Shared-Memory Optimizations for Inter Virtual Machine Communication[1]

YI REN, National University of Defense Technology
LING LIU and QI ZHANG, Georgia Institute of Technology
QINGBO WU, JIANBO GUAN, JINZHU KONG, HUADONG DAI and LISONG SHAO, National University of Defense Technology

Virtual machines (VMs) and virtualization are one of the core computing technologies today. Inter-VM communication is not only prevalent but also one of the leading costs for data intensive systems and applications in most of the datacenters and cloud computing environments. One way to improve inter-VM communication efficiency is to support co-resident VM communication using shared memory based methods and resort to the traditional TCP/IP for communications between VMs that are located on different physical machines. In recent years, several independent kernel development efforts have been dedicated to improving communication efficiency between co-resident VMs using shared memory channels, and the development efforts differ from one another in terms of where and how the shared memory channel is established. In this paper, we provide a comprehensive overview of the design choices and techniques for performance optimization of co-resident inter-VM communication. We examine the key issues for improving inter-VM communication using shared memory based mechanisms, such as implementation choices in the software stack, seamless agility for dynamic addition or removal of co-resident VMs, multilevel transparency, as well as advanced requirements in reliability, security and stability. An in-depth comparison of state-of-the-art research efforts, implementation techniques, evaluation methods and performance is conducted. We conjecture that this comprehensive survey will not only provide the foundation for developing the next generation of inter-VM communication optimization mechanisms, but also offers opportunities to both cloud infrastructure providers and cloud service providers and consumers for improving communication efficiency between co-resident VMs in virtualized computing platforms.

Categories and Subject Descriptors: D.4.4 [**Operating Systems**]: Communications Management — Network communication; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design — Network communications; C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.3 [**Computer-Communication Networks**]: Network Operations

General Terms: Design, Performance

Additional Key Words and Phrases: Residency-aware, inter virtual machine communication, shared memory, seamless agility, multilevel transparency.

1. INTRODUCTION

Virtual machines are the creations of hardware virtualization. Unlike physical machines, software running on virtual machines is separated from the underlying hardware resources. Virtual machine monitor (VMM or hypervisor) technology enables a physical machine to host multiple guest virtual machines (VMs) on the same hardware platform. As a software entity, VMM runs at the highest system privilege level and coordinates with a trusted VM, called host domain (Dom0) or host OS, to enforce isolation across VMs residing on a physical machine. Each of the VMs is running on a guest domain (DomU) with its own operating system (guest OS). To

[1]The authors from NUDT were supported by grants from National Nature Science Foundation of China (NSFC) under grant NO. 60603063, Young Excellent Teacher Researching and Training Abroad Program of China Scholarship Council (CSC), and the authors from Georgia Tech is partially supported by USA NSF CISE NetSE program, SaTC program, IUCRC FRP program and a grant from Intel ISTC on Cloud Computing.

An earlier version of this paper appeared in the proceedings of IEEE 6th International Conference on Cloud Computing. June 27-July 2, 2013, Santa Clara, CA, USA, titled "Residency-aware virtual machine communication optimization: design choices and techniques".

Authors' Address: Y. Ren (corresponding author), College of Computer Science, National University of Defense Technology, 47 Yanwachi St., Changsha 410073, Hunan, P. R. China; Email: renyi@nudt.edu.cn; L. Liu, Q. Zhang, College of Computing, Georgia Institute of Technology, 266 Ferst Dr., Atlanta, GA 30332-0765, USA; Q. Wu, J. Guan, J. Kong, H. Dai, and L. Shao, College of Computer Science, National University of Defense Technology, 47 Yanwachi St., Changsha 410073, Hunan, P. R. China.

date, VMM based solutions have been widely adopted in many data centers and cloud computing environments [Armbrust et al. 2010; Gurav and Shaikh 2010; YOUNGE et al. 2011; Anderson et al. 2013].

1.1 Problems of Co-Resident VM Communication and Related Work

It is well known that the VMM technology benefits from two orthogonal and yet complimentary design choices. *First,* VMM technology by design enables VMs residing on the same physical machine to share resources through time slicing and space slicing. *Second,* VMM technology introduces host-neutral abstraction, which treats all VMs as independent computing nodes regardless of whether these VMs are located on the same host machine or different hosts.

Although VMM technology offers significant benefits in terms of functional isolation and performance isolation, live migration enabled load balance, fault tolerance, portability of applications, and higher resources utilization, both design choices carry some performance penalties. *First,* VMM offers significant advantages over native machines when VMs co-resident on the same physical machine are not competing for computing and communication resources. However, when co-resident VMs are competing for resources under high workload demands, the performance of those co-resident VMs is degraded significantly compared to the performance of native machine, due to the high overheads of switches and events in host/guest domain and VMM [Pu et al. 2013; Mei et al. 2013]. Furthermore, different patterns of high workload demands may have different impacts on the performance of VM executions [Wang et al. 2011; Imai et al. 2013]. Second, several research projects [Huang et al. 2007; Huang 2008; Kim et al. 2008; Wang et al. 2008b; Radhakrishnan and Srinivasan 2008; Ren et al. 2012] have demonstrated that even when the sender VM and receiver VM reside on the same physical machine, the overhead of shipping data between co-resident VMs can be as high as the communication cost between VMs located on separate physical machines. This is because the abstraction of VMs supported by VMM technology does not differentiate whether the data request is coming from the VMs residing on the same physical machine or from the VMs located on a different physical machine. Concretely, Linux guest domain shows lower network performance than native Linux [Menon et al. 2006; Liu et al. 2006; SANTOS et al. 2008; YEHUDA et al. 2006; RUSSELL 2008; Li et al. 2010], when an application running on a VM communicates with another VM. Although with copying mode in Xen 3.1, [Kim et al. 2008] shows that the inter-VM communication performance is enhanced but still significantly lagging behind compared to the performance on native Linux, especially for VMs residing on the same physical machine.

The two main reasons for the performance degradation of co-resident VM communication are [Wang 2009]: (i) long communication data path through the TCP/IP network stack [Kim et al. 2008; Wang et al. 2008b; Ren et al. 2012], (ii) lack of communication awareness in CPU scheduler and absence of real time inter-VM interactions [Govindan et al. 2007; Kim et al. 2009; Ongaro et al. 2008]. The first category of solutions to improve the performance of inter-VM communication is to use a shared memory channel mechanism for communication between co-resident VMs to improve both communication throughput and latency [Huang et al. 2007; Huang 2008; Zhang et al. 2007; Kim et al. 2008; Wang et al. 2008b; Radhakrishnan and Srinivasan 2008; Ren et al. 2012; Zhang et al. 2015; Diakhaté et al. 2008; Eirakuet al. 2009; Koh 2010; Gordon 2011a; Gordon et al. 2011b; Ke 2011; Macdonell 2011]. An alternative category of solutions is to reduce inter-VM communication latency by optimizing CPU scheduling policies [Govindan et al. 2007; Kim et al. 2009; Ongaro et al. 2008; Lee et al. 2010]. Research efforts in this category show that by optimizing CPU scheduling algorithms at the hypervisor level, both inter-VM communication and I/O latency can be improved. For instance, [Govindan et al. 2007] introduces an enhancement to the SEDF scheduler that gives higher priority to I/O domains over the CPU intensive domains. [Ongaro et al. 2008] provides specific optimizations to improve the I/O fairness of the credit scheduler. Task-aware scheduling [Kim et al. 2009] focuses on improving the performance of I/O intensive tasks within different types of workloads by lightweight partial boosting mechanism. Incorporating the knowledge of soft real-time applications, [Lee et al. 2010] proposes a new scheduler based on the credit scheduler to reduce the latency. However, none of the work in this category actually deals with the inter-VM communication workloads or take co-located VM communications into consideration.

In this paper, we focus on reviewing and comparing the solutions in the first category, namely improving inter-VM communication efficiency using shared memory based mechanisms. To date, most of the kernel research and development efforts reported in the literature are based on shared memory for improving communication efficiency among co-resident VMs. Furthermore, most of the documented efforts are centered on open source hypervisors, such as Xen platform and KVM platform[2]. Thus, in this paper we present a comprehensive survey on shared memory based techniques that are implemented on either Xen or KVM platform for inter-VM communication optimization.

Relatively speaking, there are more shared memory efforts on Xen platform than that on KVM platform in the literature, such as IVC [Huang et al. 2007; Huang 2008], XenSocket [Zhang et al. 2007], XWAY [Kim et al. 2008], XenLoop [Wang et al. 2008b], MMNet [Radhakrishnan and Srinivasan 2008] on top of Fido [Burtsevet al. 2009] and XenVMC [Ren et al. 2012]. While all KVM-based efforts, such as VMPI [Diakhaté et al. 2008], Socket-outsourcing [Eiraku et al. 2009; Koh 2010] and Nahanni [Gordon 2011a; Gordon et al. 2011b; Ke 2011; Macdonell 2011], are recent development since 2009. One reason could be that Xen open source was made available since 2003 and KVM is built on hardware containing virtualization extensions (e.g., Intel VT or AMD-V) that were not available until 2005. Interestingly, even the development efforts on the same platform (Xen or KVM) differ from one another in terms of where in the software stack the shared memory channel is established and how the inter-VM communication optimization is carried out.

We see a growing demand for a comprehensive survey of the collection of concrete techniques and implementation choices on inter-VM communication optimization. Such comprehensive study not only can help researchers and developers to design and implement the next generation inter-VM communication optimization technology, but also offers cloud service providers and cloud service consumers an opportunity to further improve the efficiency of inter-VM communication in virtualized computing platforms.

1.2 Scope and Contributions of the Paper

This paper provides a comprehensive survey of the literature on co-resident inter-VM communication methods, focusing on the design choices and implementation techniques for optimizing the performance of the co-resident inter-VM communication. We make two original contributions.

[2]VMCI Socket of VMware [VMware Inc. 2007] was a commercial implementation to improve the efficiency of inter-VM communication using a shared memory approach. From the publically available documentations on VMCI, such as the description of its API, its programming guide, etc., we found that VMCI introduces AF_VMCI, a new socket interface, which is not compatible to the current standard interface, and VMCI socket is implemented below system calls layer and above transport layer. Unfortunately, we could not find more detail about VMCI and thus we did not include VMCI in this article.

—*First*, we present the core design guidelines and identify a set of key issues for improving inter-VM communication using shared memory based mechanisms, including choices of implementation layer in the software stack, seamless agility for dynamic addition or removal of co-resident VMs, multilevel transparency, as well as advanced requirements in reliability, security and stability. By seamless agility, we mean that residency-aware inter-VM communication mechanisms support dynamic addition or removal of co-resident VMs including VM creation, VM shutdown, VM live migration, etc., in an automatic and transparent way. We conjecture that through this in-depth study and evaluation, we provide better understanding of the set of criteria for designing the next generation shared memory channel for co-resident VMs.

—*Second,* we conduct a comprehensive survey of representative state-of-the-art research efforts on both Xen and KVM platforms using a structured approach. Concretely, we provide an in-depth analysis and comparison of existing implementation techniques with respect to the architectural layout, the fundamental functionalities, how they achieve the design goals, additional implementation choices, the software environment and source code availability, as well as inter-VM communication efficiency. To the best of our knowledge, this is the first effort that provides a comprehensive survey of the Inter-VM communication methods from three different perspectives: implementation layer in software stack, seamless agility and multilevel transparency.

The rest of this paper is organized as follows. Section 2 provides an overview of the basic concepts and terminology of network I/O and shared memory structures in VMMs, which are fundamental for understanding the design choices and functional requirements of shared memory based inter-VM communication optimization. Section 3 presents design guidelines and key issues about shared memory based communication mechanisms for co-resident VMs, including high performance, seamless agility and multilevel transparency. Based on the functional requirements and design choices outlined in Section 3, we provide a comprehensive comparison of existing work in five subsequent sections: architectural layout comparison in Section 4, common fundamental functionalities in Section 5, the support for seamless agility, including race condition handling, in Section 6, the multilevel transparency in Section 7, and other implementation consideration and optimizations, including buffer size, software environment, source code availability and performance comparison in Section 8. We conclude the paper in Section 9.

2. BACKGROUND AND PRELIMINARY

VMM technology to date is broadly classified into two categories [Pearce et al. 2013]: Type I VMMs, which are hypervisor based VMMs running directly on hardware, such as Xen and VMware ESX Server, and Type II VMMs, which are also known as hosted VMMs and represented by Linux KVM and VMware Workstation. Since the technical details of related work in industry is not publically available, in this paper, we focus primarily on representative open source VMMs, such as Xen and KVM. In this section, we give a brief description of network I/O and communication mechanisms among domains for Xen and KVM respectively.

2.1 Xen Network Architecture and Interfaces

2.1.1. Network I/O Architecture. Xen is a popular open-source x86/x64 hypervisor, which coordinates the low-level interaction between VMs and physical hardware [Barham et al. 2003]. It supports both full-virtualization and para-virtualization. With full-virtualization, in virtue of hardware-assisted virtualization technology, the hypervisor lets VMs run unmodified operating systems. For each guest domain, it provides full abstraction of the underlying physical system. In contrast, with para-virtualization, it requires the guest OS to be modified and no hardware-assisted virtualization technology is needed. The para-virtualization mode provides a more efficient and lower overhead mode of virtualizations.

In para-virtualization mode, Dom0, a privileged domain, performs the tasks to create, terminate and migrate guest VMs. It can also access the control interfaces of the hypervisor. The hypervisor utilizes asynchronous hypercalls to deliver virtual interrupts and other notifications among domains via Event Channel.

Xen exports virtualized network devices instead of real physical network cards to each DomU. The native network driver is expected to run in the Isolated Device Domain (IDD), which is typically either Dom0 or a driver specific VM. The IDD hosts a backend network driver. An unprivileged VM uses its frontend driver to access interfaces of the backend daemon. Figure 1 illustrates the network I/O architecture and interfaces. The frontend and the corresponding backend exchange data by sharing memory pages, either in copying mode or in page flipping mode. The sharing is enabled by Xen Grant Table mechanism that we will introduce later in this section. The bridge in IDD handles the packets from the network interface card (NIC) and performs the software-based routine in the receiver VM.

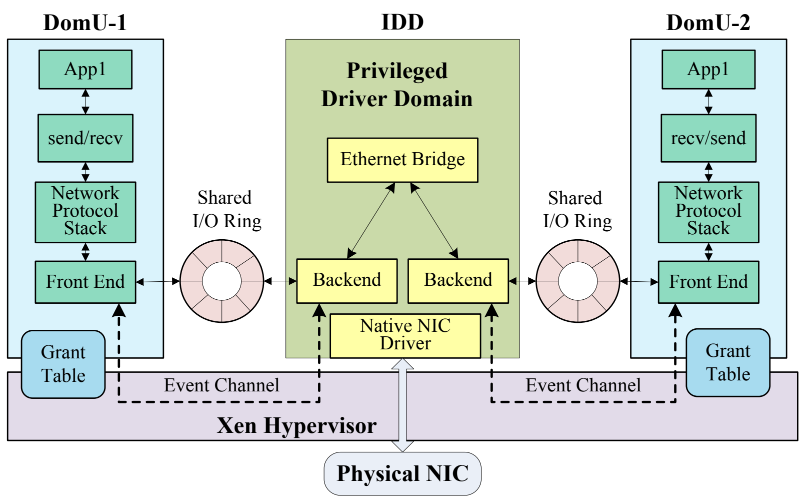


Fig.1. Xen network I/O architecture and interfaces

When the NIC receives a network packet, it will throw an interrupt to the upper layer. Before the interrupt reaches IDD, Xen hypervisor handles the interrupt. First, it removes the packet from NIC and forwards the packet to the bridge. Then the bridge de-multiplexes the packet and delivers it to the backend interface corresponding to the receiver VM. The backend raises a hypercall to the hypervisor for page remapping so as to keep the overall memory allocation balanced. After that, data is copied and the receiver VM gets the packet as if it comes from NIC. The process of packet sending is similar but performed in a reverse order.

Each I/O operation in split I/O model requires involvement of the hypervisor or a privileged VM, which may become a performance bottleneck for systems with I/O intensive workloads. VMM-bypass I/O model allows time-critical I/O operations to be processed directly by guest OSes without involvement of the hypervisor or a privileged VM. High speed interconnects, such as InfiniBand [Infiniband Trade Association. 2015], can be supported by Xen through VMM-bypass I/O [Liu et al. 2006].

