

ZIVM: A Zero-Copy Inter-VM Communication Mechanism for Cloud Computing

Hamid Reza Mohebbi (Corresponding author)

School of Computer Engineering, Iran University of Science and Technology

P.O. Box 16846-13114, University Road, Hengam Street

Resalat Square, Narmak, Tehran, Iran

Tel: 98-21-7724-0540 E-mail: mohebbi_hr@comp.iust.ac.ir

Omid Kashefi

School of Computer Engineering, Iran University of Science and Technology

E-mail: kashefi@comp.iust.ac.ir; kashefi@ieee.org

Mohsen Sharifi

School of Computer Engineering, Iran University of Science and Technology

E-mail: msharifi@iust.ac.ir

Received: January 17, 2011

Accepted: September 13, 2011

Published: November 1, 2011

doi:10.5539/cis.v4n6p18

URL: <http://dx.doi.org/10.5539/cis.v4n6p18>

Abstract

With the advent of virtualization technology and its propagation to the infrastructure of Cloud distributed systems, there is an emergent request for more effective means of communication between virtual machines (VMs) lying on distributed memory than traditional message based communication means. This paper presents a distributed virtual shared memory mechanism called ZIVM (Zero-copy Inter-VM) to ease the programming of inter-VM communications in a transparent way and at the same time to achieve comparable and bearable execution performance. ZIVM has been implemented as a virtual cluster on a hosted virtual machine using the KVM hypervisor. Experimental results have shown near native performance in terms of latency and bandwidth.

Keywords: Cloud Computing, Inter-VM Communication, Virtualization, Shared Memory

1. Introduction

Virtualization technologies (VTs) have been employed in various contexts computation world from operating systems, programming languages and compilers to servers and networks (Smith, 2005). VT can bring availability, scalability, isolation, portability, manageability. System virtual machines allow the sharing of the underlying physical machine resources between different virtual machines, each running its own operating system. The software layer providing the virtualization is called a virtual machine monitor (VMM) or hypervisor. A hypervisor can run on bare hardware or on top of an operating system. There are a number of virtualization techniques such as: Hardware-assisted virtualization, where some support for virtualization is built into the underlying hardware. Full virtualization, sufficiently simulates the underlying hardware to allow software, typically a guest operating system, to run unmodified. Paravirtualization is a virtualization technique that presents a software interface to virtual machines that is similar but not identical to that of the underlying hardware. The VM control program includes a hypervisor-call handler that intercepts DIAG (Diagnose) instructions used within a virtual machine. This provides fast-path non-virtualized execution of file-system access and other operations (Smith, 2005).

Using virtual execution environment can provides support of legacy applications (Marshall, 2006). Nevertheless, VT have not much been attended in the context of high performance computing (HPC), mostly because it usually incurs substantial performance degradation. Recently, advances in hardware and software virtualization have addressed some of these performance issues (Mergen, 2006). Virtual clusters try to virtualize a single physical cluster into multiple independent virtual clusters to provide a virtual server. Virtual clusters greatly benefit from

the ease of management brought by virtual machines. Live migration capabilities could provide fault tolerance and load balancing transparently to applications. Using virtualization for improving the performance of HPC applications has been studied. Achieving the best possible performance in a cluster requires the system can dynamically reorganized the cluster according to the needs and requirements of applications and dynamic management of resources in a cluster requires portability of the code and execution (Sharifi, 2008).

Cloud computing mechanisms use virtualization in their infrastructure to enhance the sharing of remote resources by providing users with control over remote resources. Clouds also provide a virtualized platform for users to create and manage the software stack from the operating system to the applications (Marshall, 2010). Providing such an infrastructure in a cloud environment needs an effective communications means between VMs.

A critical performance requirement of most distributed systems, like HPC clusters and clouds, is the existence of efficient means of communication between the virtualized execution environments (i.e., virtual machines -- VMs). This requirement has not been supported by VT yet. We thus try to propose a more effective communication mechanism between VMs in this paper, to mitigate the performance degradation of VT such that VT can be more readily deployed in performance critical systems.

Distributed shared memory (DSM) provides efficiency, simplicity and ease of programming for distributed systems (Nitzberg, 1991). SMs and DSMs have various types from non-transparent, less efficient but portable easy to modify middleware implementations (Nitzberg, 1991) to transparent, efficient but non-portable hard to modify kernel level implementations (Trevisan, 2002 & Keleher, 1994). Traditional, most common and efficient way of data communication in distributed systems is to use distributed shared memory (DSM). DSMs provide efficiency, simplicity and ease of programming for distributed systems (Nitzberg, 1991). DSMs may implemented as middleware, which are non-transparent, less efficient but portable and easy to modify (Gobert, 2005), or implemented at kernel level, which are transparent, efficient but non-portable and hard to modify (Keleher, 1994 & Sharifi, 2010).

