Residency-Aware Virtual Machine Communication Optimization: Design Choices and Techniques

Conference Paper · June 2013

DOI: 10.1109/CLOUD.2013.117

8 authors, including:

Yi Ren
National University of Defense Technology
19 PUBLICATIONS **101** CITATIONS

SEE PROFILE

Ling Liu
Georgia Institute of Technology
672 PUBLICATIONS **21,357** CITATIONS

SEE PROFILE

Qi Zhang
Meta
75 PUBLICATIONS **1,617** CITATIONS

SEE PROFILE

Residency-Aware Virtual Machine Communication Optimization: Design Choices and Techniques

Yi Ren
College of Computer Science
National University of Defense Technology
Changsha, P. R. China, 410073
renyi@nudt.edu.cn

Ling Liu, Qi Zhang
School of Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332–0250, USA
{lingliu, qzhang90}@cc.gatech.edu

Qingbo Wu, Jie Yu, Jinzhu Kong, Jianbo Guan, Huadong Dai
College of Computer Science
National University of Defense Technology
Changsha, P. R. China, 410073
guanjb@nudt.edu.cn

***Abstract*—Network I/O workloads are dominating in many data centers and cloud computing environments today. One way to improve inter Virtual Machine (VM) communication efficiency is to support co-resident VM communication by using shared memory based approaches and to resort to the traditional TCP/IP for inter-VM communications between VMs that are located on different physical hosts. Although a number of independent efforts are dedicated to improving communication efficiency between co-resident VMs, they differ from one another in terms of how the inter-VM communication optimization is carried out and where in the software stack the shared memory channel is established. In this paper, we provide an in-depth overview of the design choices and techniques for optimizing the performance of the co-resident inter-VM communication, with dual objectives. First, we describe the core design guidelines and key issues for optimizing inter-VM communication by using shared memory based mechanisms. Typical issues include choices of implementation layer in the software stack, seamless agility for VM live migration and VM dynamic deployment support, multilevel transparency. Second, we conduct a comprehensive analysis of representative state-of-the-art research efforts and implementation techniques based on the core design guidelines. We also give an analysis of future requirements in advanced features such as reliability, security and stability. The research reported in this paper not only provides the reference for developing the next generation of inter-VM communication optimization mechanisms, but also offers opportunities for both cloud infrastructure providers and cloud service consumers to improve inter-VM communication efficiency in virtualized platforms.**

Keywords-residency-aware; inter-VM communication; shared memory; seamless agility; multilevel transparency

I. INTRODUCTION

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

Although VMM technology offers significant benefits in terms of functional and performance isolation, live migration based load balance, fault tolerant, portability of applications, 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 host are non-competing in terms of network and computing resources. However, the performance of VMs is significantly degraded compared to that of native machine when co-resident VMs are competing for resources under high workload demands due to high overheads of switches and events in host/guest domain and VMM [1]. Second, the communication overhead 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 co-resident VMs or not. Several research projects [2-11] have demonstrated that. Linux guest domain shows lower network performance than native Linux [12], when an application running on a VM communicates with another VM. [4] showed that with copying mode, the inter-VM communication performance is enhanced but still lagging behind compared to the performance on native Linux, especially for co-resident VMs.

There are two main reasons for the performance degradation of co-resident VM communication [3, 5-6]: (i) long communication data path through the TCP/IP network stack, (ii) lack of communication awareness in CPU scheduler and absence of real time inter-VM interactions. Thus in order to improve the performance of network intensive applications running in virtualized computing environments, there are two categories of solutions to improve the performance of inter-VM communication: one is to use memory sharing approach for co-resident VMs to improve both communication throughput and latency, and the other is to reduce inter-VM communication latency by optimizing CPU scheduling policies. One of the main arguments is that TCP/IP based network communication is inefficient when co-resident VMs communicate with one another since TCP/IP was originally designed for inter physical machine communication via LAN/Internet. To date, most of the research and kernel development efforts reported in literature are based on shared memory to improve communication efficiency among co-resident VMs. And most of the documented efforts are centered on open source hypervisors, such as Xen and KVM. Thus, in this paper we

978-0-7695-5028-2/13 $26.00 © 2013 IEEE
DOI 10.1109/CLOUD.2013.117

present a comprehensive survey on shared memory based techniques for inter-VM communication optimization.

Relatively speaking, there are more shared memory efforts on Xen platform than KVM platform in the literature. Related work such as IVC [2], XenSocket [3], XWAY [4], XenLoop [5], MMNet [6] above Fido [7] and XenVMC [8] are on Xen platform. While all KVM-based efforts, such as VMPI [9], Socket-outsourcing [10] and Nahanni [11], are recent development since 2009. One reason could be due to the fact 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.