2.1.2. Communication Mechanisms among Domains. Xen provides the shared I/O Ring buffers between the frontend and the backend, as shown in Figure 1. These I/O ring buffers are built upon the Grant Table and Event Channel mechanisms, two main communication channels for domains provided by Xen. Grant Table is a generic mechanism to share pages of data between domains in both page mapping mode and page transfer mode. The Grant Table contains the references of granters. By using the references, a grantee can access the granter's memory. Grant Table mechanism offers a fast and secure way for DomUs to receive indirect access to the network hardware through Dom0. Event Channel is an asynchronous signal mechanism for domains on Xen. It supports inter/intra VM notification and can be bound to physical/virtual interrupt requests (IRQs).

In addition, Xen provides XenStore as a configuration and status information storage space shared between domains. The information is organized hierarchically. Each domain gets its own path in the store. Dom0 can access the entire path, while each DomU can access only its owned directories. XenStore has been utilized by some existing projects as a basic mechanism to facilitate the tracing of dynamic membership update of co-resident VMs.

2.2 QEMU/KVM

KVM is an open source full virtualization solution for Linux on x86 hardware which supports virtualization extension. It consists of two components: one is QEMU, which is a hardware emulator running on the host Linux as a user-level process and provides I/O device model for VM; the other is a loadable KVM kernel device driver module, which provides core virtualization infrastructure including virtual CPU services for QEMU and supports functionalities such as VM creation, VM memory allocation. QEMU communicates with the KVM module through a device file, */dev/kvm*, which is created when the KVM module is loaded into the kernel. The KVM module is capable of supporting multiple VMs. QEMU/KVM virtualizes network devices of the host OS to allow multiple guest OSes running in different VMs to access the devices concurrently.

KVM also supports two modes of I/O virtualization: full virtualization through device emulation and para-virtualization I/O model by Virtio [Russell 2008].

2.2.1. Network I/O through Device Emulation. Emulated devices are software implementations of the hardware, such as E1000, RTL8139. Device emulation is provided by the user space QEMU. It makes the guest OS using the device interacts with the device as if it were actual hardware rather than software. There is no need to modify corresponding device driver in the guest OS.

Figure 2 illustrates the architecture of KVM full virtualized network I/O. When the guest OS tries to access the emulated hardware device, the I/O instruction traps into the KVM kernel module. Then the module forwards the requests to QEMU. Then QEMU asks the guest OS to write data into the buffer of the virtualized NIC (VNIC) and copies the data into the TAP device. And the data is forwarded to the TAP device of the destination guest OS by the software bridge. When the TAP device receives the data, it wakes up the QEMU process. QEMU process first copies the data into its VNIC buffer, from where copies the data to the virtual device in the destination guest OS. Then, QEMU notifies the KVM kernel module to receive the data. The KVM kernel module sends interrupts to notify the guest OS about the data arrival. Finally, through virtual driver and network protocol stack, the data is passed to the corresponding applications.

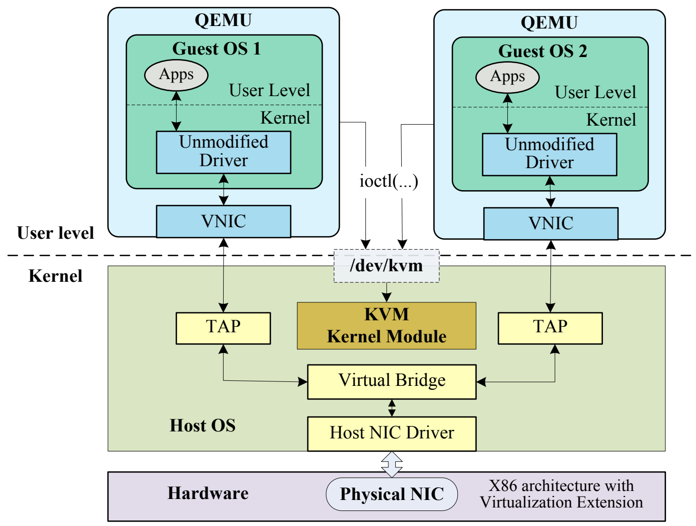


Fig.2. The architecture of KVM full virtualized network I/O

2.2.2. Virtio based Network I/O. Although emulated devices offer broader compatibility than para-virtualized devices, the performance is lower due to the overhead of context switches across the barrier between user-level guest OS and Linux kernel (host OS), as well as the barrier between Linux kernel and user space QEMU. Therefore, Virtio, a para-virtualized I/O framework was introduced. Virtio device driver consists of two parts: the front-end is in the guest OS, while the back-end is in the user space QEMU. The front-end and the back-end communicate with a circular buffer, through which the data can be copied directly from the kernel of host OS to the kernel of guest OS. Virtio avoids some unnecessary I/O operations. It reduces the number of data copies and context switches between the kernel and user space. It provides better performance than device emulation approaches.

3. OVERVIEW OF DESIGN CHOICES

In this section, we first discuss the motivation by comparing TCP/IP and shared memory based approaches to better understand the design guidelines we promote in this paper. Then we present the design objectives for optimizing co-resident inter-VM. We elaborate on the design choices and identify the key issues in designing and implementing a high performance and light weight inter-VM communication protocol based on shared memory mechanisms.

3.1 TCP/IP vs. Shared Memory based Approaches

Modern operating systems, such as Linux, provide symmetrical shared memory facilities between processes. Typical inter process communication mechanisms in Linux are System V IPC and POSIX IPC. With the IPC mechanisms, the processes communicate in a fast and efficient manner through shared memory or message channels since data shared between processes can be immediately visible to each other [Renesse 2012].

TCP/IP is pervasively used for inter-VM communication in many virtualized data centers, regardless whether the VMs are co-resident on the same physical machine or separate physical machines. However, TCP/IP, originally designed for communication among different computer hosts interconnected through a communication network, and it is not optimized for communication between VMs residing on the same physical machine.

Compared with IPC, communication via TCP/IP based network protocol takes longer time because the data transfer from a sender to a receiver has to go through the TCP/IP protocol stack. Concretely, with native TCP/IP network, if a sender process wants to transmit data to a receiver process, first the data is copied from the user space to the kernel space of the sender VM. Then it is mapped to the memory of the network device, and the data is forwarded from the sender VM to the network device of the receiver VM via the TCP/IP network. After that, the data is mapped from the network device to the kernel space of the receiver VM and copied to the user space of the receiver process. Inter-VM communication adds another layer of kernel software stack that the data has to travel along the path from the sender VM to the receiver VM via VMM. On Xen platform, with Xen virtualized frontend-backend network, after the data reaches the buffer of NIC in the sender VM, the data is then transferred to the bridge in the IDD of the sender VM. Via TCP/IP, the data is routed to the corresponding backend of the IDD on the host of the receiver VM. Finally, the data is copied to the receiver VM. In summary, the data transferred from the sender VM to the receiver VM typically goes through a long communication path via VMM on sender VM's host, TCP/IP stack and VMM on receiver VM's host. Similarly, on KVM platforms, data transferred between the sender VM and the receiver VM also incurs multiple switches between VM and VMM in addition to going through the TCP/IP stack.

By introducing shared memory based approaches to bypass traditional TCP/IP protocol stack for co-resident VM communication, we may obtain a number of performance optimization opportunities: (i) the number of data copies is reduced by shortening the data transmission path, (ii) unnecessary switches between VM and VMM are avoided by reducing dependency to VMM, and (iii) using shared memory also makes data writes visible immediately. In short, shared memory based approaches have the potential to achieve higher communication efficiency for co-resident VMs.

3.2 Design Objectives

The ultimate goal of introducing a fast co-resident VM communication mechanism to co-exist with the TCP/IP based inter-VM communication protocol is to improve the performance by shortening the data transfer path and minimizing the communication overhead between co-resident VMs. Concretely, when the sender VM and the receiver VM are co-resident on the same host, the data will be transmitted via the local shared memory channel (local mode) and bypass the long path of TCP/IP network stack. When the sender VM and the receiver VM reside on different hosts, the data will be transferred from sender to receiver through traditional TCP/IP network channel (remote mode). To establish such a shared memory based inter-VM communication channel, the following three core capabilities should be provided: (i) intercept every outgoing data request, examine and detect whether the receiver VM is co-resident with the sender VM on the same host, (ii) support both local and remote inter-VM communication protocols, and upon detection of local inter-VM communication, automatically switching and redirecting the outgoing data request to the shared memory based channel instead, and (iii) the shared memory based inter-VM communication should be incorporated into the existing virtualized platform in an efficient and fully transparent manner over existing software layers [Burtsev et al. 2009; Eiraku et al. 2009; Kim et al. 2008; Radhakrishnan and Srinivasan 2008; Ren et al. 2012; Wang et al. 2008b; Wang 2009]. More importantly, the implementation of a shared memory based approach to inter-VM communication should be highly efficient, highly transparent and seamlessly agile in the presence of VM live migration and live deployment.

The objective of high efficiency aims at significant performance enhancement of different types of network I/O workloads, including both transactional and streaming TCP or UDP workloads.

The objective of seamless agility calls for the support of on-demand detection and automatic switch between local mode and remote mode to ensure that the VM live migration and the VM deployment agility are retained.

Finally, the objective of multi-level transparency of the shared memory based mechanism refers to the transparency over programming languages, OS kernel and VMM. Such transparency ensures that there is no need for code modifications, recompilation or re-linking to support legacy applications.

We argue that high performance, seamless agility and transparency should be fully respected when co-resident VM communication optimization is incorporated in an existing virtualization platforms, be it Xen or KVM or any other VMM technology. We also would like to point out that a concrete implementation choice should make a careful tradeoff among conflicting goals. We will elaborate each of these three objectives in the subsequent sections respectively. In addition, we conjecture that the next generation of shared memory based inter-VM communication facilities should support other desirable features, such as reliability, security and stability, which will be elaborated in Section 3.6.

3.3 Design Choices on High Performance and High Efficiency

The performance of a shared memory inter-VM communication mechanism depends on a number of factors, such as the choice of implementation layer in the software stack, the optimization for streaming network I/O workloads, and the support for necessary network I/O optimization.

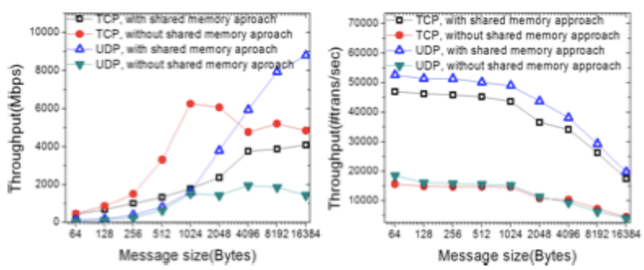
Implementation layer in software stack brings potential impacts on programming transparency, kernel-hypervisor level transparency, seamless agility and performance efficiency. Based on the choice of which layer the shared memory based inter-VM communication protocol is implemented, existing approaches can be classified into following three categories.

3.3.1. User Libraries and System Calls Layer (Layer 1). This is the simplest and most straightforward way to implement a shared memory based inter-VM communication protocol. Concretely, we can simply modify the standard user and programming interfaces in layer 1. This approach introduces less switching overhead and fewer data copies for crossing two protection barriers: from guest user-level to guest kernel-level and from guest OS to host OS. However, it exposes the shared memory to the user-level applications running on guest OSes and sacrifices user-level programming transparency. Most of the existing projects that choose the layer 1 approach fall into the following two categories: (i) in the HPC environment where specific interfaces based communication dominates, such as MPI (Message Passing Interface); and (ii) the earlier efforts of developing co-located VM communication mechanisms on the KVM platform.

3.3.2. Below System Calls Layer, above Transport Layer (Layer 2). An alternative approach to implement shared memory based inter-VM communication is below system calls layer, above transport layer. There are several reasons of why the layer 2 solutions may be more attractive. Due to hierarchical structure of TCP/IP network protocol stack, when data is sent through the stack, it has to be encapsulated with additional headers layer by layer in the sender node. Furthermore, when the encapsulated data reaches the receiver node, the headers will be removed layer by layer accordingly. However, if the data is intercepted and redirected in a higher layer in the software stack, it will lead to two desirable results: smaller data size and shorter processing path (less processing time on data encapsulation and the reverse process) [Wang et al. 2008b]. Based on this observation, we argue that implementation in higher layer can potentially lead to lower latency and higher throughput of network I/O workloads. In contrast, establishing the shared memory channel in layer 1 makes it very hard to maintain programming language transparency. Hence, layer 2 solutions are more attractive compared to layer 1 solutions [Ren et al. 2012].

3.3.3. Below IP Layer (Layer 3). The third alternative method is to implement the shared memory based inter-VM communication optimization below the IP layer. The advantages of this approach over those implemented at layer 1 or layer 2 include: (i) TCP/IP features, such as reliability and so on, are left intact and remain to be effective; (ii) existing third party tool, such as netfilter [Ayuso 2006], remains to be available for hooking into TCP/IP network path to facilitate the implementation of packets interceptions. However, layer 3 is the lowest in the software stack. Thus, it potentially leads to higher latency due to higher network protocol processing overheads, more data copy operations and context switches across barriers.

3.3.4. Problems with existing solutions. Implementing shared memory based inter-VM communication for co-resident VMs at layer 1 has some obvious shortcomings due to the need to modify applications. Thus, most of existing shared memory inter-VM communication mechanisms are implemented at either layer 2 or layer 3. However, implementation at layer 2 will result in missing some important TCP/IP features, such as reliability, and some existing third party tool, such as netfilter [Ayuso 2006]. Layer 3 solutions offer high transparency and high reliability but may incur high overhead. Among the collection of shared memory based inter-VM protocols, XenLoop is the most representative in terms of performance, seamless agility, programming transparency and availability of open source release. To better understand the performance of shared memory based inter-VM approaches, we conduct extensive experimental evaluation of XenLoop [Zhang et.al 2013a]. Figure 3 shows the results of running XenLoop for both TCP and UDP workloads by Netperf [Netperf 2015].



(a) Streaming workload (b) Transactional workload

Fig.3. XenLoop Performance measured by NetPerf

It compares the performance of TCP STREAM, UDP STREAM, TCP TRANSACTION, and UDP TRANSACTION workloads that are running on VMs with and without shared memory approach respectively. We make three interesting observations. *First*, for UDP STREAM workload, shared memory based co-located inter-VM communication performances is up to 6 times higher than native co-located inter communication. Also the performance of UDP STREAM workloads increases dramatically when the message size grows above 1KB. *Second*, the performance of transactional workloads is always beneficial using shared memory approach. *Third*, the performance of TCP STREAM workloads running with shared memory approach is always worse than that in native inter-VM communication, irrespective of message size.

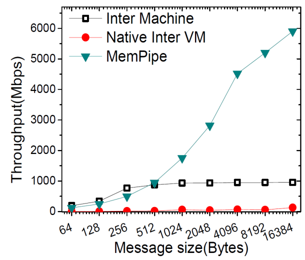
In order to understand the reasons of poor performance of shared memory mechanisms under streaming TCP workloads, we conduct further investigation by using Oprofile [Levon 2014], a low overhead system wide profiler for Linux. We found that the frequent software interrupts incurred in shared memory based inter-VM communications can severely degrade the performance of TCP STREAM workloads between two co-located VMs. We analyze the types and the amount of events occurred in the VM kernel while TCP streaming workloads are running between co-located VMs with MemPipe [Zhang et al. 2015]. Table 1 shows the top three most frequent kernel functions that are invoked during the TCP STREAM workloads. We noticed that the function *__do softirq* is executed in a very high frequency compared with others. In Linux network subsystem, *__do softirq* is an interrupt handler responsible for extracting packets from the socket buffer and delivering them to the applications. The CPU stack switching cost brought by executing a software interrupt handler is non-negligible, especially in the case where the frequency of software interrupt is high.

Table 1. Top Three Most Frequent Invoked Kernel Functions

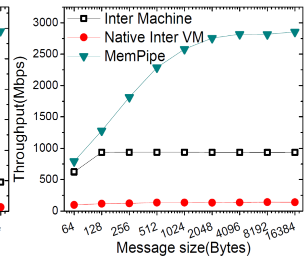
samples	image	function
4684 (%7.16)	**vmlinux**	**__do_softirq**
229 (%0.35)	vmlinux	csum_partial_copy_generic
222 (%0.34)	vmlinux	tcp_ack

These results indicate that reducing the frequency of software interrupts can be a potential opportunity to further improve the performance of shared memory based inter VM communication system, especially for TCP streaming workloads. Interesting to note is that our observation is aligned with the active-tx/rx technologies that are adopted by the NIC drivers to constantly monitor the workload and adaptively tune the hardware interrupt coalescing scheme. This helps to reduce the OS overhead encountered when servicing one interrupt per received frame or per transmitted frame. However, when a shared memory mechanism and its packet interception are implemented at layer 2 (below the systems call layer and above transport layer), the inter-VM network frames are transmitted via shared memory and never go through the hardware or virtual NICs. Thus, the adaptive-tx/rx technologies supported in NIC can no longer help to better manage the overhead of software interrupts in the shared memory based inter-VM communication protocols.