We believe using VT can aggregates all goods for DSMs. DSM employing VT can be portable and easier to modify while preserving transparency and efficiency. To realize our belief, we propose ZIVM as a DSM replacement for cloud distributed systems, which is efficient, transparent, and highly portable to even heterogeneous systems.

In this paper, we propose a zero-copy inter-VM communication method (ZIVM) in cloud distributed virtual systems as a transparent, efficient, easy to modify, and highly portable replacement for traditional DSM communication, in support of heterogeneous execution environments. We provide transparency in application layer (Radhakrishnan, 2008) of guest OS. ZIVM is portable between different guest OSes and VMMs. In other words, we can use ZIVM to provide a distributed heterogeneous environment.

The rest of paper is organized as follows. A number of inter-VM communication mechanisms are introduced in Section 2 to provide a context for our proposition. Section 3 presents our approach of a new zero copy inter-VM communication mechanism for cluster computing, called ZIVM. Section 4 studies our implementation if ZIVM in KVM hypervisor. Section 5 evaluation of ZIVM using iperf and lmbench benchmarks and Section 6 concludes the paper.

2. Related Work

Security is an important concern in virtualized environments, and VMM provides isolation between VMs. At the same time distributed cloud applications needs efficient communication mechanisms between VMs. In this section, we investigate the related work.

The IVC (Huang, 2007) and VMPI (Diakhaté, 2008) using distributed message-passing paradigm to provide inter-VM communications. This approach is the most useful transition path for minimally modified distributed software architectures into a multi-core environment. With the great benefits for system management, such as fault tolerance, performance isolation, and load balancing, VMs are an attractive solution for HPC (Wang, 2009).

Huang in IVC proposed a library that provides direct-shared memory communication between co-located VMs. The IVC modifies MVAPICH2, an MPI library over InfiniBand, to allow user applications to use a socket-style user API for IVC-aware applications. MVAPICH2-ivc extends MVAPICH2. It automatically chooses between IVC and network communication and hides the complexities of IVC-specific APIs from user applications. IVC sets up shared memory regions through the Xen grant table mechanisms. IVC consists of two parts: a user-space communication library and a kernel driver.

The goal of Diakhate in the VMPI (Diakhaté, 2008) project has been to obtain MPI communications between

VMs in a host. To achieve this goal efficiently, it provides a virtual device that transfer messages between VMs. This virtual device provides message-passing mechanisms for OS. Using this virtual device, it can be received a good portability between VMs, because it just need to implement a driver for guest OS. This project has been implemented in KVM virtual machine monitor, and can obtain near native performance in ping-pong tests. It can outperform native user space MPI implementations without introducing privileged code on the host.

Another approach for providing inter-VM communications is to use a simple shared memory channel between the communicating domains for exchanging network packets, which require fewer hypercalls and bypasses part or whole of the default network datapath (Wang, 2009). Socket-Outsourcing (Eiraku, 2009), XenSockets (Zhang, 2007) Xway (Kim, 2008) and XenLoop (Wang & Wright, 2008) are using this approach.

Eriaku has presented socket outsourcing as a system for providing fast networking between VMs in a host. Socket outsourcing proposed the mechanism that replace socket layer in guest OS with host OS. This mechanism by eliminating further message copies in guest and host OS, improved the network performance in a host. Actually, this method enables network packets to bypass the protocol stack in guest OSes. Socket outsourcing was implemented in two representative operating systems (Linux and NetBSD) and two virtual machine monitors (KVM and PansyVM). These VMMs provided support for socket outsourcing through shared memory, event queues, and VM-specific Remote Procedure Call between a guest and a host OS. Socket outsourcing can achieve native performance, but it is need to made changes in guest OS and VMM, so it cannot be portable between all of guest OSes and VMMs.

XenSocket is a high throughput unidirectional inter-VM communication mechanism that uses a shared memory buffer between communicating VMs to bypass the network protocol stack completely. Receiver VM allocates 128 KB pool of pages and asks the Xen hypervisor to share those pages to the sender VM. These pages are reused in a circular buffer. XenSocket provides a one-way communication pipe. Test results show that XenSocket inter-domain throughput approaches that of inter-process communication using UNIX domain sockets. However, applications and libraries need to be modified to explicitly invoke these calls. In the absence of support for automatic discovery and migration, XenSockets is primarily used for applications that are already aware of the co-location of the other VM endpoint on the same physical machine, and which do not expect to be migrated.

XWay is another shared memory based inter-domain communication mechanism that provides an accelerated communication path between VMs on the same physical machine. Unlike XenSocket, which cannot provide binary compatibility and only provides a one-way communication channel, XWay provides full binary compatibility for applications communicating over TCP sockets. XWay architecture is composed of three layers: switch, protocol, and driver. It takes a significant amount of effort to implement a new socket module that supports all socket options for a new transport layer. To reduce the development effort as much as possible, a virtual socket, called the XWay socket, is introduced. The XWay switch layer transparently switches between TCP socket and XWay protocol. XWay protocol layer supports TCP socket semantics for data send/receive operations such as blocking and non-blocking mode I/O. XWay defines a virtual device and a device driver to represent XWay socket. XWay achieves high performance by bypassing TCP/IP stacks and providing a direct shared-memory communication channel between VMs in the same machine. However, XWay can only support TCP communication, and requires significant changes to the Linux kernel code.

