIO	2.74	3.39	3.40	4.98	7.90	17.00	31.22	62.07	121.09
virtsocket	3.12	3.24	3.27	3.25	3.62	3.69	3.79	7.36	14.01
Unix Domain socket	0.79	0.79	0.82	0.87	0.98	1.14	2.13	2.37	4.26

ization, we implemented virtsocket, which achieves an significant improvement in both throughput and latency as experimental results show.

2) Reduce the VM exits by a ring buffer data structure and scheduler. The ring buffer data structure descriptor list stores I/O requests and issues according to scheduler. multiple I/O requests are merged and issued by only one hypercall, which reduces the cost brought by frequent VM exits.

Virtsocket is aiming at a general socket library for applications in different architecture virtualization environments. By now, the research is still in progress and the ongoing research points include vicious memory access, asynchronous I/O and better hypercall scheduling algorithm. We believe that poor I/O performance in industrial virtualization fields needs to be improved urgently in innovative ways. Shared memory I/O optimization is verified as a possible solution which is expected to play a big role in the future.

ACKNOWLEDGEMENT

We would like to thank everybody involved in researches and experiments, with whom new ideas are sparking and experiments are conducted successfully. Especially, we would like thank Chuliang Weng and Yuan Luo, who offer us wise guidance and valuable suggestions. This work was supported in part by the National Basic Research Program of China under Grant 2013CB338004, and the Research Fund for the Doctoral Program of Higher Education of China under Grant 20100073110016. We would also like to acknowledge the foundation TS0520103001 of Shanghai Jiao Tong University and University of Leuven.

REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield. *Xen and art of Virtualization*. In Proceedings of the 2nd conference on Real, Large Distributed Systems, Vol. 37 (October 2003), pp. 164-177.

[2] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. *kvm: the Linux virtual machine monitor*. In OLS 2007: Proceedings of the 2007 Ottawa Linux Symposium.

[3] Intel. Hardware-Assisted Virtualization Technology - *Improving the fundamental flexibility and robustness of traditional software-based virtualization solutions*. http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/hardware-assist-virtualization-technology.html

[4] AMD. *AMD Virtualization*. http://sites.amd.com/us/business/it-solutions/virtualization/Pages/virtualization.aspx

[5] QEMU. *QEMU: Open Source Processor Emulator*. http://wiki.qemu.org/Main_Page

[6] Ding S, Ma R, Liang A, et al. *Optimization for Inter-VMs network performance on KVM*.

[7] R. Russell, R. *virtio: towards a de-facto standard for virtual I/O devices*. SIGOPS Oper. Syst. Rev. 42, 5 (2008), 95103.

[8] VHost. http://www.linux-kvm.org/page/UsingVhost

[9] Agesen O, Mattson J, Rugina R, et al. *Software techniques for avoiding hardware virtualization exits*. Tech. rep., VMware, 2011.

[10] J. Wang, K.-L. Wright, and K. Gopalan. *XenLoop: a transparent high performance inter-vm network loopback*. in HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing. New York, NY, USA: ACM, June 2008, pp. 109-118.

[11] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. *XenSocket: A High-Throughput Interdomain Transport for Virtual Machines*. in Mid-dleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware. New York, NY, USA: Springer, November 2007, pp. 184-203.

[12] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim, *Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen*, inVEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. New York, NY, USA: ACM, March 2008, pp. 11-20.

[13] Burtsev A, Srinivasan K, Radhakrishnan P, et al. *Fido: Fast inter-virtual-machine communication for enterprise appliances*, in Proceedings of the 2009 conference on USENIX Annual technical conference. USENIX Association, 2009: 25-25.

[14] Mohebbi H R, Kashefi O, Sharifi M. *Zivm: A zero-copy inter-vm communication mechanism for cloud computing*. Computer and Information Science, 2011, 4(6): p18.

[15] Sheng Yang *Extending KVM with new Intel Virtualization technology*. http://www.linux-kvm.org/wiki/images/c/c7/KvmForum2008\$kdf2008_11.pdf

[16] Gao W, Zhu Y, Jia Z, et al. *BigDataBench: a Big Data Benchmark Suite from Web Search Engines*. arXiv preprint arXiv:1307.0320, 2013.