In our MemPipe [Zhang et al. 2015] system we overcome this problem by introducing an *anticipatory time window (ATW)* based notification grouping algorithm to substitute the simple and intuitive *per packet notification issuing (PNI)* algorithm. The idea is that, for sender VM, instead of issuing notification to the receiver VM for each single network packet, it is always anticipating that more packets will arrive in the near future. Thus, we set the anticipation time window t such that the sender can partition notifications into multiple ATW based notification partitions of the same size N, such that each partition batches N packets in a single notification. The reduced number of notifications between the sender VM and the receiver VM can significantly cut down the amount of software interrupts to be handled in both sender and receiver VMs. Proper setting of the parameter N and the ATW interval t is critical for achieving optimal performance. In MemPipe, each streaming partition is formed when N packets have arrived or when the ATW interval t expires. This ensures that the ATW based notification incurs only bounded delay by t and a notification will be issued by the sender at most every t time, even when there are less than N new packets in the shared memory. Figure 4 shows the experimental evaluation of MemPipe, which is a XenLoop like implementation of shared memory inter-VM communication protocol on KVM platform [Zhang et.al 2015].



(a) UDP streaming



(b) TCP streaming

Fig.4. MemPipe throughput performance by running NetPerf

Figure 4 measures the UDP and the TCP streaming throughput by varying the message size from 64B to 16KB using Netperf. Figure 4(a) shows that, for UDP workloads, throughput increases as the message size increases for all 4 scenarios. When message size is larger than 256B, throughput in inter machine and native inter VM scenarios becomes relatively stable, which indicates network communication channel is saturated. In contrast, MemPipe consistently outperform native inter-VM scenario in all message sizes, and MemPipe outperforms the Inter Machine scenario when the message size is larger than 512B and the performance gap increases as the message size increases from 0.5KB to 16KB with up to 32 times higher throughput compared with that in native inter VM case. This is because for small message sizes, the performance is dominated by the per-message system call overhead. However, as the message size increases, the performance becomes dominated by the data transmission. The advantages of using shared memory overweight the overhead caused by per-message system call. Similarly, for TCP streaming workloads, as shown in Figure 4(b), MemPipe reliably outperforms the other two scenarios for all message sizes thanks to the *ATW* based notification grouping algorithm.

In summary, the MemPipe experience shows that the design of a shared memory inter-VM communication protocol needs to take into account the layer in which the shared memory channel will be established and the necessary optimizations due to bypassing the higher layer in the software stack.

3.4 Design Choices for Seamless Agility

Seamless agility is another important design criterion in addition to high performance. By seamless agility, we mean that both the detection of co-resident VMs and the switch between local and remote mode of inter-VM communication should be done automatically and adaptively even in the presence of VM live migration and VM dynamic deployment (e.g., on-demand addition or removal of VMs).

VM live migration is one of the most attractive features provided by virtualization technologies. It provides the ability to transport a VM from one host to another in a short period of time as if the VM's execution has not been interrupted. VM live migration allows VMs to be relocated to different hosts to respond to the load balancing due to varying load or performance, to saving power, recovery from failures, and improving manageability. It is considered by many as a "default" feature of VM technologies.

However, VMs are by design not aware of the existence of one another directly due to the abstraction and isolation support by virtualization. Thus, we need an effective mechanism to identify which VMs are residing on the same host and to determine whether VMs are still on the same physical machine when co-resident VMs are created, migrated in/out, failed or terminated. We argue that automatic co-resident VM detection should be robust and reliable in the presence of live migration and dynamic VM deployment. For example, after VM_1 being detected as co-resident with VM_2, due to load balancing, VM_1 needs to migrate from host A to host B. We say a shared memory channel is reliable and robust if it can ensure that the shared memory channel between these two VMs is only established when they are co-located on the same host. As soon as VM_1 is migrated to another host, the communication between VM_1 and VM_2 will be switched back to remote TCP/IP protocol.

Therefore, seamless agility calls for the support of on-demand co-resident VM detection and automatic switch between local mode and remote mode to retain the benefits of VM live migration and the flexibility for VM deployment. In the remaining of this section, we discuss concrete techniques for implementing auto-detection and auto-switching capabilities.

3.4.1. Automatic Detection of Co-Resident VMs. Two different methods can be used to detect co-resident VMs and maintain VM co-residency information. The simplest one is static, which collects the membership of co-resident VMs during the system configuration time prior to runtime. Such collection is primarily done manually by the administrator, and is assumed unchanged during runtime. Thus, user applications are aware of co-residency information of communicating VMs by static VM detection method. The most commonly used co-resident VM detection method is dynamic, which provides automatic detection mechanisms. In contrast to static method that fails to detect the arrival of new VMs or the departure of existing VMs without administrator's intervention, dynamic co-resident VM detection can be done either periodically and asynchronously or as a tightly integrated synchronous process with the live migration and VM dynamic deployment subsystem:

— The privileged domain or corresponding self-defined software entity periodically gathers co-residency information and transmits it to the VMs on the same host.

— VM peers advertise their presence/absence to all other VMs on the same host upon significant events, such as VM creation, VM shutdown, VM live migration in/out, and so forth.

The first approach is asynchronous and needs centralized management by host domain. It is relatively easier to implement since co-residency information is scattered in a top-down fashion and the information is supposed to be sent to VMs residing on the same host consistently. However, the frequency or time period between two periodical probing operations needs to be configured properly. If the period is set longer than needed, the delay may bring inaccuracy to the co-residency information, leading to possible race conditions. For instance, if VM_1 migrated from host A to another host B is still communicating with VM_2 on host A via shared memory channel, then it may lead to system errors. However, if the time period is set to be too short, it might lead to unnecessary probing and CPU cost.

The second approach is event-driven and synchronous. When a VM migrates out/in, related VMs are notified and the co-residency information is updated as an integral part of the live migration transaction. Thus, the co-residency information is kept fresh and updated immediately upon the occurrence of the corresponding events. When the list of co-resident VMs on a physical machine is updated, it is protected from read. Thus, the consistency of the VM co-residency information is maintained.

3.4.2. Transparent Switch between Local Mode and Remote Mode. Two tasks are involved in performing the transparent switch between local and remote mode:

— To identify if the communicating VMs are residing on the same physical machine;

— To automatically determine the right spot of where and when to perform the transparent switch.

For the first task, the unique identity of every VM and the co-resident information are needed. *<Dom ID, IP/Mac address>* pairs can be used to uniquely identify domains. Whether it is IP address or Mac address, it depends on in which layer the automatic switch feature is implemented. Maintaining co-resident VMs within one list makes the identification easier. The co-resident membership information is dynamically updated by the automatic detection mechanism for co-resident VMs. For the second task, one of the good spots for determining the local or remote switching is to intercept the requests before setting up or tearing down connections or before the sender VM transfers the data. For example, we can make the checking of whether the co-resident VM list is updated and the setting up a connection or the transferring data between two VMs as one single atomic transaction to ensure the correctness.

3.5 Multilevel Transparency

The next design objective is the maintenance of multilevel transparency for developing efficient and scalable inter-VM communication mechanisms. We argue that three levels of transparency are desirable for effective inter-VM communication optimization using shared memory approaches.

3.5.1. User-Level Transparency. User-level transparency refers to a key design choice regarding whether applications can take advantages of the co-resident inter-VM communication mechanism without any modifications to the existing applications and user libraries. With user-level transparency, legacy network applications using standard TCP/IP interfaces do not need to be modified in order to use the shared memory based communication channel. To achieve this level of transparency, shared memory based inter-VM communication is supposed to be implemented in a layer lower than the system calls and user libraries such that there is no modification to layer 1 and thus applications.

3.5.2. OS Kernel Transparency. By OS kernel transparency, we mean that incorporating shared memory channel requires no modification to either host OS kernel or guest OS kernel, thus no kernel recompilation and re-linking are needed. No customized OS kernel and kernel patches need to be introduced, which indicates a more general and ready to deploy solution. To obtain the feature of OS kernel transparency, one feasible approach is to use non-intrusive and self-contained kernel modules. Kernel modules are compiled separately from OS kernel, so recompiling and re-linking the kernel can be avoided. Moreover, they can be loaded at runtime without system reboot. A typical kernel module based approach is to utilize standard virtual device development framework to implement the guest kernel optimization as standard and clean kernel driver module, which is flexible to be loaded/unloaded. In fact, most of the state of art hypervisors support emulated devices (network, block, etc.) to provide functionalities to guest OSes.

3.5.3. VMM Transparency. With VMM transparency, no modification to VMM is required to incorporate shared memory communication mechanisms, maintaining the independence between VMM and guest OS instances. It is well known that modifying VMM is hard and can be error prone, thus it is highly desirable to use and preserve the existing interfaces exported by the VMM instead of modifying them.

The design choice of multilevel transparency closely interacts with other design choices, such as implementation layers, seamless agility, degree of development difficulty, and performance improvement. For example, the benefits of layer 1 implementation, such as less kernel work, fewer data copies and lower overhead of context switches across the boundaries, are obtained at the cost of sacrificing user-level transparency. We will detail some of these interactions in Section 4 when we compare the architectural design of existing representative systems.

3.6 Other Desired Features

In addition to high performance, implementation layer in software stack, seamless agility and multilevel transparency, we argue that full consideration on how to provide reliability, security and stability are also important for the next generation co-residency aware inter-VM communication mechanisms. However, most of the related work to date pays little attentions to these advanced features.

3.6.1. Reliability. By reliability, we mean that a shared memory based implementation of residency-aware inter-VM communication should be error free and fault tolerant. Concretely, the incorporation of a shared memory channel into the existing virtualization platform should not introduce additional errors and should be able to handle exceptions introduced smoothly and automatically. We below discuss two example exception handlings: connection failures and race conditions upon VM migration or dynamic VM deployment.

Connection failures. Connection failures may occur in a virtualization system for both local connections (connection between VMs on the same host) and remote connections (connection between VMs on separate hosts). For local connection failures, the shared memory buffer mapped by the communicating VMs should be deallocated by explicitly unmapping those pages. For remote connections, we will resort to the traditional methods for network failures recovery.

Race conditions. Race conditions may occur when the co-resident VM detection is performed dynamically but periodically. For example, when a VM_1 on host A is migrated to host B right after the co-resident VM detection has updated the list of co-resident VMs and before the list of co-resident VMs will be updated in the next time interval, it is possible that VM_1 communicates with VM_2 still via the previously established shared memory channel on host A, which can lead to race condition due to connection failure since VM_1 is no longer present on host A.

One approach to address this type of race conditions is to use the *synchronous update method* instead of asynchronous update method such as a periodic update based on a pre-defined time interval or static update method. The synchronous update method enables the update to the list of co-resident VMs to be trigged synchronously with a live migration transaction, offering strong consistency support. More importantly, the synchronous update method also avoids race conditions. However, the support of synchronous update method requires to modify the live migration module and the dynamic VM deployment module to add the support of synchronization.

Alternative to synchronous update, if the automatic co-resident VM detection mechanism is accompanied by an *asynchronous update method*, which periodically updates the list of co-resident VMs, then the period update method will periodically check the live migration log and the dynamic VM deployment log to see if any existing VMs in the co-resident VM list have been removed or migrated out to other host or if any new VM has been added to or migrated into the current host. If yes, it will trigger an update to the list of co-resident VMs on the host. The problem with the periodic update method is the possibility of race condition during the time interval between two consecutive updates, as illustrated by the above-mentioned example. Thus, any shared memory based inter-VM communication protocol that adopts periodic update method for refreshing the co-resident VM list will need to provide an exception handling method to address the possible occurrence of race conditions. There are two possible approaches to this problem: race condition prevention and race condition resolution.

Race condition prevention. To prevent the occurrence of a race condition under the periodic update method, we need to add a VM-status checking module as an integral part of operations, such as connection establishment. Prior to the communication from one VM to another co-resident VM, this consistency checking module will double check if they remain to be co-located and their shared memory channel is established. This consistency checking is performed by examining XenStore or facilities like the live migration log from the most recent time point of the periodic update. Upon detecting the addition of new VMs or the removal of existing VMs on current host machine, for instance, if it detects that VM_1 has been migrated from host A to host B, then it will trigger three tasks: (i) the switching of the communication channel between VM_1 (on host B) and VM_2 (on host A) to the remote mode using conventional TCP/IP protocol; (ii) the shared memory tear-down procedure; and (iii) the update to the co-resident VM list. The advantage of this solution compared to the synchronous update method is that it is independent of the live migration module, the dynamic VM deployment module, and possibly other relevant modules at the cost of additional delay for every connection establishment.

Race condition resolution. One way to avoid the additional delay introduced in connection establishments to deal with possible race conditions is to handle connection errors using time-out until the next round of the periodic update is performed. We have two cases to consider:

Case 1: adding new VMs. The communication between the new VM and the other co-resident VMs will be using remote mode instead of shared memory mode due to the fact that the co-resident VM list is not yet updated.

Case 2: removal of existing VMs. The communication between the removed VM and the other co-resident VMs will be still using the shared memory mode. Thus, when VM_1 is migrated from host A to host B, its resource allocations on host A have been de-allocated. Thus, the attempt to use the shared memory communication between VM_1 (on host B) and VM_2 (on host A) will result in a failure and eventually timeout. If VM_1 (on host B) is the sender, it will fail until its co-resident VM list on host B is updated by the next update interval. Similarly, if VM_1 (on host B) is the receiver, it will not be able to access the data placed by VM_2 on their previously established shared memory channel on host A, leading to a communication failure for the sender until the co-resident VM list of VM_2 on host A is updated by the next update interval.

In either case, some inter-VM communications are delayed until the periodic update to the co-resident list is performed. This solution is light-weight compared to the other alternatives, and allows us to avoid the additional overhead introduced to every connection (local and remote). One can further study the adaptive setting of the time interval for periodic update method, for example, with smaller interval for frequent live migration and larger interval for infrequent live migration.

3.6.2. Security. By security, we mean that the shared memory based implementation should provide proper protection level for memory sharing between co-resident VMs based on the degree of mutual trust between co-resident VMs. For example, by implementing the shared memory based inter-VM communication mechanism as a kernel module, it manages the access to the globally allocated shared memory by keeping track of the mapping between shared memory regions for each pair of communicating VMs, and performing address boundary check for each access attempt before it grants the access to the shared memory. By utilizing the protection models of host/guest OSes, it assumes that a sender VM has implicit trust to its receiver VM. One way to introduce higher level of security protection is to add additional level of explicit trust among VMs to allow a VM to choose which other VMs it wants to communicate with. One way to establish such explicit trust is based on past transaction history and feedback ratings [Su et al. 2015]. Other possible solutions can be found from the excellent survey by [Pearce et al. 2013]. Also [Gebhardt and Tomlinson 2010] presents some suggestions on creating a proper security model and protection strategies for virtual machine communications.

3.6.3. Performance Stability. A shared memory inter-VM communication protocol should provide stable and consistent performance for different access protocols, such as UDP and TCP, different types of workload characterizations, such as different data sizes, different data request rates and request patterns, such as transactional or streaming workloads. We have shown in Section 3.3.4 (Figure 3) that some existing shared memory approaches deliver good performance improvement on UDP workloads for co-resident inter-VM communications compared to using conventional approaches but small or no performance improvement for larger message sizes and streaming TCP workloads [Zhang et al. 2013a]. We argue that to enable the wide deployment of a shared memory approach to inter-VM communication, we need to ensure the performance stability for shared memory based inter-VM communication mechanisms, no matter the network protocol is TCP or UDP, the workload is transactional or streaming, the size of messages is small or large, the incoming rate of the messages is normal or very high, the number of co-resident VMs is small or large. Furthermore, when the VM live migration and VM dynamic deployment are present, the throughput performance should be stable regardless whether it is before or after the migration.