Wang presents XenLoop as a high performance inter-VM network loopback channel in Xen virtual machine monitor. XenLoop intercepts outgoing network packets beneath the network layer and shepherds the packets destined to co-resident VMs through a high-speed inter-VM shared memory channel that bypasses the virtualized network interface. Guest VMs using XenLoop can migrate transparently across machines without disrupting ongoing network communications, and seamlessly switch between the standard network path and the XenLoop channel. XenLoop needs to change the network layer in guest OS and VMM, and has not ability to apply on all of OSes and VMMs.

Radhakrishnan offers an inter-VM communication mechanism called MMNet (Radhakrishnan, 2008) that differs from the previous approaches in that it eliminates data copies across VM kernels by map-ping the entire kernel address space of one VM into the address space of its communicating peer VM in a read-only fashion. MMNet eliminates data copies and hypervisor calls in the critical path by mapping in the entire kernel address space of the peer VM. This approach relaxes memory isolation between VMs. MMNet exports a generic link layer device interface as a loadable kernel module. Security is the major concern of MMNet and VMs in communication cannot fully trust of each other.

3. Proposed Inter-VM Communication Approach

In this section, we propose the architecture, design and implementation issues of the ZIVM. To achieve transparency in application layer of the guest OS, we must choose a way that user application do not aware of distribution of shared memory and how it act. For this reason, we decided to use POSIX standard (Wang, 2008), and provide a POSIX interface for applications. POSIX is a standard for writing portable programs of different OSES and hardware. To provide portability in kernel layer of guest OS, we design a general virtual device that proffer shared memory between VMs to guests. Using a general driver for virtual device, we can obtain application portability in guest OS, this driver provides a POSIX interface for user applications. For different guest OSES we can write drivers for our virtual device, and by adding this driver to guest OSES we can provide a distributed heterogeneous environment in a host. At the rest of paper, we more talk about the implementation of portability and transparency.

3.1 ZIVM Architecture

With this architecture, we can achieve high portability between different guest OSES. Our device is a general PCI device and writing driver for it in different OSES is similar to writing a PCI graphic card, so it is easy to provide a heterogeneous virtual execution environment with this architecture. To achieve portability in VMM we must choose a way that needs minimal changes in VMM. Most VMMs already emulate several devices to provide basic functionality to VMs (network, block device, etc) it is possible to emulate a new virtual device without modifying the core of the VMM, so we can use this mechanism on different VMMs with minimal changes to VMMs, while preserving VMM's portability.

The driver in the guest OS provides a shared memory with POSIX interface to user applications, and handled all the initializations of and communications with the shared memory. This driver provides application, distribution, and location transparency in the guest OS and application is not aware of shared memory location and how it acts.

The virtual device in VMM provides a zero copy shared memory. This virtual device provides a communicational channel between VMs, and each VM has direct access to shared memory. All the shared memory has been mapped to the virtual address space of each VM, and VMs Read and Write directly to the shared memory. Guest OSES does not copy shared object to its own memory because shared memory is a part of its memory and coping overhead of this method is zero. Shared memory is also accessible through host OS. With this virtual device we can obtain transparency in guest OS's kernel, it means that the kernel of guest OS does not aware of shared memory properties and behaviors. Figure 2 shows an overview of shared memory virtual device.

4. ZIVM IMPLEMENTATION

In this section we present details on our preliminary of ZIVM implementation. We are using Linux as both guest and host OS and KVM as hypervisor.

KVM (Kivity, 2007) is a Linux kernel module that allows Linux to act as a hypervisor thanks to hardware virtualization capabilities of newer commodity processors. It is thus capable of virtualizing unmodified guest OSES but also supports paravirtualization to optimize performance critical operations. Using KVM, VMs are standard Linux processes that can be managed and monitored easily. A slightly modified version of QEMU that is a user space component is used to perform various initialization tasks, and emulating VM's devices. As a result, implementing a new virtual device does not introduce additional code into the host kernel. Everything is performed inside QEMU that is a user space process.

4.1 Guest Implementation

As we pointed out in section 3, we need a driver in guest OS that handles shared memory communications and synchronizations. Moreover, this driver provides a POSIX shared memory interface for user applications. We implemented this driver as a Linux driver for Linux guests.

To maximize portability in hypervisor, our virtual device is accessed through the VirtIO interface. This interface abstracts the hypervisor specific virtual device handling instructions into a hypervisor agnostic interface (Russell, 2008). Using VirtIO, the same drivers can be used to handle several hypervisors implementations of a given device. Only one hypervisor specific driver is needed to implement the interface itself. As a result, the same drivers can be used for different implementations of our virtual device in different VMMs. For example, as shown in Figure 3, a KVM VirtIO driver is used to communicate with different implementations of virtual device using the VirtIO interface. VirtIO is similar to Xen paravirtualized device drivers. Until now just network and disk device drivers implemented using VirtIO (Russell, 2008). The VirtIO interface is based on buffer descriptors, which are lists of pointers and sizes packed in an array. Operations include inserting these descriptors into a

queue, signaling the hypervisor and checking if a buffer descriptor has been used.

Our driver has a set of registers that presents the state of shared memory. Our virtual device provides interrupts between VMs, so our driver must support them and present interrupt status as registers. To support simultaneous access to shared memory, we provide semaphore and a wait queue in the driver. The driver provides a POSIX shared memory interface with open, release, read, write, seek and some ioctl functionalities. The ioctls provides system calls for initializing semaphore, calling events on queues and request an interrupt to shared memory. Our driver exposes the virtual device as a character device and uses file descriptor to define ports. The port numbers allow several communication channels to be opened on each virtual machine. We use file descriptors to describe shared objects and access them.

4.2 Host Implementation

The host emulated a shared memory virtual device. It provides a zero copy shared memory between VMs. The virtual device is a pool of shared objects. Our QEMU instances are slightly modified so that they define our virtual device as a PCI device. This device maps a shared memory object as a PCI device in the guest. This device has two BARs, BAR0 for registers and BAR1 for the memory region that maps the shared object. The memory region can be mapped into user space in the guest, or read and written if required.

Interrupts are supported between multiple VMs and host by using a shared memory server that is connected to with a socket character device. The server passes file descriptors for the shared memory object and eventfds (our interrupt mechanism) to the respective qemu instances. When using interrupts, VMs communicate with a shared memory server that passes the shared memory object file descriptor using SCM_RIGHTS. The server assigns each VM an ID number and sends this ID number to the qemu process along with a series of eventfd file descriptors, one per guest using the shared memory server. These eventfds will be used to send interrupts between guests. Each guest listens on the eventfd corresponding to their ID and may use the others for sending interrupts to other guests.

We have two registers for the interrupt mask and status registers. Mask and status are only used with pin-based interrupts. They are unused with MSI interrupts. A read-only register is used to reports the guest's ID number. Interrupts are triggered when a message is received on the guest's eventfd from another VM.

5. Evaluation

Metrics of evaluating how good such system-level approaches are, have a wide variety from non-functional metrics like transparency to more experimental ones like network throughput. We investigate the effectiveness of our proposed approach from two general aspects: 1) features of the approached, and 2) communication and networking performance.

To compare the features of our proposed approach with other credible inter-VM communication works, we gathered a list of effective features and checked whether the approaches has support of that features or not?

To evaluate the performance of communication and networking of our proposed approach, we considered the network throughput and network latency, and compared the benchmark results with other approaches.

5.1 Feature Comparison

To summarize related work and compare them to ZIVM, first we need to introduce the main features of inter-VM communication. These features are (Wang, 2009):

- Guest User transparency: Network applications and/or communication libraries need to be rewritten against new APIs and system calls for IVC, VMPI, Socket-Outsourcing, and XenSocket.
- Guest Kernel Transparency: with IVC, Socket Outsourcing, XWay and XenLoop, guest OS code needs to be modified and recompiled.
- Transparent Migration Support: Socket-Outsourcing, XenLoop and ZIVM can support VM Migration transparently.
- Complete Memory Isolation: Only VMPI and MMNet can not provide complete memory isolation between VMs.
- Location in Software Stack: Socket-Outsourcing uses a new socket protocol. XenSocket and XWay are modified the socket layer. Both MMNet and XenLoop are below the IP layer. ZIVM, IVC and VMPI use a user library and system calls for providing inter-VM communication.
- Copying Overhead: Only ZIVM uses zero copy while XenLoop has 4 copies and others use 2 copies.

- Distributed Support: All of related work has been designed to be used in a host, and only our work has ability to support distributed manner and we need to made changes in the guest OS to act like a cloud distributed system.

Table 1 compares aforementioned inter-VM mechanisms. The main feature of ZIVM is distributed support ability. Moreover it can provide user and guest OS transparency and copying overhead of it is zero.

5.2 Networking Performance

In any communication system, minimize the latency of transmission and maximize the throughput of the communication channel is the main goal (Wang, 2008). Therefore, to evaluate the communication and networking performance of our proposed approach, we measured the latency and throughput of ZIVM. All experiments were performed using a test machine with quad-core AMD Athlon II X4 2.6 GHz processor, 4MB cache, and 4GB main memory. We deployed KVM 88 for the hypervisor and Linux CentOS 5.4 with 2.6.32.4 kernel for the guest OS. We configured two guest VMs on the test machine with 512 MB of memory allocation each for inter-VM communication experiments. Our experiments compare the following communication scenarios:

- KVM when using VirtIO as paravirtualization method.
- XenLoop mechanism in Xen 3.2.0 hypervisor.
- KVM when using ZIVM.