3.7 Concluding Remarks

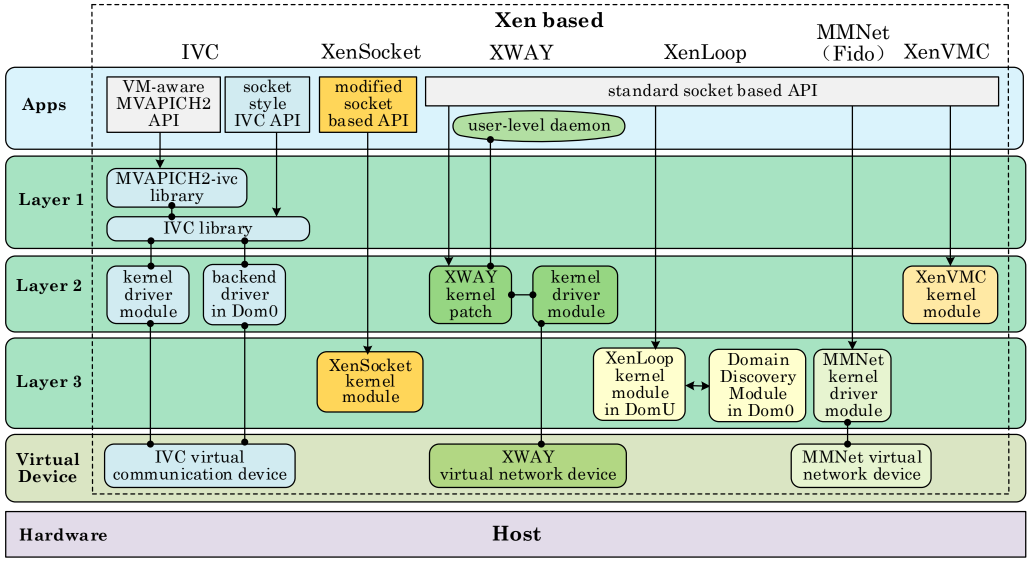
We would like to state that a careful tradeoff should be made regarding different design choices and requirements. For example, the choice of implementation layer is directly related to user-level transparency, and it also has impact on the performance of shared memory approaches. In addition, achieving higher reliability and security often leads to certain loss of performance. Thus, reliability and security facilities should be carefully designed and incorporated to minimize the unnecessary performance overhead. Finally, customizability and configurability are highly desirable and highly recommended features for designing and implementing a scalable and reliable shared memory based co-resident VM communication protocol.

4. ARCHITECTURAL LAYOUT: A COMPARISON

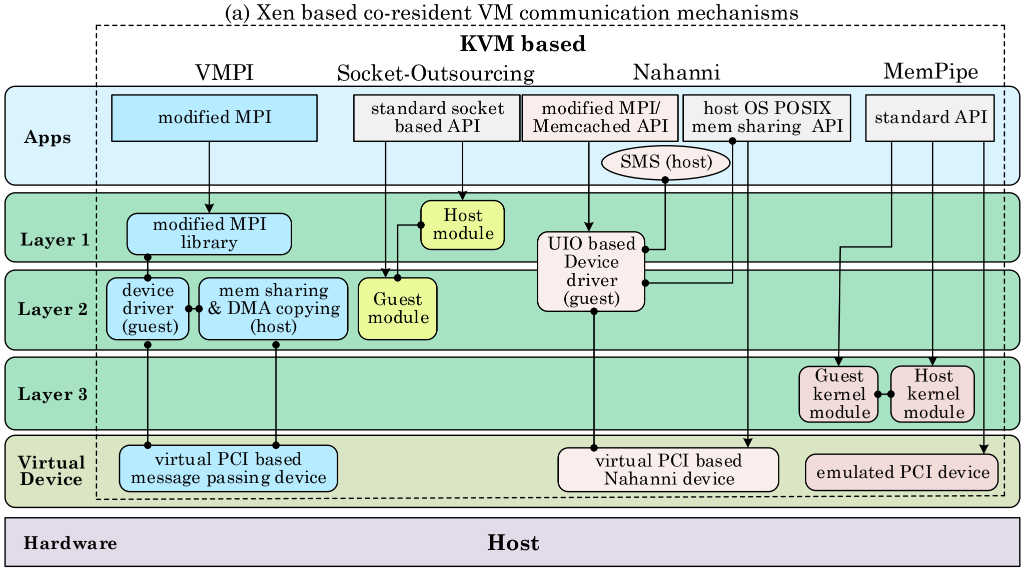
In this section, we classify existing representative shared memory inter-VM communication mechanisms based on their architecture layout in the software stack and provide a comprehensive analysis and comparison on their design choices, including the implementation techniques for fundamental functionalities, the seamless agility support, the multilevel transparency and additional features. In addition we will compare Xen based implementations with KVM based implementations and identify similarity and differences in their design and implementation choices.

Figure 5 shows our classification of existing representative shared memory approaches by their implementation layers in the software stack: Layer 1 is the user libraries and system calls layer, Layer 2 is below system calls layer and above transport layer, and Layer 3 is below IP layer. To facilitate the comparative analysis, we show the Xen based systems and KVM based systems in Figure 5(a) and Figure 5(b) respectively. We illustrate each system with one or several components and show that different components may be designed and implemented at different layers within a single system. For presentation clarity, we defer the discussion on the modifications to VMMs to Section 7.3. We say that an existing shared memory system belongs to the category of layer X (X=1, 2, or 3), when its shared memory based inter-VM communication channels are established in this layer.

For Xen-based systems, we include IVC [Huang et al. 2007], XenSocket [Zhang et al. 2007], XWay [Kim et al. 2008], XenLoop [Wang et al. 2008b], MMNet [Radhakrishnan and Srinivasan 2008] and XenVMC [Ren et al. 2012] in our comparative analysis study. For KVM based system, we include VMPI [Diakhaté *et al.* 2008], Socket-Outsourcing [Eiraku et al. 2009], Nahanni [MacDonell 2011], MemPipe [Zhang et al. 2015] in our comparative analysis study. There are some recent developments in KVM category [Hwang et al. 2014, Zhang et al. 2014b, Zhang et al. 2015]. We have added some discussions on the new developments to date when it is appropriate.



(a) Xen based co-resident VM communication mechanisms



(b) KVM based co-resident VM communication mechanisms

Fig.5. Architectural layout of co-resident VM communication mechanisms

4.1 User Libraries and System Calls Layer

Implementing the shared memory based inter-VM communication in the user libraries and system calls layer has a number of advantages as we have discussed in Section 3. IVC on Xen platform, VMPI and Nahanni on KVM platform are the representative efforts in this layer.

4.1.1. IVC. IVC is one of the earliest shared memory efforts based on Xen hypervisor [Huang et al. 2007]. It is designed for cluster-based HPC environment and is a representative Xen based approach whose user libraries are implemented in layer 1. Different from other related work on Xen platform, IVC is developed based on VMM-bypass I/O model instead of split I/O model.

IVC consists of three parts: a user space VM-aware communication IVC library, a user space MVAPICH2-ivc library and a kernel driver. The IVC library supports shared memory based fast communication between co-resident VMs, which provides socket style interfaces. Supported by IVC, the MVAPICH2-ivc library is developed, which is derived from MVAPICH2, an MPI library over Infiniband. The kernel driver is called by the user space libraries to grant the receiver VM the right to access the sharing buffer allocated by the IVC library and gets the reference handles from Xen hypervisor.

To verify the performance of IVC, [Huang et al. 2007; Huang 2008] conduct evaluations of MVAPICH2-ivc on cluster environment with multi-core systems and PCI-Express InfiniBand Host Channel Adapters (HCAs). InfiniBand is a kind of interconnect offering high bandwidth and low latency through user-level communication and OS-bypass. It can be supported with Xen platform via VMM-bypass I/O [Liu et al. 2006]. High performance and additional features make Infiniband popular in HPC cluster computing environments. Evaluation results demonstrate that in the multi-core systems with Infiniband interconnections, IVC achieves comparable performance with native platforms [Huang et al. 2007; Huang 2008].

4.1.2. VMPI. VMPI [Diakhaté et al. 2008] is an Inter-VM MPI communication mechanism for co-resident VMs targeted to HPC cluster environment on KVM platform. In VMPI, only local channels are supported. Different from other related work, VMPI supports two types of local channels: one to allow fast MPI data transfers between co-resident VMs based on shared buffers accessible directly from guest OSes' user spaces, the other to enable direct data copies through the hypervisor. VMPI provides a virtual device that supports these two types of local channels.

To implement VMPI, both guest and host OS and the hypervisor are extended and modified. In guest implementation, the device is developed as a PCI (Peripheral Component Interface) device for each guest OS based on Virtio framework. It offers shared memory message passing API similar but not compatible to MPI via the device driver to guest OSes. The modified user libraries and the device enable applications in guest OSes to use shared memory instead of TCP/IP network to communicate with each other. *mmap()* is used to map the shared memory of the device to user space. *ioctl* are used to issue DMA requests. In host implementation, basic functionalities, such as memory allocation and sharing, DMA copies are provided. Slight modifications are made to QEMU instances to allow them to allocate memory from a shared memory pool.

Experimental results show that VMPI achieves near native performance in terms of MPI latency and bandwidth [Diakhaté et al. 2008]. Currently, VMPI only supports a small subset of MPI API. And its scalability is limited since it does not support a varying number of co-resident VMs to communicate by using *fork()* to create the QEMU instances.

4.1.3. Nahanni. Nahanni [Macdonell 2011] provides inter co-resident VM shared memory API and commands for both host-to-guest and guest-to-guest communication on KVM platform. In order to avoid the overhead of crossing protection barriers from guest user-level to guest kernel and from guest OS to host, it is designed and implemented mainly in layer 1. Nahanni's interfaces are visible to user space applications. MPI-Nahanni user-level library is implemented for computational science applications. Memcached client and server are modified and extended to benefit from Nahanni [Gordon 2011a; Gordon et al. 2011b]. It supports only local channels. Both stream data and structured data are supported to aim at broader range of applications.

Nahanni consists of three components: a POSIX shared memory region on the host OS, a modified QEMU that supports a new Nahanni PCI device named *ivshmem*, and a Nahanni guest kernel driver developed based on UIO (Userspace I/O) device driver model. The shared memory region is allocated by host POSIX operations. It is mapped to QEMU process address space via *mmap()* and is added to *RAMblocks* structure in QEMU, which makes it possible to manage virtual device memory through available interfaces. After the driver of device *ivshmem* is loaded, the mapped memory can be used by guest applications through mapping it to guest user space via *mmap()* operation. Shared-Memory Server (SMS), a standalone host process running outside of QEMU is designed and implemented to enable inter-VM notification.

Evaluation results show that applications or benchmarks powered by Nahanni achieve better performance [Macdonell 2011; Gordon 2011a; Gordon et al. 2011b; Ke 2011]. However, to take advantage of Nahanni, it is required to rewrite applications and libraries, or to modify and extend existing applications and libraries. In addition, Nahanni by design does not consider VM live migration. Thus it is supposed to be used by applications that do not expect to migrate or to make switches between local and remote mode.

4.2 Below System Calls Layer, above Transport Layer

Implementing shared memory based inter-VM communication below system calls layer and above transport layer represents the layer 2 solution. As we discussed in Section 3, although the implementation of shared memory channel in layer 1 can potentially lead to lower latency and higher throughput of network I/O workloads, it requires modification of user-level applications and libraries and thus has the worst user-level transparency. This motivates the research efforts to explore the implementation of shared memory mechanisms at lower layer. XWAY, XenVMC and Socket-outsourcing are the representative developments to date in layer 2.

4.2.1. XWAY. XWAY is another inter-VM communication optimization for co-resident VMs [Kim et al. 2008]. XWAY is designed based on the belief that it is not practical to rewrite legacy applications with new APIs even if implementation at lower layer of software stack indicates some performance loss. The design of XWAY makes efforts to abstract all socket options and keeps user-level transparency. XWAY modifies the OS kernel by patching it. It intercepts TCP socket calls below system calls layer and above transport layer.

Hierarchically, XWAY consists of three components: switch, protocol and device driver. They are implemented as a few lines of kernel patch and a loadable kernel module. Upon the very first packet delivery attempt, the switch component is used to intercepts socket-related calls between the INET and TCP layer. It determines if the receiver is co-resident VM or not. Then it transparently chooses between TCP socket and local XWAY protocol which should be called whenever a message is transmitted. The protocol component conducts the tasks of data transmission via the device driver. The device driver plays the basic role to support XWAY socket and XWAY protocol. It writes data into the sharing buffer or reads data from it. It also transfers events between the sender and the receiver, and makes callback to upper components when necessary. In the implementation of XWAY, the virtual device is represented as XWAY channels.

Evaluation results show that under various workloads XWAY achieves better performance than native TCP socket by bypassing the long TCP/IP network stack and providing direct shared memory based channel for co-resident VMs [Kim et al. 2008].

4.2.2. XenVMC. XenVMC is another residency-aware inter-VM communication protocol implemented at layer 2, with transparent VM live migration and dynamic VM deployment support [Ren et al. 2012]. It satisfies the three design criteria: high performance, seamless agility and multilevel transparency. For XenVMC, each guest OS hosts a non-intrusive self-contained XenVMC kernel module, which is inserted as a thin layer below system calls layer and above the transport layer.

XenVMC kernel module contains six sub modules: Connection Manager, Data Transfer Manager, Event Manager, System Call Analyzer, VM State Publisher, and Live Migration Assistant. Connection Manager is responsible for establishing or tearing down shared memory based connections between two VM peers. Data Transfer Manager is responsible for data sending and receiving. Event Manager handles data transmission related notifications between the sender and the receiver. System Call Analyzer enables transparent system call interception. It intercepts related system calls and analyzes them. If co-resident VMs are identified, it bypasses traditional TCP/IP paths. VM State Publisher is responsible for announcement of VM co-residency membership modification to related guest OSes. Live Migration Assistant supports transparent switch between local and remote mode together with other sub modules.

Experimental evaluation shows that compared with virtualized TCP/IP method, XenVMC improves co-resident VM communication throughput by up to a factor of 9 and reduces corresponding latency by up to a factor of 6 [Ren et al. 2012].

4.2.3. Socket-outsourcing. Socket-outsourcing [Eiraku et al. 2009] enables the shared memory inter-VM communication between co-resident VMs by bypassing the network protocol stack in guest OSes. Socket-outsourcing is implemented in layer 2. It supports two representative operating systems (Linux and NetBSD), with two separate versions for two VMMs (Linux KVM and PansyVM). We focus our discussion on its Linux KVM version.

Socket-outsourcing consists of three parts: a socket layer guest module, the VMM extension and a user-level host module. In guest OS, a high level functionality module in socket layer is replaced to implement the guest module. Concretely, socket functions in structure *proto_ops* for TCP and UDP are replaced with self-defined ones so that it can bypass the protocol stack in guest OSes by calling the high level host module implemented in host OS and improve the performance of co-resident inter-VM communication. The outsourcing of the socket layer is called Socket-outsourcing. Socket-outsourcing supports standard socket API. It is user transparent. However, the VMM is extended to provide: (i) shared memory region between co-resident VMs, (ii) event queues for asynchronous notification between host module and guest module, and (iii) VM Remote Procedure Call (VRPC). The user-level host module acts as a VRPC server for the guest module. It provides socket-like interfaces between the guest module and the host module.

Experimental results show that by using Socket-outsourcing a guest OS achieves similar network throughput as a native OS using up to four Gigabit Ethernet links [Koh 2010]. From the results of an N-tier Web benchmark with significant amount of inter-VM communication, the performance is improved by up to 45% than conventional KVM hosted VM approach [Eiraku et al. 2009]. VM live migration is not a part of the design choices and thus not supported by Socket-outsourcing.

4.3 Below IP Layer

Implementing the shared memory based inter-VM communication below the IP layer has several advantages, such as higher transparency, as outlined in Section 3. However, layer 3 is lower in the software stack and implementation at layer 3 may potentially lead to higher latency due to higher protocol processing overheads and higher number of data copies and context switches across barriers. XenSocket, XenLoop and MMNet are the representative efforts developed in layer 3 on Xen platform. MemPipe [Zhang et al. 2015] is a recent implementation on KVM platform in layer 3. All these efforts are focused on optimization techniques for high performance.

4.3.1. XenSocket. XenSocket [Zhang et al. 2007] provides a shared memory based one way co-resident channel between two VMs and bypasses the TCP/IP network protocol stack when the communication is local. It is designed for applications in large scale distributed stream processing system. Most of its code is in layer 3 and is compiled into a kernel module. It is not binary compatible with existing applications. And it makes no modification to either OS kernel or VMM.