We used an unmodified version of Iperf (Hsing, 1998) and LMBench (McVoy, 1996) benchmarks. Iperf is a tool for measuring various network criterions. LMBench is a suite of portable benchmarks that measures a UNIX system's performance in terms of various bandwidth and latency measurement units. Table 2 compares the measured average bandwidth across three different communication scenarios using Iperf and LMBench benchmarks.

The ZIVM improved the average of bandwidth, because ZIVM provides direct access to shared memory and the packets does not go through network layers. Figure 3 shows the comparison of bandwidth variation versus message size. The bandwidth of ZIVM for small messages was lower than XenLoop and paravirtual KVM, but when the size of messages increased, the bandwidth got higher. The reason for such reaction is in direct access to shared memory, when size of messages is small, the number of access to shared memory is too much and when the size of messages increase the number of accesses is decreased, so higher bandwidth is accessible. To measure latency between two VMs in a host, we use LMBench benchmark and Figure 4 shows the results of comparison between evaluation scenarios. ZIVM improved the latency time, but it is so close to XenLoop.

6. Conclusion and Future Works

In this paper, we presented the design and implementation of ZIVM, a zero-copy inter-VM communication mechanism on a host. ZIVM provides an infrastructure for cloud computing. We implemented this mechanism using a virtual device in hypervisor and created an execution environment using a special driver in each guest. This project prepares an infrastructure for creating reconfigurable virtual clouds. The main features of this project are achieving transparency alongside portability in a zero copy shared memory inter-VM communication mechanism. Our evaluation shows that it achieves higher bandwidth and lower latency in comparison with XenLoop and paravirtual KVM.

In the future, we intend to extend this mechanism and make it distributed between hosts to provide a heterogeneous distributed cloud computing system. We also plan to extend our virtual device so that it supports additional features such as live migration in a host or other hosts, and add VM scheduling policies to reduce overhead and improve performance.

References

- Adams, K., & Agesen, Ole. (2006). A comparison of software and hardware techniques for x86 virtualization. 12th international conference on architectural support for programming languages and operating systems. San Jose. California. USA.
- AMD Virtualization. (2008). AMD-V Nested Paging. White paper. [Online] Available: <http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx>
- Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, L., & Warfield, A. (2003). Xen and the art of virtualization. 19th ACM symposium on operating systems principles. Bolton Landing. NY. USA.

- Bartholomew, D. (2006). QEMU a Multihost Multitarget Emulator. *Linux Journal*.
- Budruk, R., & Anderson, D., Shanley, T. (2003). *PCI Express System Architecture*. Addison-Wesley.
- Diakhaté, F., Perache, M., Namyst, R., & Jourden, H. (2008). Efficient Shared Memory Message Passing for Inter-VM Communications. 3rd Workshop on Virtualization in High-Performance Cluster and Grid Computing (VHPC'08), as part of Euro-Par, Las Palmas de Gran Canaria, Canary Island, Spain.
- Eiraku, H., Shinjo, Y., Pu, C., Koh, Y., & Kato, K. (2009). Fast Networking with Socket Outsourcing in Hosted Virtual Machine Environments. ACM Symposium on Applied Computing, Hawaii, USA.
- Gobert, J., Rubini, A., & Hartman, G. K. (2005). *Linux Device Drivers*. Third Edition. O'Reilly Media.
- Hsing, C.H., & Kremer, U. (1998). *IPERF: A Framework for Automatic Construction of Performance Prediction Models*. First Workshop on Profile and Feedback-Directed Compilation. Paris. France.
- Huang, W., Santhanaraman, G., Jin, H. W., Gao, Q., & Panda, D. K. (2006). Design and Implementation of High Performance MVAPICH2: MPI2 over InfiniBand. Int'l Symposium on Cluster Computing and the Grid (CCGrid). Singapore.
- Huang, W., Koop, M.J., Gao, Q., & Panda, D.K. (2007). Virtual machine aware communication libraries for high performance computing. ACM/IEEE conference on Supercomputing. Reno. Nevada.
- Intel Corporation. Intel Cluster Toolkit 3.0 for Linux.
- Keleher, P., Dwarkadas, S., Cox, A.L., & Waenepoel, W.Z. (1994). Treadmarks: distributed shared memory on standard workstations and operating systems. Winter 1994 USENIX Conference. San Francisco. California.
- Kim, K., Kim, C., Jung, S., Shin, H., & Kim, J. S. (2008). Inter-domain socket communications supporting high performance and full binary compatibility on Xen, 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. Seattle. WA. USA. <http://dx.doi.org/10.1145/1346256.1346259>
- Kivity, A., Kamay, Y., Laor, D., Lublin, D., & Liguori, A. (2007). *KVM: the Linux virtual machine monitor*. Linux Symposium 2007. Ottawa. Canada.
- Lewine, D.A. (1991). *POSIX Programmer's Guide Writing Portable UNIX Programs with the POSIX.1 Standard*. O'Reilly & Associates.
- Marshall, P., Keahey, K., & Freeman, T. (2010). Elastic Site: Using Clouds to Elastically Extend Site Resources. IEEE/ACM International Symposium on Cluster. Cloud and Grid Computing (CCGrid 2010). Melbourne. Australia.
- Marshall, D., Reynolds, W., & McCrory, D. (2006). *Advanced Server Virtualization: VMware and Microsoft Platforms in the Virtual Data Center*. Auerbach Publications. ISBN 0-8493-3931-6. <http://dx.doi.org/10.1201/9781420013160>
- McVoy, L., & Staelin, C. (1996). Lmbench: Portable Tools for Performance Analysis. USENIX Annual Technical Conference. San Diego. CA.
- Mergen, M.F., Uhlig, V., Krieger, O., & Xenidis, J. (2006). Virtualization for High Performance Computing. *SIGOPS Operating Systems Review*.
- Neiger, G., Santoni, A., Leung, F., Rodgers, D., & Uhlig, R. (2006). Intel virtualization technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*.
- Nitzberg, B., & Lo, V. (1991). Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8), 52-60.
- Pfister, G.F. (2001). Aspects of the InfiniBand(tm) Architecture. Proceedings of the 3rd IEEE International Conference on Cluster Computing. California. USA.
- POSIX: 2008. (2008). The Austin Common Standards Revision Group", The Open Group. [Online] Available: <http://www.opengroup.org/austin/>
- Radhakrishnan, K., & Srinivasan, K. (2008). Mmnet: An Efficient Inter-VM Communication Mechanism. Xen Summit. Boston.
- Russell, S. (2008). Virtio: Towards a de-facto Standard for Virtual I/O Devices. publication of the *ACM Special Interest Group on Operating Systems (SIGOPS)*, 42(5).
- Sharifi, M., & Hasani, M. (2008). VCE: A New Personated Virtual Cluster Engine for Cluster Computing. The 3rd IEEE International Conference on Information and Communication Technologies: From Theory to

Applications (ICTTA'08).

Sharifi, M., Mousavi K. E., Kashyian M., & Mirtaheri S. L. (2010). A Platform Independent Distributed IPC Mechanism in Support of Programming Heterogeneous Distributed Systems, *Journal of Supercomputing*, Springer, DOI: 10.1007/s11227-010-0452-9, Published Online: 25 June 2010.

Smith, J.E., & Nair, R. (2005). *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann. ISBN 1-55860-910-5.

Trevisan, T.S., Costa, V.S., Whately, L., & Amorim, C.L. (2002). Distributed Shared Memory in Kernel Mode. The 14th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'02). Vitoria/ES Brazil.

Wang, J. (2009). Survey of State-of-the-art in Inter-VM Communication Mechanisms. *Research Proficiency Report*. Binghamton University.

Wang, J., Wright, K.L., & Gopalan, K. (2008). XenLoop: a Transparent High Performance Inter-VM Network Loopback. *HPDC08*, Boston.

Zhang, X., McIntosh, S., Rohgati, P., & Griffin, J. (2007). XenSocket: A High-Throughput Interdomain Transport for Virtual Machines. *Middleware*.

Table 1. A feature-wise comparison of Inter-VM mechanisms ZIVM

Approaches Features	IVC	VMPI	Socket Outsourcing	XenSocket	XWay	XenLoop	MMNet	ZIVM
Guest Application Transparency	✗	✗	✗	✗	✓	✓	✓	✓
Guest Kernel Transparency	✗	✓	✗	✓	✗	✗	✓	✓
Transparent Migration Support	✗	✗	✓	✗	✗	✓	✗	✓
Complete Memory Isolation	✓	✗	✓	✓	✓	✓	✗	✓
Location in Software Stack	UserLib + SysCalls	UserLib + SysCalls	Socket Layer	Below Socket Layer	Below Socket Layer	Below IP Layer	Below IP Layer	UserLib + SysCalls
Copying Overhead	2	2	2	2	2	4	2	Zero
Distributed Support	✗	✗	✗	✗	✗	✗	✗	✓

Table 2. Average bandwidth comparison

	XenLoop	KVM-VirtIO	KVM-ZIVM
Iperf (Mbps)	139.3	137.6	139.3
LMbench (Mbps)	141.6	139.1	143.8