In XenSocket, there are two types of memory pages shared by the communicating VM peers: a descriptor page and buffer pages. The descriptor page is used for control information storage. While the buffer pages are used for data transmission. They work together to form a circular buffer. To establish a connection, the shared memory for circular buffer is allocated by the receiver VM and later mapped by the sender VM. Then the sender writes data into the FIFO buffer and the receiver reads data from it in a blocking mode. The connection is torn down from the sender's side after data transfer to ensure that the shared resources are released properly. To enhance the security, application components with different trust levels are placed on separate VMs or physical machines.

Performance evaluation shows that XenSocket achieves better bandwidth than TCP/IP network [Zhang et al. 2007]. Without the support of automatic co-resident VM detection, XenSocket is primarily used by applications that are aware of the co-residency information and do not expect to migrate or make switches between local and remote mode during runtime. Once XenSocket is used, the remote path will be bypassed.

4.3.2. XenLoop. XenLoop [Wang et al. 2008] provides fast inter-VM shared memory channels for co-resident VMs based on Xen memory sharing facilities to conduct direct network traffic with less intervention of the privileged domain, compared with that of traditional TCP/IP network communication. It provides multilevel transparency and needs no modification, recompilation, or re-linking to existing applications, guest OS kernel and Xen hypervisor. To utilize netfilter [Ayuso 2006], a third party hook mechanism, XenLoop is implemented below IP layer, the same layer as netfilter resides.

XenLoop consists of two parts: (i) a kernel module named XenLoop module which is loaded between the network layer and link layer into each guest OS that want to benefit from the fast local channel, and (ii) a domain discovery module in Dom0. Implemented on top of netfilter, the module in guest OS intercepts outgoing network packets below the IP layer and automatically switches between the standard network path and a high speed inter-VM shared memory channel. *<guest-ID, MAC address>* pairs are used to identify every VM. One of the kernel parts of XenLoop module is its fast bidirectional inter-VM channel. The channel structure is similar to that of XWAY. The channel consists of two FIFO data channels (one for data sending, the other for data receiving) and a bidirectional event channel that is used to enable notifications of data presence for the communicating VM peers. The module in Dom0 is responsible to discover co-resident VMs dynamically and maintain the co-residency information, with the help of XenStore. XenLoop supports transparent VM live migration via the above modules.

Evaluations demonstrate that XenLoop increases the bandwidth by up to a factor of 6 and reduces the inter-VM round trip latency by up to a factor of 5, compared with frontend-backend mode [Wang et al. 2008b]. Although XenLoop satisfies the three design criteria we outlined in Section 3, we have shown in Figure 3 (Section 3) some problems inherent in the XenLoop implementation, due to the choice of layer 3, such as bypassing some important optimizations provided at the NIC layer.

4.3.3. MMNet. Different from other related work, MMNet [Radhakrishnan and Srinivasan 2008] works together with Fido framework [Burtsev et al. 2009] to provide shared memory based inter-VM communication optimization for co-resident VMs on Xen platform. Fido was designed for enterprise-class appliances, such as storage systems and network-router systems. It offers three fundamental facilities: a shared memory mapping mechanism, a signaling mechanism for inter-VM synchronization and a connection handling mechanism. Built on top of Fido, MMNet emulates a link layer network device that provides shared memory based mechanisms to improve the performance of co-resident inter-VM communication. Its driver is implemented in lowest layer of the protocol stack. It easily achieves user-level transparency by providing a standard Ethernet interface. Fido and MMNet together give the user a view of standard network device interfaces, while the optimization of shared memory based inter-VM communication is hidden beneath IP layer. MMNet enables switching between local and remote communication mode by updating the IP routing tables.

Different from other Xen based related work, Fido leverages the relaxed trust model to enable zero copy and to reduce data transfer overhead. It maps entire kernel space of the sender VM to that of the receiver VM in a read-only manner to avoid unnecessary data copies and to ensure the security. Actually, it is designed for communication between VMs that are trustable to each other, where the mapping of guest OSes' memory is acceptable since the possibility of malicious memory access is low in a private, well-controlled environment. As for reliability, Fido provides heart-beat check based connection monitoring and failure handling mechanisms to be capable of detecting VM failures and conducting proper operations accordingly.

MMNet obtains lower performance overhead by incorporating relax trust model [Burtsev et al. 2009; Radhakrishnan and Srinivasan 2008]. Evaluation [Burtsev et al. 2009] shows that for TCP STREAM and UDP STREAM workload, MMNet provides near native performance. For TCP STREAM workload, it achieves twice throughput compared with XenLoop. And for UDP STREAM workload, MMNet increased the throughput by up to a factor of 4 compared with that of Netfront, its latencies are comparable to XenLoop for smaller message sizes. As the message sizes increase ($\geqslant$ 8KB), MMNet outperforms XenLoop by up to a factor of 2.

4.3.4. MemPipe. MemPipe is the most recent shared memory inter-VM communication method implemented in layer 3 on KVM platform [Zhang et al. 2015]. To optimize the performance of layer 3 implementation, MemPipe introduced a dynamic shared memory pipe framework for efficient data transfer between co-located VMs with three interesting features: (i) MemPipe promotes a dynamic proportional memory allocation mechanism to enhance the utility of shared memory channels while improving the co-located inter-VM network communication performance. Instead of statically and equally allocating a shared memory pipe to each pair of co-located communicating VMs, MemPipe slices the shared memory into small chunks and allocates the chunks proportionally to each pair of VMs based on their runtime demands. (ii) MemPipe introduces two optimization techniques: time-window based streaming partitions and socket buffer redirection. The former enhances the performance of inter-VM communication for streaming networking workloads and the latter eliminates the network packet data copy from sender VM's user space to its VM kernel. (iii) MemPipe is implemented as kernel modules to achieve high transparency. Its functionalities are split between a kernel module running in the guest kernel and a kernel module in the host kernel. MemPipe kernel module in the host is responsible for allocating the shared memory region from the host kernel memory and initializing the allocated shared memory region so that guest VMs are able to build their own memory pipes. The MemPipe kernel module in a guest VM manages the shared memory pipes for its communication with other co-located VMs. It consists of peer VM organizer, packet analyzer, memory pipe manager, emulated PCI device, and error handler.

The Peer VM Organizer enables each VM to distinguish its co-located VMs from remote VMs (VMs running on a different host machine). It is implemented using a hashmap, with the Mac address of a VM as the hash key and the corresponding data structure used for establishing the memory pipe as the hash value. Packet Analyzer helps VMs to determine whether a network packet is heading to their co-located VMs or remote VMs. Memory Pipe Manager consists of four parts: Pipe Initiator, Pipe Reader/Writer, Pipe Analyzer, and Pipe Inflator/Deflator. Since KVM does not allow sharing memory between the host and the guest VM, MemPipe creates an emulated PCI device in each VM to overcome this limitation. The PCI device takes the memory, which is allocated and initialized by the Shared Memory Initiator in the host, as its own I/O region. Then it maps its I/O region into the VM's kernel address, and transfers the base virtual address to the Memory Pipe Manager. Thus, the Memory Pipe Manager is able to access the shared memory from this virtual address. Although the based virtual addresses may not be the same in different VMs, they are pointing to the same physical address: the beginning of the memory allocated by the Shared Memory Initiator. After putting a packet into a shared memory pipe, the sender VM notifies the receiver VM through the Events Handler to fetch the packets.

Experimental evaluations show that MemPipe outperforms existing layer 3 systems such as XenLoop (recall Figure 4 in Section 3) for both transactional workloads and streaming workloads under TCP and UDP. The most recent release of MemPipe is built on KVM [Kivity et al. 2007] 3.6, QEMU (http://wiki.gemu.org) 1.2.0, and Linux kernel 3.2.68.

4.4 Comparing Xen Based Implementation vs. KVM Based Implementation

Shared memory based inter-VM communication optimization for co-resident VMs is desirable for both Xen based virtualization environment and KVM based virtualization environment. Early research is primarily targeted for HPC MPI applications. Recent trends are towards more generic computing areas such as large scale distributed computing, Web transaction processing, and cloud computing and big data analytics services. Understanding the similarity and subtle differences between Xen based implementations and KVM based implementation is critical and beneficial for the development and deployment of next generation shared memory based inter-VM communication systems.

In general, Xen based shared memory implementations and KVM based efforts share all the design objectives, functional and non-functional requirements for shared memory based inter-VM communication optimization. For example, they share common fundamental functionalities, such as memory sharing, connection setup and tear-down, local connection handling, data sending and receiving, to name a few. In addition, they also share the non-functional requirements such as high performance, multi-level transparency, seamless agility, reliability and security.

However, Xen and KVM are different virtualization platform with different network architectures and interfaces as outlined in Section 2. These architectural and network interface differences contribute to the differences in the design and implementation of their respective shared memory communication mechanisms. For Xen based shared memory communication development efforts, there are more fundamental mechanisms available, such as Xen Grant Table, Xen Event Channel and XenStore, which provide good foundation to simplify the design and implementation of shared memory channels between co-resident VMs. In comparison, KVM does not offer as many fundamental programmable capabilities as Xen. Thus, KVM based research and development efforts need to focus more on mechanisms to provide a local memory sharing method for guest-guest VM communication and host-guest VM communication. Although *virtqueues* in Virtio framework can be used like I/O ring buffers, Virtio is based on DMA semantics, which is not suitable for every design. Regardless of whether using Virtio framework or not, the QEMU/KVM needs to be extended to provides similar facilities like the Xen Grant Table, Xen Event Channel and XenStore. For example, to extend current hypervisor for providing underlying support of shared memory, Nahanni and VMPI modify QEMU, Socket-outsourcing modifies KVM module and QEMU. As layer 1 implementation, VMPI and Nahanni choose to sacrifice their user-level transparency to simplify the implementation of the shared memory channel and reduce potential context switch overhead. Thus, the shared memory channel in both systems is visible to applications. The legacy applications need to be rewritten or modified to know about the co-residency information of the respective VMs in order to benefit from the shared memory based inter-VM communication mechanisms. As layer 2 implementation, Socket-outsourcing provides the shared memory region between co-resident VMs, event queues for asynchronous notification between host and guest, and VM remote procedure call (VRPC) using user-level host module as a VRPC server for the guest with socket-like interfaces between the guest and the host. Moreover, all three existing KVM based developments provide no support for seamless agility, namely no support for automated switching between local and remote communication channels. MemPipe is the only full-fledged shared memory system on KVM platform, which satisfies the three design criteria discussed in Section 3: high performance, seamless agility and multi-level transparency.

Table II summarizes the architectural design and general features of existing representative shared memory inter-VM communication systems:

Table II. Architectural Layout and General Features

	Xen Based						**KVM Based**			
	IVC	***XenSocket***	***XWAY***	***XenLoop***	***MMNet (Fido)***	***XenVMC***	***VMPI***	***Socket-outsourcing***	***Nahanni***	***MemPipe***
Scope of application	HPC	Distributed processing	Network intensive	Network intensive	Enterprise-class services	Network intensive	HPC	Not specified	HPC, MemCached	Network intensive
Type of API	IVC-specific API, VM-aware MVAPICH2 API	Modified socket based API	Standard API	Standard API	Standard API	Standard API	Modified MPI	Standard socket based API	Modified MPI &Memcached API	Standard API
I/O virtualization model supported	VMM-bypass I/O	Split I/O	Split I/O	Split I/O	Split I/O	Split I/O	Virtio based	Software emulated I/O	Software emulated I/O	Software emulated I/O
Location of local shared memory channel	Layer 1	Layer 3	Layer 2	Layer 3	Layer 3	Layer 2	Layer 1	Layer 2	Layer 1	Layer 3
Form of main components	Libraries, kernel driver	Kernel module	Kernel patch, kernel driver	Kernel modules	Kernel driver	Kernel module	Library, kernel driver, patch for QEMU	Host module, guest module, patch for KVM & PansyVM	UIO based device driver, patch for QEMU/KVM	Host module, Guest module Patch for KVM

4.5 Recent Advances in network I/O virtualization

Recent research on network I/O virtualization has centered on improving the inter-VM network I/O performance by software defined network (SDN) and network function virtualization (NFV). Representative technology includes the single root I/O virtualization (SR-IOV) [MSDN 2014] for making PCI devices interoperable, and Intel Data Plane Development Kit (DPDK) [Sanger 2013; Intel 2013] for fast packet processing using multicore systems.

SR-IOV capable devices allow multiple VMs to independently share a single I/O device and can move data from/to the device by bypassing the VMM. SR-IOV offers an attractive alternative for the virtualization of high performance interconnects such as InfiniBand. [Jose et al. 2013] shows that by combining SR-IOV with InfiniBand, instead of TCP/IP networks, based on VMM-bypass I/O model, it can obtain near to native performance for inter node MPI point to point communication. [Zhang et al. 2014a, Zhang et al. 2014b] show that by introducing VM residency-aware communication, the performance of MPI communication on SR-IOV based InfiniBand clusters can be further improved.

Although SR-IOV improves the communication between VM and its physical device, it cannot remove the overhead of co-located inter-VM communication. This is because with SR-IOV, packets still need to travel through the network stack of the sender VM, to be sent from the VM to the SR-IOV device, and then sent to the receiver VM. This long path can still lead to unnecessarily high cost for co-located inter-VM communication, especially for larger sizes messages.

Alternatively, Intel DPDK [Intel 2013] is a software framework that allows high throughput and low-latency packet processing. It allows the applications to receive data directly from NIC without going through the Linux kernel, and eliminates the overhead of interrupt driven packet processing in traditional OS. As a set of libraries and drivers, Intel DPDK utilizes huge pages in guest VMs and multicore processing to provide applications direct access to the packets on NIC. The huge pages in DPDK are statically allocated to each VM. However, DPDK is restricted and on its own cannot yet support flexible and fast network functions [Sanger 2013].

NetVM [Hwang et al. 2014] is the most recent development by utilizing shared memory mechanism on top of DPDK. NetVM shows that the virtualized edge servers can provide fast packet delivery to VMs bypassing the hypervisor and the physical network interface. However, NetVM is limited to run on a DPDK enabled multicore platform, and no open source is made available to date.

In summary, shared memory based inter-VM communication mechanisms are important value-added methods for further improving the performance of network I/O virtualization.

5. FUNDAMENTAL FUNCTIONAL COMPONENTS : DESIGN COMPARISON

All shared memory based residency-aware inter-VM communication mechanisms must provide three fundamental functionalities: (i) memory sharing structures and corresponding facilities, (ii) shared memory channel (local connection) set up/ tear down, and (iii) sending and receiving data. Figure 6 illustrates the interactions of these three common functional components for enabling inter-VM communication.

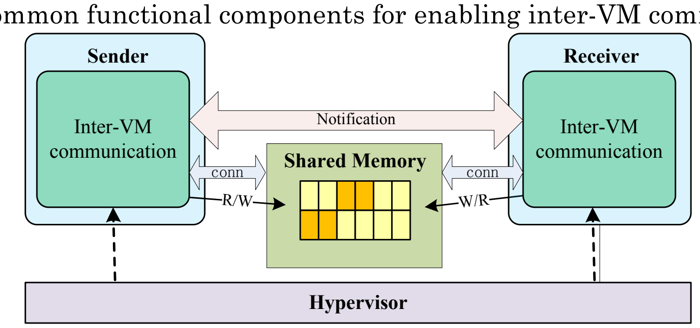


Fig. 6. Fundamental common functionalities of inter-VM communication between co-resident VMs

5.1 Memory Sharing

For VMs to communicate via their shared memory channel, it is necessary to build a buffer of shared physical pages for both VMs such that the two VMs can communicate directly by accessing the specific slot in the buffer.

For existing shared memory systems implemented on Xen platform, Xen Grant Table is usually used as a fundamental memory sharing facility to offer mapping/transfer pages between sender VM and receiver VM. Grant Table provides an efficient and secure way for memory sharing. Grant Table API is accessible from OS kernel, which explains to some extent why most of Xen based shared memory mechanisms are implemented in the kernel. Implementation at kernel-level makes it easier to support both shared memory based local channel and traditional network based remote channel since functionalities, such as packet interception and redirection, can be supported in the kernel between user applications and inter-VM communication enabling components.

However, different shared memory methods implemented on the KVM platform may differ from one another in terms of their support for memory sharing: *virtqueues* in Virtio framework provides the interfaces to manage a list of buffer descriptors, which are used by VMPI as ring buffers to implement its messaging passing MPI. However, *virtqueues* itself does not provide the semantics of guest-guest or host-guest memory sharing. Nahanni shows that instead of using Virtio framework it is preferable to expose *ivshmem*, the virtual device, as a PCI device to the guest OSes. Nahanni's shared memory is created by the host OS with POSIX API and is mmapped to the shared memory region of the virtual PCI device, which is exposed by the device driver to guest OS and is mmapped again to the user-level. For KVM, user-level QEMU is capable of accessing guest physical memory. By taking advantage of this feature, Socket-outsourcing implements its self-defined memory sharing algorithms.

5.2 Local Connection Handling

When the sender VM or the receiver VM detects the first network traffic to its co-resident communicating VM, it initiates the procedures to establish shared memory based local connection. The local connections are set up usually as a client-server model between the sender and the receiver. The simplified procedures are: first, memory for shared buffer is allocated by the client who sponsors the connection establishment procedure, and then the client passes the handles of the buffer to the server, which maps the communication buffer to its own address space through inter-VM shared memory facilities. The event channel or similar structures are initialized for control flow between communicating VMs.

After the data transfer is finished, the sender or the receiver is shut down, or when the connection is to be switched from local to remote, the local connection needs to be torn down. Different from traditional network connection, where the sender and the receiver can be shut down independently, specific care should be given to correctly tear down the shared memory based local channel to make sure that the shared memory is unmapped properly and consequently, the memory is de-allocated.

5.3 Data Transmission

Once the shared memory channel is established, the local connection is initialized, the send VM can send the network traffic to the receiver VM via the local channels. There are two popular ways to organize the shared buffer for data transferring: circular buffer based method and non-circular buffer based method.

5.3.1. Circular Buffer based Approaches. The circular buffer based approaches organize the shared buffer into two producer-consumer circular buffers (one for sending and one for receiving) with one event channel for every pair of communicating VMs. The producer writes data into the buffer and the consumer reads data from it in an asynchronous manner. The event channel is used to notify the communicating VMs of the presence of data in the circular buffer. The offset in the buffer should be well maintained. Circular buffer based mechanisms offer the advantage that data in the buffer is kept in order and thus no explicit cross-domain synchronization is needed.

XenSocket, XWAY, XenLoop and XenVMC are representative shared memory based methods on Xen platform, which incorporate the circular buffer mechanism. IVC and VMPI are layer 1 shared memory implementations on KVM. They share the idea with above related work but with small differences. For IVC channel, the shared buffer consists of two circular buffers containing multiple data segments and a pair of producer/consumer pointers. Instead of using Xen Event Channel based notification mechanism, both the sender and the receiver check the pointers to determine if the buffer is full or if data has arrived. VMPI provides two types of local channels: a producer/consumer circular buffer for small messages ($\leqslant$32KB) and a rendezvous protocol based DMA channel for larger messages. Different from other existing work, XenSocket channel is one way and does not support bi-directional data transmission.

5.3.2. Non-Circular Buffer based Approaches. There are a number of existing shared memory mechanisms, such as MMNet on Xen platform, Nahanni and Socket-outsourcing on KVM platform, which organize their buffer for data sharing differently from the circular buffer based approaches. MMNet is built on top of Fido, which adopts a relaxed trust model and maps the entire address space of the sender VM into the receiver VM's space before any data transmission is initiated, so that the data is transferred with zero copy. For Socket-outsourcing, the guest module allows the host module to access its memory regions, which makes memory sharing between host and guest possible. Event queues are allocated in the shared memory and can be accessed via self-defined API to enable asynchronous communication between the host module and the guest module. Socket like VRPC protocol is implemented between the sender and the receiver to set up or tear down connections, to transfer data, and so forth. For Nahanni, it modifies QEMU to support a virtual PCI device named *ivshmem*. The shared memory object allocated by host POSIX operations is mapped to the shared buffer of this device. The UIO device driver in guest OS makes the shared buffer available to the guest user-level applications. Its SMS coordinates with Linux *eventfds* and *ivshmem's* register memory mechanisms to provide inter-VM notification mechanism for Nahanni. Different from other related work, Nahanni supports not only stream data but also structured data.

Table III summarizes the features of fundamental functionalities for both Xen based and KVM based shared memory systems

Table III. Features of Fundamental Functionalities

	Xen Based						**KVM Based**			
	IVC	***XenSocket***	***XWAY***	***XenLoop***	***MMNet (Fido)***	***XenVMC***	***VMPI***	***Socket-outsourcing***	***Nahanni***	***MemPipe***
Approach to bypassing traditional network path	Via VM-aware MVAPICH 2-ivc library	Via modified API and user libraries	Socket-related calls interception	Netfilter hooks	IP routing tables updating	Transparent system call interception	Via modified MPI and user libraries	Replacing related kernel functions with self-defined ones	Via modified API and device ivshmem	Replacing / Extend kernel functions with self-defined ones
Auxiliary facilities for local connection and data exchange	Grant Table Event Channel	Grant Table Event Channel	Grant Table Event Channel	Grant Table XenStore Event Channel	Grant Table XenStore Event Channel	Grant Table XenStore Event Channel	Virtio	Shared memory, event queues, VRPC of extended KVM	PCI, mmap, UIO, eventfds	Shared memory, PCI, event queues,
Complete memory isolation	Yes	Yes	Yes	Yes	No Relaxed	Yes	Yes	Yes	Yes	Yes
Bi-directional connection	Yes	No One way	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Data buffer type	Circular buffer	Circular buffer	Circular buffer	Circular buffer	Address space mapping	Circular buffer	Circular buffer for message (≤32KB)	N/A	Host: POSIX memory object Guest: device memory region	Circular buffer

6. SEAMLESS AGILITY : COMPARATIVE ANALYSIS

Recall Section 3.4, seamless agility refers to the support for (i) automatic detection of co-resident VMs, (ii) automatic switch between local shared memory mode of communication and remote TCP/IP communication, and (iii) dynamic addition or removal of co-resident VMs. These functionalities are critical for shared memory based inter-VM communication to work effectively and correctly in the presence of VM live migration and VM dynamic deployment. We dedicate this section to provide a comparative analysis on whether and how seamless agility is supported in existing representative shared memory based co-resident VM communication mechanisms.

6.1 Automatic Detection of Co-resident VMs

As mentioned in Section 3.4, there are two approaches to maintain VM co-residency information: static method and dynamic method.

For Xen based shared memory systems, XenSocket does not provide any support for dynamic addition or removal of co-resident VMs, and thus cannot work in conjunction with the VM live migration. XWAY [Kim et al. 2008] utilizes a static method to generate and maintain its co-resident VM list. When a sender VM sends a request to a receiver VM, XWAY refers to the static file that lists all co-resident VMs to determine whether the receiver VM resides on the same physical machine or not. If yes, XWAY performs automatic switch between shared memory based local channel and traditional network path. However, XWAY does not support dynamic addition or removal of co-resident VMs. Thus, XWAY only works when VM live migration and VM dynamic deployment are performed manually.

IVC [Huang et al. 2007] initially registers the VM co-residency information in the backend driver of Dom0 in a static manner. During runtime, IVC provides API for communicating client (sender) and server (receiver) to be registered to form a local communication group and to obtain the group membership information such as *magic id*. All registered domains with the same *magic id* will communication through local IVC channels. Although IVC's original co-resident VM information is collected in a static manner, it supports automatic detection of co-resident VMs by dynamically maintaining IVC-active list, a list of active virtual connections between co-resident VMs.

XenLoop [Wang et al. 2008] provides a soft-state domain detection mechanism to monitor the addition/removal of co-resident VMs and to dynamically gather and update VM co-residency information. A Domain Discovery module in Dom0 periodically scans all guest entries in XenStore, where each entry represents its state. Then the module advertises the updated information to all the VMs covered by the existed entries and the VMs update their local *<guest-ID, Mac address>* lists.

MMNet does not require a coordinator in Dom0. Each member VM writes to the XenStore directory to advertise its presence and watches for membership updates. When corresponding events occur, the IP routing tables are updated accordingly. MMNet [Radhakrishnan and Srinivasan 2008] claims that it provides automatic VM detection mechanism and supports VM live migration, but related technical details are not made available.

XenVMC differs from MMNet in that there is no centralized management. Co-resident VMs and their related local channels information are organized in a structure defined as *vms[]*, which is an array of co-resident VMs maintained by every VM. Each item in *vms[]* represents one of the co-resident VMs on the physical host machine (excluding the Dom0 VM itself). It stores not only co-residency information, but also the data of local channels. When one VM is created or migrated into a new host, *vms[]* of this VM will be initialized. When an event such as creation, migration in/out or shutdown of a co-resident VM occurs, all the other VMs on the same physical machine will be notified so that their *vm[]* will be correctly updated. The *vms[]* entry of a VM will be deleted, upon receiving the request to migrate out, from its current physical host machine.

For KVM based shared memory systems, VMPI [Diakhate et al. 2008], Nahanni [Macdonell 2011] and Socketoutsoucing [Eiraku et al. 2009] all provide no support for transparent detection of co-resident VMs. Nahanni's shared memory server (SMS) is originally designed and implemented to enable inter-VM notification. The guest OS is required to be registered with SMS when it is first connected to SMS. However, automatic co-resident VM detection and auto-switch between local channel and remote network path are not supported in Nahanni, In order to support dynamic co-resident VM membership maintenance, Nahanni needs to be extended. For example, to facilitate Memcached client and server to use Nahanni, Adam Wolfe Gordon implemented Locality Discovery, a user-level mechanism, to judge if the client and the server application are in two co-resident VMs [Gordon 2011a; Gordon et al. 2011b]. MemPipe [Zhang et al. 2015] extends QEMU/KVM to provide full-fledged support for seamless agility for shared memory communication on KVM platform. Similarly, for Xen based platform, XenLoop and XenVMC support fully transparent VM migration, transparent switch between local and remote mode, as well as automatic co-resident VM detection.

6.2 Auto-Switch between Local Mode and Remote Mode of Inter-VM Communication

To support transparent switch between local mode and remote mode, the following two tasks should be performed: (i) automatically identify whether the communicating VMs are co-resident or not and (ii) able to intercept outgoing request/data and redirect the control to the corresponding mode of communication.

For Xen based shared memory solutions, IVC has two kinds of communication channels available in MVAPICH2-ivc: a shared memory based local IVC channel over user space shared memory for co-resident communicating VMs and a network based remote channel over InfiniBand for communicating VMs on separate hosts. As VMs migrates in/out, MVAPICH2-ivc library supports intelligent switch between the local IVC channel and remote network channel via its communication coordinator implemented in user space libraries, which is responsible to dynamically set up or tear down the connections when VM migration occurs. It also keeps IVC-active list and updates it when necessary. IVC offers the flexibility of supporting VM live migration. However, since information such as *magic id* must be preconfigured, live migration is not supported with fully transparency.

XWAY determines whether the receiver VM is co-resident or not by referring to a preconfigured static file through its switch component at the first packet delivery. Then it decides to choose between TCP socket and local XWAY protocol to transmit the data.

XenVMC transparently intercepts every network related system call and executes its user defined system call handler. The handler analyzes from the context whether the communicating VMs are co-resident or not. If yes, it bypasses the TCP/IP path by executing the codes for shared memory based communication. Otherwise, it recovers entry address of original handler and executes it. The entries in system call table are modified in a way that is transparent to the OS kernel.

MMNet supports automatic co-resident VM membership maintenance, which enables communicating VMs to establish connections dynamically. MMNet also allows running applications within a VM to seamlessly switch from/to the local shared memory path by updating the IP routing tables accordingly.

Instead of implementing the interception mechanism from scratch, XenLoop uses netfilter [Ayuso 2006], a third party hook mechanism provided inside the Linux kernel for kernel modules to register callback functions within the network stack, to intercept every outgoing network packet below IP layer and to determine the next hop node by the information in its header. Since netfilter resides below IP layer, its interception mechanism indicates longer data transmission path compared with alternative approaches in layer 2.

For KVM based shared memory communication methods, such as Nahanni and VMPI, the user applications need to explicitly specify whether local optimized channel or original network channel is to be used via different API: specific shared memory API or standard network API. Nahanni does not support the transparent switch between local and remote mode. Socket-outsourcing replaces traditional socket communication functions with shared memory oriented ones at the socket layer, without modifying standard socket API. However, no technical details are provided to indicate that it supports the transparent switch between local and remote mode. MemPipe [Zhang et al. 2015] supports the auto-switch between local shared memory communication and remote inter-VM communication through conventional network channel as a part of its kernel module implementation.

6.3 Discussion on Race Condition

Seamless agility not only refers to auto-detection of co-resident VMs and auto-switch between local and remote communication channels, but also requires the guarantee that residency-aware inter-VM communication is reliable and robust in the presence of dynamic addition or removal of co-resident VMs. Ideally, dynamic addition or removal of a VM should be made visible to other VMs on the same physical host machine immediately after the event occurs. However, when VMs on the same host cannot be notified of the membership changes immediately and synchronously upon the update to the VM co-residency information, race condition may occur. Thus, not only the detection of VM co-residency information should be done on demand and whenever a VM communicates with another VM. We also need to provide immediate update mechanisms for refreshing the list of co-resident VMs on the respective physical host machine(s). For example, when a VM is migrated out to another physical host machine, it should notify all its co-resident VMs on both the current host machine and the new destination host machine to update their co-resident VM list. This will avoid errors concerning local/remote connection management and pending data handling due to the fact that the co-resident VM list is out of date.

Recall our discussion on reliability in Section 3.6.1, we have described what race conditions are and how they may occur for shared memory inter-VM communication in the presence of dynamic VM migration and dynamic VM deployment. We have also briefly described some race condition prevention or resolution mechanisms. In general, the following three categories of operations need co-residency information either explicitly or implicitly and thus may become vulnerable when the co-residency information is not kept up to date: (i) *connection establishment*, (ii) *pending data handling* for data remained from former local/remote connections, and (iii) *connection tearing down*. Let t_{op} denote the beginning time for an operation, t_{event} denote the time when the addition or removal of a VM occurs, and $t_{notified}$ denote the time when corresponding co-resident VMs are notified of the events. If the gap between t_{event} and $t_{notified}$ is large and t_{op} happens to fall into the time interval specified by the gap, then a race condition is created due to the inconsistency between the up-to-date co-resident VM information and the out of date co-resident VM list maintained in the VMs of the respective host machines. In the rest of this section, we will provide a comparative analysis on how existing shared memory systems handle such race conditions.

6.3.1 Race Condition Handling in Existing Shared Memory Systems.

For residency aware inter-VM communication mechanisms that adopt static methods for detection of co-resident VMs, such as IVC and XWAY, no race condition (as defined in Section 3.6.1) exists. The reason is that once the co-residency information is prefigured, no membership changing is allowed during runtime, unless the static file with membership information is modified and the co-resident VM list is updated via specific API manually.

For implementations with dynamic co-resident VM detection, race conditions may occur if the update to the co-resident VM list is asynchronous and deferred with respect to the on-demand addition or removal of VMs. For example, for several Xen based shared memory systems, such as XenLoop, MMNet and XenVMC, which support dynamic co-resident VM detection via XenStore, race conditions exist. However, neither of these systems has discussed how they handle race conditions in the presence of VM live migration.

In order to support race condition prevention, XenStore based module needs to be extended to enable synchronous notification to be triggered before any update transaction over the XenStore to update the co-residency information of a VM is committed. Compared to the prevention methods that require modifying XenStore based module to maintain strong consistency, the race condition resolution methods use the optimistic approaches to handle or compensate the possible errors after the occurrence of race condition. They maintain weak consistency in the sense that the notification of changes to the co-residency information caused by VM live migration is delayed until the respective VMs initiate or be involved in an inter-VM communication request.

For KVM based shared memory systems, only MemPipe provides auto-detection of co-resident VMs and auto-switch between local mode and remote mode of inter-VM communications. In addition, the first release of MemPipe is implemented as a solution in layer 3. It maintains the co-resident VM membership information asynchronously using a periodic update method. Also MemPipe checks the update to the co-residency information before determining whether to switch the inter-VM communication to the local shared memory channel.

6.3.2 Potential Impacts on Connection Management and Pending Data Handing. To better understand how race condition management may affect the correctness and performance of residency aware inter-VM communication mechanisms, Table IV shows the potential impact for connection management, for handling of pending data remained from previous local/remote connections, and for connection tear-down along two dimensions: (i) events leading to the change of co-resident VM membership, and (ii) operations affected by race condition.