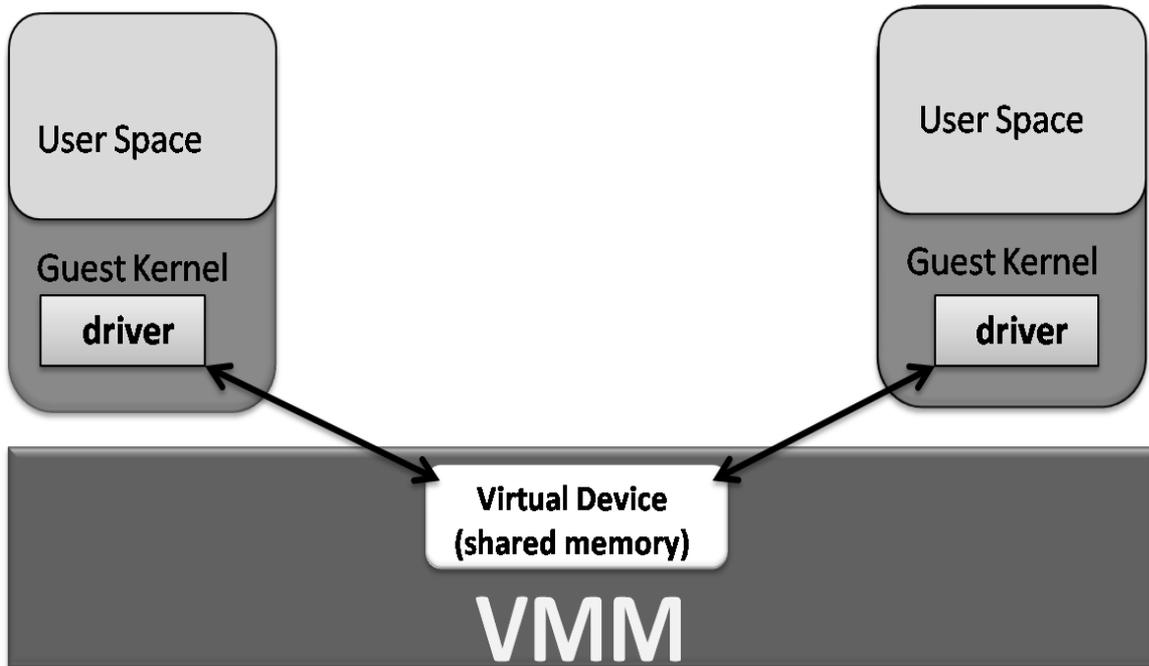


Figure 1. ZIVM architecture

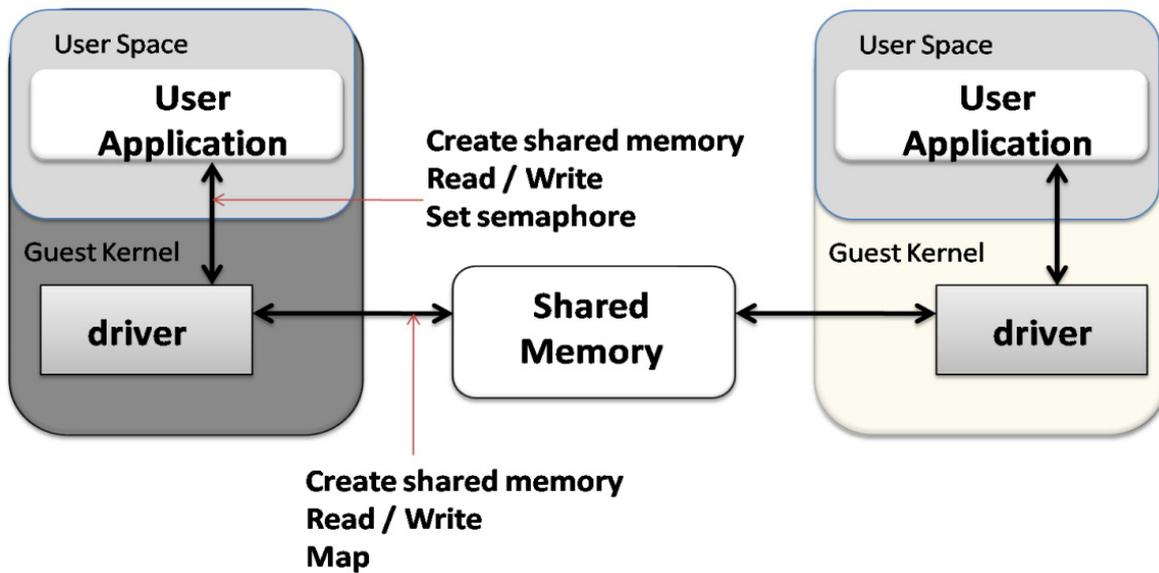


Figure 2. Overview of shared memory virtual device

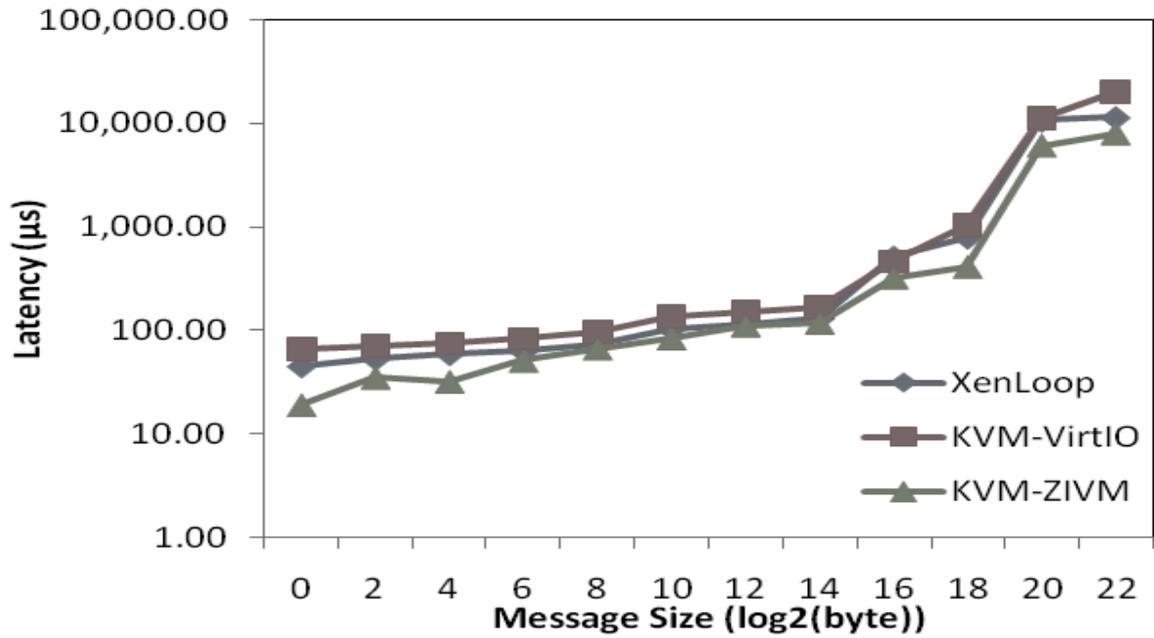