Table IV. Impacts of Race conditions on Connection Management and Pending Data Handling

	VM addition		**VM removal**	
	VM migration in	**VM creation**	**Migration out**	**VM shutdown**
Connection establishment	No error Performance overhead	No error Deferred establishment	Error	Error
Pending data handling	N/A	N/A	Layer 2 approachs: Error Layer 3 approachs: No error	N/A
Connection tearing down	No error	N/A	Possible Error	N/A

Note that not all the events are included in above table. For example, events such as VM reboot, VM destroy are not enumerated, since they are similar to VM addition or VM removal under connection establishment or tear-down. The commonality of these events is that they all need co-resident VM information explicitly or implicitly to proceed.

For race conditions in the case of VM addition, no additional error will be introduced, though additional performance overhead may be experienced in some cases. Concretely, when VM_i is migrated to host A from another physical machine host B, if VMs on host A have not been notified of the addition of VM_i, then they can continue their remote communication with VM_i. Thus, no pending data needs to be handled and connection tearing down is performed as if VM_i were on host B, no error occurs. Once the list of co-resident VMs is updated, for example through periodic update scheme, the race condition will be naturally eliminated. Similarly, when VM_i is added to host A through dynamic VM deployment, before other VMs on the same host A become aware of the co-residency of VM_i, connections to VM_i cannot be set up until it is visible to its communicating VMs. Thus, no pending data and connection tearing down are needed.

For race conditions in the case of VM removal, the situations are somewhat more complicated. After VM_i is migrated out from current host A to host B, it needs to switch all its shared memory based communications with VMs on host A (if any) from its original local channel to remote channel between host A and host B. Without such switch, due to race condition, VM_i may attempt to communicate with VMs on host A through the local shared memory channels that were setup previously, which can lead to errors since without switching to the remote channel, VMs cannot communicate with each other across physical hosts. To enable such switch, we need tear-down the local channels between VM_i and VMs on host A. Upon the command of local connection tearing down, a local channel should not be released until all its pending data are processed. Regarding the pending data remained from original local channel, if the local channel is established below IP layer (layer 3), then data transfer error is tolerable and is transparent to end users since upper layer TCP/IP protocol has the ability to handle such error and to make amends, although the amending process may lead to additional overhead. On the other hand, if the local channel is in layer 2, the residency aware inter-VM communication mechanism itself will need to provide functionalities to guarantee the reliability of pending data transmission, as outlined above.

In summary, race conditions do not necessarily lead to errors. The general guideline for preventing race conditions is to ensure that the co-resident VM membership is updated immediately and synchronously with the event of VM addition or VM removal and before conducting any corresponding operations listed in Table IV. If the update to the co-residency information is done asynchronously and periodically, then the proper tuning of the settings for update cycle period is critical: smaller frequency leads to higher freshness quality at higher cost. With proper prevention or resolution methods, race conditions due to VM addition or removal do not happen with high frequency. Thus, light-weighted solutions that can guarantee correctness while maintaining acceptable performance are desirable.

Table V shows a comparison of the seamless agility support in existing representative shared memory inter-VM communication systems:

Table V. The feature of seamless Agility

	Xen Based						KVM Based			
	IVC	*XenSocket*	*XWAY*	*XenLoop*	*MMNet (Fido)*	*XenVMC*	*VMPI*	*Socket-outsourcing*	*Nahanni*	*MemPipe*
Co-resident VM membership maintenance	Yes Static	No	Yes Static	Yes Dynamic	Yes Dynamic	Yes Dynamic	No	No	Yes	Yes
Automatic switch between local and remote channels	Yes	No	Yes	Yes	Yes	Yes	No	No	No	Yes
Transparent VM live migration support	Not fully transparent	No	No	Yes	Yes	Yes	No	No	No	Yes
Dynamic VM deployment support	No	No	No	Yes	Yes	Yes	No	No	No	Yes

7. MULTILEVEL TRANSPARENCY: COMPARATIVE ANALYSIS

7.1 User-Level Transparency

Among Xen based representative approaches, XWAY, XenLoop, XenVMC and MMNet achieve user-level transparency by lower layers design choices and no modification to layer 1. IVC is not user-level transparent for generic applications which use IVC library. However, for MPI applications, they can take the advantages of MVAPICH2-ivc without modification. XenSocket introduces XenSocket API, a new type of socket family to allow users to communicate across shared memory channels. The API is different from current standard sockets. The Grant Table reference is required to be passed to *connect()* explicitly as a parameter. It uses one shared variable to indicate the number of bytes for writes in the circular data channel. Thus in order to benefit from XenSocket, applications need to be developed/modified to incorporate the proposed interfaces.

Among KVM based representative approaches, Socket-outsourcing is implemented in layer 2. MemPipe is implemented in layer 3. Both keep the feature of user-level transparency. VMPI exposes shared memory message passing API that is not compatible with standard MPI. For Nahanni, user-level transparency and the ease of usage are sacrificed to achieve simplicity of implementation and potentially better performance. For example, to allow Memcached based applications to benefit from Nahanni, Memcached server is modified to use Nahanni API. Memcached client library is also extended to identify whether a Memcached server is local. Since Nahanni only provides local shared memory channel, it does not support inter-VM communication across physical machines.

7.2 Guest OS Kernel Transparency

Among the existing representative shared memory systems, XWAY and Socket-outsourcing are not OS kernel transparent. XWAY offers full binary compatibility for applications communicating over TCP sockets. However, it gives up the transparency of OS kernel by patching the kernel. Except for the patch, other part of XWAY is implemented as a virtual network device and its kernel driver. Socket-outsourcing modifies the kernel by replacing existing socket functions with self-defined ones.

XenSocket, XenLoop, XenVMC, MMNet, IVC, MemPipe, Nahanni and VMPI are OS kernel transparent. Among them, XenSocket, XenLoop, XenVMC and MemPipe are designed as kernel modules. MMNet, IVC, Nahanni and VMPI are implemented as kernel device driver modules. For instance, MMNet is in the form of a kernel driver module of a link layer device. The OS kernel part of IVC is a kernel driver module for a para-virtualized Xen network device. Nahanni's guest OS part is implemented as a UIO device driver for virtual PCI device *ivshmem*, which is created in the same way as a graphics device is. VMPI is implemented as kernel driver module of a virtual PCI character device for message passing.

7.3 VMM/Hypervisor Transparency

Xen based existing representative developments utilize existing Xen Grant Table, XenStore and Xen Event Channel to facilitate the design and implementation of shared memory communication channel and the notification protocol. Almost all of them keep the feature of VMM transparency except that IVC modifies the VMM to enable VM live migration.

KVM based development efforts, such as Nahanni, VMPI and Socket-outsourcing, are not VMM transparent. Before Nahanni is merged into QEMU as a sharing memory facility at the user-level, there is no mechanism provided by QEMU/KVM to support for host-guest and guest-guest memory sharing as Xen Grant Table does on Xen platform [Macdonell 2011]. For Nahanni, QEMU is modified to provide virtual device *ivshmem* to enable the management of shared memory. For VMPI, slight modifications are made to QEMU to implement the emulation of a new virtual device. For Socket-outsourcing, VMM is extended to provide the support for implementing facilities including shared memory, event queues, and VRPC. MemPipe [Zhang et al. 2015] is the recent shared memory inter-VM communication system on KVM platform, which provides VMM transparency in addition to user-level and guest OS kernel level transparency.

Table VI summarizes a comparison of the multilevel transparency feature in existing representative shared memory communication systems.

Table VI. Multilevel Transparency Features

	Xen Based						KVM Based			
	IVC	*XenSocket*	*XWAY*	*XenLoop*	*MMNet (Fido)*	*XenVMC*	*VMPI*	*Socket-outsourcing*	*Nahanni*	*MemPipe*
User-level transparency	No	No	Yes	Yes	Yes	Yes	No	Yes	No	Yes
OS kernel transparency	Yes	Yes	No	Yes	Yes	Yes	Yes	No	Yes	Yes
VMM level transparency	No	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes

We observe from Table VI that four existing shard memory developments meet the requirement of multilevel transparency. They are XenLoop, MMNet and XenVMC on Xen platform and MemPipe on KVM platform. These four systems also meet the seamless agility requirement according to Table V by supporting dynamic co-resident VM detection and automatic switch between local and remote channels. However, they employ different approaches in memory sharing and implementation layer: (i) MMNet maps the entire address space of the sender VM into the receiver VM's space before data transmission, while XenLoop, XenVMC and MemPipe map requested pages on demand; (ii) XenVMC builds shared memory based local channels in layer 2, and XenLoop, MMNet and MemPipe establish local channels in layer 3; and (iii) XenLoop, XenVMC and MMNet manage the shared memory by static allocation of shared memory regions to the communicating VMs on the same host, whereas MemPipe uses dynamic proportional resource allocation mechanism to manage shared memory regions for communicating VMs on a physical host.

8. PERFORMANCE COMPARISONS AND IMPACT OF IMPLEMENTATION CHOICES

8.1 Functional Choices

TCP and UDP support. TCP and UDP are two of the most commonly used network protocols. TCP is connection-oriented while UDP is connectionless. They provide different level of reliability and their performance differs due to different features they offer. Typically TCP and UDP protocols are used in different application areas. Among existing shared memory inter-VM communication systems, XenLoop, MMNet, XenVMC and Socket-outsourcing currently support both TCP and UDP workloads. While XenSocket and XWAY to date support only inter-VM communication for TCP oriented applications. For VMPI and Nahanni, only local channels are provided, neither TCP nor UDP is supported. For IVC, the technical detail for TCP/UDP support is not available.

Blocking I/O and non-blocking I/O. Shared memory buffer access algorithms can be implemented to support either blocking or non-blocking I/O. With blocking I/O, a thread is blocked until the I/O operations are finished. With non-blocking I/O, the functions return immediately after activating the read/write operations. Generally speaking, blocking I/O mode is easy to implement but less efficient than non-blocking I/O mode. XWAY and XenVMC support both modes, which is desirable for shared memory based inter-VM communication mechanisms. IVC offers *ivc_write* and *ivc_read* functions that are implemented as non-blocking I/O mode. The socket operations in host module of Socket-outsourcing indicate its non-blocking I/O feature. XenSocket does not support non-blocking I/O mode.

Host-guest and guest-guest communication. For shared memory approaches on Xen platform, they all leverage on Xen Grant Table, and offer optimized inter-VM communication across domains. In contrast, for KVM platform, the communicating VMs are either host OS or guest OS, thus host-guest and guest-guest are the two typical modes of inter-VM communication. Based on different design objectives and different implementation techniques adopted, KVM based shared memory systems support different communication types: Nahanni supports both host-guest and guest-guest communication. In Socket-outsourcing, its guest module allows the host module to access its memory region through VRPC and Event interfaces between the host module and the guest module. VMPI explicitly supports guest-guest communication. MemPipe provides direct support for host-guest communication and indirect support for guest-guest communication.

Buffer size. A majority of existing shared memory developments choose to design and implement the buffer for data sharing as FIFO circular structure. Experimental observation shows that the FIFO buffer size may impact on achievable bandwidth [Wang et al. 2008b]. IVC, XenLoop and XenVMC support tunable buffer size to offer tunable bandwidth. Also in XenLoop, if the packet size is larger than the FIFO buffer size, then the packet is transmitted by traditional network path. XenSocket utilizes fixed-size buffer that is comprised of 32 buffer pages and each page size is 4KB.

Table VII provides a comparison on different implementation choices by existing shared memory implementations.

Table VII. Additional Implementation Choices

	Xen Based						KVM Based			
	IVC	*XenSocket*	*XWAY*	*XenLoop*	*MMNet (Fido)*	*XenVMC*	*VMPI*	*Socket-outsourcing*	*Nahanni*	*MemPipe*
TCP/UDP support	N/A	TCP	TCP	Both	Both	Both	N/A	Both	N/A	Both
Blocking / non-blocking	Non-blocking	Blocking	Both	N/A	N/A	Both	N/A	N/A	N/A	Both
Host-guest / guest-guest	Inter domain	Inter domain	Inter domain	Inter domain	Inter domain	Inter domain	Guest-guest	Host-guest	Both	Both
Data buffer size	Tunable	4KB*32	N/A	Tunable	N/A	Tunable	N/A	N/A	N/A	Tunable

8.2 Software Environment and Source Code Availability

All existing representative shared memory channels are implemented and evaluated exclusively with different versions of Linux kernel and VMM as shows in Table VIII. Among them, XenSocket [Zhang and Mcintosh 2013b], XWAY [Kim et al. 2013], XenLoop [Wang et al. 2008], Nahanni [Macdonell 2014] and MemPipe [Zhang et al. 2015] are open source systems. Nahanni open source code is included into the QEMU/KVM release since its version 0.13 from August 2010. It has become a part of QEMU/KVM, the default hypervisor in Ubuntu and Red Hat Enterprise Linux, which are two of the major Linux distributions.

Table VIII. Software Environment and Source Code Availability

	Xen Based						KVM Based			
	IVC	*XenSocket*	*XWAY*	*XenLoop*	*MMNet (Fido)*	*XenVMC*	*VMPI*	*Socket-outsourcing*	*Nahanni*	*MemPipe*
Linux kernel version	2.6.6.38	2.6.6.18	2.6.16.29	2.6.18.8	2.6.18.8	3.13.0	N/A	2.6.25	Since 2.6.37	3.2.68
VMM version	Xen 3.0.4	Xen 3.0.2	Xen 3.0.3 Xen3.1	Xen 3.2.0	Xen 3.2	Xen 4.5	N/A	KVM-66	Since QEMU 0.13	KVM 3.6 / QEMU 1.2.0
Source code availability	N/A	Open source	Open source	Open source	N/A	N/A	N/A	N/A	Open source	Open Source

8.3 Performance Comparison

As indicated in Table VIII, most of the existing representative shared memory inter-VM communication systems to date have not provided open source release of their implementations. For Xen based systems, only XenSocket, XWAY and XenLoop have released their software. XenVMC was developed by the authors of NUDT. For KVM based systems, only Nahanni and MemPipe release their systems as open source software. However, among the above systems, XenSocket and Nahanni are not user-level transparent (Table VI), which means existing applications and benchmarks cannot be easily deployed to run on XenSocket or Nahanni without modification. Therefore, they do not support Netperf, the widely used network I/O benchmark. As a result, XWAY, XenLoop, XenVMC and MemPipe become candidates for the performance comparison. XenLoop and MemPipe are implemented in layer 3 and XWAY and XenVMC are implemented at layer 2. Also compared with XenLoop, XenVMC and MemPipe, which support both TCP and UDP semantics and meet all three design criteria: shared memory optimization, seamless agility and multilevel transparency, XWAY falls short on several aspects: (i) XWAY does not support UDP and (ii) XWAY does not support seamless agility or OS kernel transparency. Taking the above different factors into consideration, we provide a comparison of performance based on the reported experimental results from related papers [Huang et al. 2007; Zhang et al. 2007; Kim et al. 2008; Wang et al. 2008b; Burtsevet al. 2009; Ren et al. 2012; Diakhaté et al. 2008; Eiraku et al. 2009; Koh 2010; Gordon 2011a; Gordon et al. 2011b; Ke 2011; Macdonell 2011, Zhang et al. 2015], including experiment setups, test cases or benchmarks, comparison dimensions, comparison systems, and the description of performance improvement. Table IX shows the summary information. Given that the experimental environments and system configurations vary from one system to another, we use the normalized numbers in the performance improvement column to make comparison more straightforward.