Figure 3. Bandwidth versus message size using LMBench benchmark

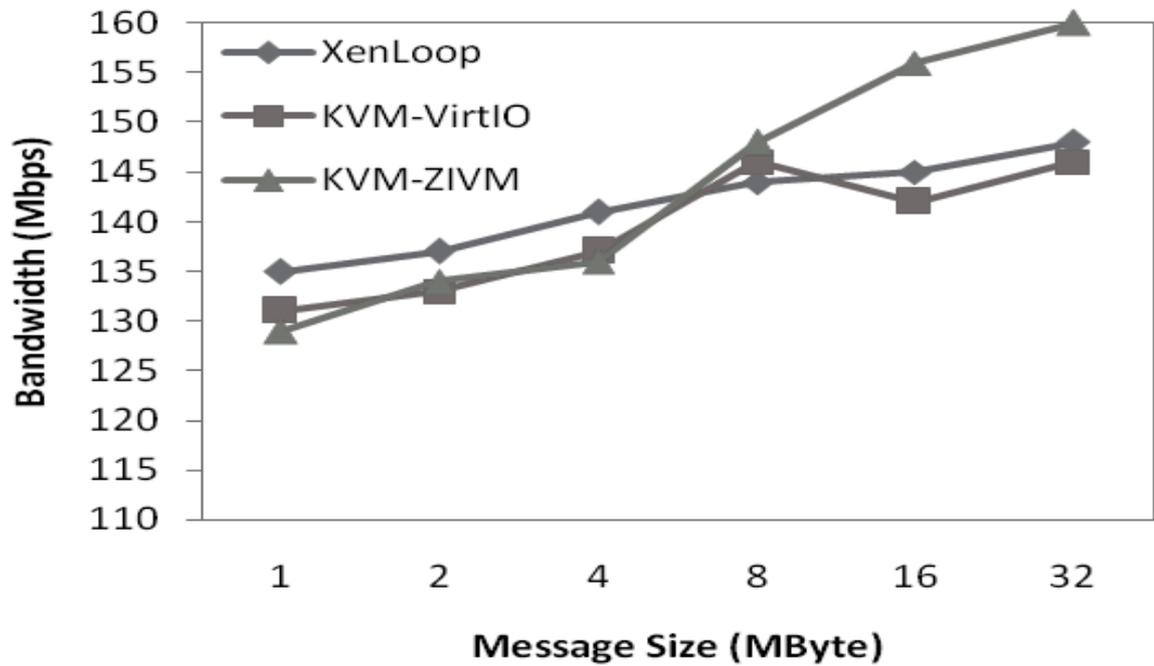


Figure 4. Latency versus message size using LMBench benchmark