Table IX. Performance Comparison for Existing Shared Memory Systems

Plat.	Name	Hardware Setup	Test Case /Benchmark	Dimensions (Netperf)	Contrast Systems	Normalized Performance Improvement
Xen	IVC	2 2.8GHz 4GB, 2 3.6GHz 2GB, 2 quad-core 2.0GHz 4GB, PCI-E IB HCAs	Intel MPI, NAS parallel, LAMMPS, NAMD, SMG2000, HPL	N/A (Migration)	IVC, inter-VM, native Linux	Near native Linux NAS parellel: up to 11%+
	XenSocket	2 2.8GHz CPU, 4GB RAM	Netperf 2.4.2	TCP STREAM TCP RR	Native Linux, inter-VM, XenSocket	Inter-VM: up to 72X
	XWAY	3.2GHz CPU, 1GB RAM	Netperf 2.4.3, Apps(scp, ftp, wget), DBT-1	N/A (Connection overhead)	Unix Domain, TCP (Xen 3.0/3.1, Page Flip/Copy), XWAY	Better than native TCP socket Binary compatable
	XenLoop	dual-core 2.8GHz CPU, 4GB RAM	Netperf, lmbench, netpipe-mpich, OSU MPI, ICMP ECHO REQUEST/REPLY	UDP SREAM TCP RR UDP RR	Inter machine, netfront, XenLoop, loopback	Latency: reduces by up to 5X Bandwidth: increases by up to 6X
	MMNet (Fido)	2 quad-core 2.1GHz CPU, 16 GB RAM, 2 1Gbps NICs	Netperf 2.4.4	TCP SREAM UDP SREAM TCP RR	Loopback, netfront, XenLoop, MMNet	TCP&UDP STREAM: about 2X (XenLoop) up to 4X (Netfront)
	XenVMC	2.67GHz CPU 4GB RAM	Netperf 2.4.5	TCP SREAM TCP RR	Netfront, XenVMC	thoughput: up tp 9X latency: improves by up tp 6X
KVM	VMPI	2 quad-core 2.33GHz CPU, 4GB RAM	Pingpong benchmark	N/A	MPICH2	near native performance
	Socket-outsourcing	3GHz CPU, 2GB RAM, 4 Gigabit NICs	Iperf, RUBiS benchmark 1.4.3	N/A	KVM-emu, KVM-virtio, KVM-out	iperf: up to 25X RUBiS: 45%+ (KVM-emu)
	Nahanni	2 quad-core 2.67GHz CPU, 48GB RAM	Modified: GAMESS, SPEC MPI2007, Memcached, MPICH2	N/A	Inter-VM	20-80% +
	MemPipe	quad-core 2.4GHz CPU, 4GB RAM, Gigabit NIC	Netperf, network apps (scope, wget, sftp…)	TCP SREAM UDP SREAM TCP RR UDP RR	Inter machine, inter-VM, MemPipe	Inter machine: 1.1X-3X Inter VM: 2.5X-65X

Note that in Table IX, the normalized performance number is with respective to each shared memory system compared to inter-machine, inter-VM and loopback scenarios, etc. Given that the experimental results are obtained with different versions of hypervisor, Linux kernel and under diverse hardware configurations, thus the higher/lower performance numbers should not be interpreted as an absolutely better/worse throughput or lower/higher latency.

9. CONCLUSION

We have presented an in-depth analysis and comparison of the state-of-the-art shared memory based techniques for inter-VM communication. To the best of our knowledge, this is the first effort that provides a comprehensive survey of the Inter-VM communication methods. Concretely, this paper makes two original contributions. *First,* we present the main design and implementation choices of the shared memory based inter-VM communication methods with respect to the implementation layer in software stack, the support of seamless agility and the guarantee of multilevel transparency. *Second*, based on the objectives and design choices, we present a comparative analysis on a selection of important issues, including how to achieve high performance, how to guarantee multilevel transparency, how to automatically detect co-resident VMs, how to support automatic switch between local and remote mode of shared-memory based inter-VM communication so that dynamic addition or removal of co-resident VMs can be supported, to name a few. We conclude that the implementation layer may have critical impact on transparency, performance, as well as seamless agility. *Third* but not the least, we provide an extensive comparison on shared memory implementations on top of two most popular and yet architecturally different open source VMM platforms, Xen and KVM. The comparison covers architectural layout, fundamental functionalities, seamless agility, multilevel transparency, additional implementation consideration, software environment and source code availability. We conjecture that this survey provides not only a comprehensive overview of important issues, design choices and practical implementation techniques for developing the next generation of shared memory based inter-VM communication mechanisms, but also offers both cloud infrastructure providers and cloud service consumers an opportunity to further improve inter-VM communication efficiency in virtualized data centers.

Acknowledgement.
The first author started this work while visiting Georgia Tech as a visiting scholar funded by CRC, China and the authors of Georgia Institute of Technology is partially supported by by the National Science Foundation under Grants IIS- 0905493, CNS-1115375, NSF 1547102, SaTC 1564097, and Intel ISTC on Cloud Computing.

REFERENCES

ANDERSON, T., BIRMAN, K., BROBERG, R., CAESAR, M., COMER, D., COTTON, C., FREEDMAN, M. J., HAEBERLEN, A., IVES, Z. G.,KRISHNAMURTHY, A., LEHR, W., LOO, B. T., MAZIERES, D., NICOLOSI, A., SMITH, J. M., STOICA, I., RENESSE, R. V., WALFISH, M., WEATHERSPOON, H., AND YOO, C. S. 2013. The NEBULA future Internet architecture. *Lecture Notes in Computer Science. 7858*. 16-26.

ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. 2010. Above the clouds: a Berkeley view of cloud computing. *Communications of the ACM. 53*, 4 (April 2010). 50-58.

AYUSO, P. N. 2006. Netfilter's connection tracking system. *USENIX*, 31, 3. http://people.netfilter.org/pablo/docs/login.pdf.

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. 2003. Xen and the art of virtualization. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM New York, NY, USA. 164-177.

BURTSEV, A., SRINIVASAN, K., RADHAKRISHNAN, P., BAIRAVASUNDARAM, L. N., VORUGANTI, K., AND GOODSON, G. R. 2009. Fido: fast inter virtual-machine communication for enterprise appliances. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*.http://static.usenix.org/events/usenix09/tech/full_papers/burtsev/burtsev.pdf.

DIAKHATÉ, F., PERACHE, M., NAMYST, R., AND JOURDREN, H. 2008. Efficient shared memory message passing for inter-VM communications. In *Proceedings of 3rd Workshop on Virtualization in High-Performance Cluster and Grid Computing (VHPC'08)*. Springer-Verlag Berlin, Heidelberg.53-62.

EIRAKU, H., SHINJO, Y., PU, C., KOH, Y., AND KATO, K. 2009. Fast networking with socket-outsourcing in hosted virtual machine environments. In *Proceedings of ACM Symposium on Applied Computing (SAC'09)*. ACM New York, NY, USA.310-317.

GEBHARDT, C., AND TOMLINSON, A. 2010. Challenges for inter virtual machine communication. Technical Report, RHUL-MA-2010-12 (September 2010). 1-18. http://www.ma.rhul.ac.uk/static/techrep/2010/RHUL-MA-2010-12.pdf.

GORDON, A. W. 2011a. Enhancing cloud environments with inter-virtual machine shared memory. M.S. thesis, Department of Computing Science, University of Alberta.

GORDON, A. W., AND LU, P. 2011b. Low-latency caching for cloud-based Web applications. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB'11)*. June 12-16, 2011, Athens, Greece. http://research.microsoft.com/en-us/um/people/srikanth/netdb11/netdb11papers/netdb11-final14.pdf.

GOVINDAN, S., NATH, A. R., DAS, A., URGAONKAR, B., AND SIVASUBRAMANIAM, A. 2007. Xen and

Co.: communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In *Proceedings of the 3rd International Conference on Virtual Execution environments (VEE'07).* San Diego, California, USA, June 13-15, 2007. ACM 2007, ISBN 978-1-59593-630-1. 126-136.

GURAV, U., AND SHAIKH R. 2010. Virtualization: a key feature of cloud computing. In *Proceedings of the International Conference and Workshop on Emerging Trends in Technology (ICWET '10).* 227-229.

HUANG, W. 2008. High performance network I/O in virtual machines over modern interconnects. Ph.D. thesis, Department of Computer Science and Engineering, Ohio State University.

HUANG, W., KOOP, M., GAO,Q., AND PANDA D.K. 2007.Virtual machine aware communication libraries for high performance computing. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC'07).* ACM New York, NY, USA. Article No. 9.

HWANG, J., RAMAKRISHNAN, K., AND WOOD, T. 2014. NetVM: high performance and flexible networking using virtualization on commodity platforms. In *Proceedings of 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14).* Seattle, 2014. WA: USENIX Association. 445–458.

IMAI, S., CHESTNA, T., AND VARELA, C. A. 2013. Accurate resource prediction for hybrid IaaS clouds using workload-tailored elastic compute Units. In *Proceedings of 6th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2013).* Dresden, Germany, December 2013. IEEE. 171-178.

Infiniband Trade Association. 2015. http://www.infinibandta.org.

Intel Corporation. 2013. Intel data plane development kit: Getting started guide. http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/intel-dpdk-programmers-guide.html.

JOSE, J., LI, M., LU, X., KANDALLA, K., ARNOLD, M., AND PANDA, D. K. 2013. SR-IOV support for virtualization on InfiniBand clusters: early experience. In *Proceedings of 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'13).* Delft, the Netherlands, May 13-16, 2013. IEEE. 385-392.

KE, X. 2011.Interprocess communication mechanisms with inter-virtual machine shared memory. M.S. thesis, Department of Computing Science, University of Alberta.

KIM, H., LIM, H., JEONG, J., JO, H., AND LEE, J. 2009. Task-aware virtual machine scheduling for I/O performance. In *Proceedings of the 5th International Conference on Virtual Execution Environments (VEE'09).* Washington, DC, USA, March 11-13, 2009. ACM 2009, ISBN 978-1-60558-375-4. 101-110.

KIM, K. 2013. XWAY project. http://xway.sourceforge.net/

KIM, K., KIM, C., JUNG, S., SHIN, H., AND KIM, J. 2008. Inter-domain socket communications supporting high performance and full binary compatibility on Xen. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08).* ACM New York, NY, USA.11-20

KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., and LIGUORI, A. 2007. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium, volume 1. June 27th–30th, 2007.* Ottawa, Ontario, Canada 2007. 225–230.

KOH, Y. 2010. Kernel service outsourcing: an approach to improve performance and reliability of virtualized systems. Ph.D. thesis, School of Computer Science, College of Computing, Georgia Institute of Technology.

LEE, M., KRISHNAKUMAR, A. S., KRISHNAN, P., SINGH, N., AND YAJNIK, S. 2010. Supporting soft real-time tasks in the Xen hypervisor. In *Proceedings of the 6rd International Conference on Virtual Execution environments (VEE'10).* Pittsburgh, PA, March 17-19, 2010. 97-108.

LEVON, J. 2014. OProfile manual. http://oprofile.sourceforge.net/doc/index.html.

LI, D., JIN, H., SHAO, Y., LIAO, X., HAN, Z., AND CHEN, K. 2010. A high-performance inter-domain data transferring system for virtual machines. *Journal of Software. 5*, 2 (February 2010). 206-213.

LIU, J., HUANG, W., ABALI, B., AND PANDA, D. K. 2006. High performance VMM bypass I/O in virtual machines. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference (ATEC'06).* 29-42.

MACDONELL, A. C. 2011. Shared-memory optimizations for virtual machines. Ph.D. thesis, Department of Computing Science, University of Alberta.

MACDONELL, A. C. 2014. Nahanni: the KVM/Qemu inter-VM shared memory PCI device. http://gitorious.org/nahanni/pages/Home.

MEI, Y., LIU, L., PU, X., SIVATHANU, S., AND DONG, X. 2013. Performance analysis of network I/O workloads in virtualized data centers. *IEEE Trans. Services Computing. 6*, 1, 48-63.

MENON, A., COX, A. L., AND ZWAENEPOEL, W. 2006. Optimizing network virtualization in Xen. In *Proceedings of the 2006 Conference on USENIX Annual Technical Conference.* USENIX Association Berkeley, CA, USA. 15-28.

MSDN. 2014. Overview of single root I/O virtualization (SR-IOV). http://msdn.microsoft.com/en-us/library/windows/hardware/hh440148%28v=vs.85%29.aspx.

Netperf. 2015. http://www.netperf.org/netperf/.

ONGARO, D., COX, A. L., AND RIXNER S. Scheduling I/O in virtual machine monitors. 2008. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE'08).* Seattle, WA, USA, March 5-7, 2008. ACM 2008, ISBN 978-1-59593-796-4. 1-10.

PEARCE, M., ZEADALLY, S., AND HUNT R. 2013. Virtualization: issues, security threats, and solutions.

ACM Computing Surveys. February 2013. Vol. 45, No. 2, Article 17.

PU, X., LIU, L., MEI, Y., SIVATHANU, S., KOH, Y., PU, C., AND CAO, Y. 2012. Who is your neighbor: net I/O performance interference in virtualized clouds. *IEEE Trans. Services Computing. 6*, 3, 314-329.

RADHAKRISHNAN, P., AND SRINIVASAN, K. 2008. MMNet: an efficient inter-VM communication mechanism. Xen Summit Boston2008. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.178.409.

REN, Y., LIU, L., LIU, X., KONG, J., DAI, H., WU, Q., AND LI,Y. 2012. A fast and transparent communication protocol for co-resident virtual machines. In *Proceedings of 8th IEEE International Conference on Collaborative Computing (CollaborateCom'12)*.70-79.

RENESSE, R. V. 2012. Fact-based inter-process communication primitives for programming distributed systems. In Proceedings of Workshop on Languages for Distributed Algorithms (LADA 2012). http://www.cs.cornell.edu/home/rvr/newpapers/lada2012.pdf.

RUSSELL, R. 2008. Virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review.42*, 5 (July 2008), 95-103.

SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND PRATT, I. 2008. Bridging the gap between software and hardware techniques for I/O virtualization. In *Proceedings of USENIX 2008 Annual Technical Conference on Annual Technical Conference (ATC'08)*. 29-42.

SANGER, R (Editor). 2013. Notes on libtrace Intel data plane development kit (DPDK) support - experimental. https://github.com/wanduow/libtrace/wiki/DPDK-Notes---Experimental.

SU, Z., LIU, L., LI, M., FAN, X., AND ZHOU, Y. 2015. Reliable and resilient trust management in distributed service provision networks. To appear in ACM Transactions on the Web (TWEB), 2015.

VMware, Inc. 2007. VMCI Overview. http://pubs.vmware.com/vmci-sdk/VMCI_intro.html.

WANG, J. 2009. Survey of state-of-the-art in inter-VM communication mechanisms. Research Report (September 2009). http://www.cs.binghamton.edu/~jianwang/papers/proficiency.pdf.

WANG, J., WRIGHT, K., AND GOPALAN, K. 2008a. XenLoop source code. http://osnet.cs.binghamton.edu/projects/xenloop-2.0.tgz.

WANG, J., WRIGHT, K., AND GOPALAN, K. 2008b. XenLoop: a transparent high performance inter-VM network loopback. In *Proceedings of the 17th ACM International Symposium on High Performance Distributed Computing (HPDC '08)*. ACM New York, NY, USA. 109-118.

WANG, Q. AND VARELA, C. A. 2011. Impact of cloud computing virtualization strategies on workloads' performance. 2011. In *Proceedings of4th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2011)*. 130-137.

YEHUDA, M. B., MASON, J., XENIDIS, J., KRIEGER, O., DOORN, L. V., NAKAJIMA, J., MALLICK, A., AND WAHLIG, E. 2006. Utilizing IOMMUs for virtualization in Linux and Xen. In *Proceedings of the 2006 Ottawa Linux Symposium (OLS'06)*. 71-86.

YOUNGE, A. J., HENSCHEL, R., BROWN, J. T., LASZEWSKI, G., QIU, J., AND FOX, G. C. 2011. Analysis of virtualization technologies for high performance computing environments. In *Proceedings of IEEE 4th International Conference on Cloud Computing (CLOUD '11)*. 9-16.

ZHANG, Q., LIU, L., REN, Y., Lee, K., Tang, Y., Zhao, X., AND Zhou, Y. 2013a. Residency aware inter-VM communication in virtualized cloud: performance measurement and analysis. In *Proceedings of IEEE 6thInternational Conference on Cloud Computing (CLOUD '13)*. IEEE. 204 - 211.

ZHANG, Q. AND LIU, L. 2015. Shared memory optimization in virtualized clouds. In *Proceedings of IEEE 2015 International Conference on Cloud Computing (CLOUD'15)*. New York, June 27-July 1, 2015. Note: In press. http://www.cc.gatech.edu/~lingliu/papers/2015/cloud2015-SharedMem.pdf.

ZHANG, J., LU, X., JOSE, J., SHI, R., AND PANDA, D.K. 2014a. Can inter-VM shmem benefit MPI applications on SR-IOV based virtualized Infiniband clusters? In *Proceedings of Euro-Par 2014 Parallel Processing - 20th International Conference*. Porto, Portugal, August 25-29, 2014. Springer, 342-353.

ZHANG, J., LU, X., JOSE, J., LI, M., SHI, R., AND PANDA, D. K. 2014b. High performance MPI library over SR-IOV enabled InfiniBand clusters. In *Proceedings of 21st annual IEEE International Conference on High Performance Computing (HiPC 2014)*. Goa, India, 2014.

ZHANG, X. AND MCINTOSH, S. 2013b. XVMSocket. http://sourceforge.net/projects/xvmsocket/.

ZHANG, X., MCINTOSH, S., ROHATGI, P., AND GRIFFIN, J. L. 2007. XenSocket: a high-throughput inter domain 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.184-203.