In this paper, we provide an in-depth overview of the design choices and techniques for optimizing the performance of the co-resident inter-VM communication, with dual objectives. First, we describe the core design guidelines and key issues for optimizing inter-VM communication using shared memory based mechanisms. Typical issues include choices of implementation layer in the software stack, seamless agility for VM live migration and VM dynamic deployment support, multilevel transparency. Second, based on the proposed guidelines, we conduct a comprehensive survey of representative state-of-the-art research efforts on both Xen and KVM platforms using a structured approach. We also give an analysis of future requirements in advanced features such as reliability, security and stability. We conjecture that the research reported in this paper is beneficial not only to researchers and developers of the next generation inter-VM communication optimization mechanisms, but also to cloud infrastructure providers and cloud service consumers that run applications on the virtualized platform.

The rest of this paper is organized as follows. Section II provides an overview of the basic concepts and terminology of network I/O and shared memory structures in VMMs. Section III discusses a selection of design guidelines and key issues for shared memory based communication mechanisms for co-resident VMs. Section IV makes a comprehensive comparison of existing work using a number of indicators based on the design guidelines. Section V analyzes further requirements in other desired features, such as reliability, security and stability. Section VI concludes the paper.

II. BACKGROUND AND PRELIMINARY

In this paper, we focus on representative work based on typical open source VMMs, Xen and KVM.

A. Xen Network Architecture and Interfaces

1) Network I/O Architecture

Xen is an open-source x86/x64 hypervisor coordinates the low-level interaction between VMs and physical hardware [13]. It supports both full-virtualization and para-virtualization. The latter provides a more efficient and lower overhead mode of virtualizations. In this mode, Dom0, a privileged domain, performs the tasks to create, terminate and migrate guest VMs (DomU). Xen exports virtualized network devices 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. Fig. 1 illustrates the network I/O architecture and interfaces. The frontend and the 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.

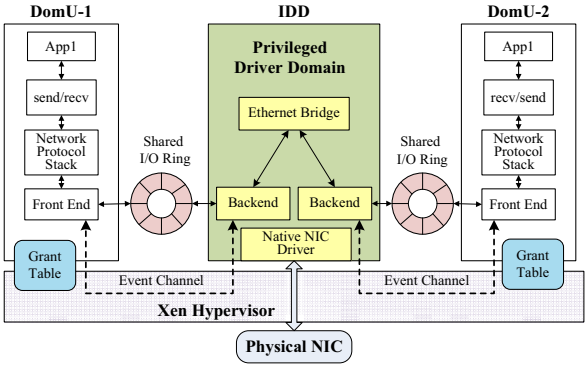

Figure 1. Xen network I/O architecture and interfaces

Each I/O operation in either split I/O or Direct I/O model requires involvement of the hypervisor or Dom0, which may turn out to be a performance bottleneck for I/O intensive systems While VMM-bypass I/O model allows time-critical I/O operations to be processed directly in guest VMs without involvement of the hypervisor or a privileged VM.

2) Communication Mechanisms Among Domains

As shown in Fig. 1, the shared I/O Ring buffers between the frontend and the backend are built upon the grant table and event channel mechanisms provided by Xen. Grant table work as a generic mechanism to share pages of data between domains. It 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. XenStore is a configuration and status information storage space shared between domains. The information is organized hierarchically. Each domain gets its own path in the store.

B. QEMU/KVM

KVM is an open-source solution for Linux on x86 hardware that supports virtualization extension. It consists of two parts: one is QEMU, which is a hardware emulator running in the host OS as a user level process and provides I/O device model for VM; the other is a 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. The OS running in a VM is called guest OS. There are two modes of I/O virtualization for KVM.

1) Full-Virtualized Network I/O by Device Emulation

Device emulation is provided by user space QEMU. It makes the guest OS using the device interacts with the device as if it were actual hardware rather than software. And there is no need to modify corresponding device driver in the guest OS. Fig. 2 illustrates the architecture of KVM full-virtualized network I/O. When the guest OS tries to access the emulated 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 virtualized NIC (VNIC) and copies the data into TAP device. And the data is forwarded to the device of destination guest OS by the software bridge. When the TAP device receives the data, it wakes up the QEMU process, who first copies the data into its VNIC buffer, from where copies the data to the virtual device in the destination guest OS. After that, QEMU notify the KVM kernel module to receive the data. And the module sends interrupts to notify the guest OS about the data arriving. Through virtual driver and network protocol stack, the data is passed to applications.

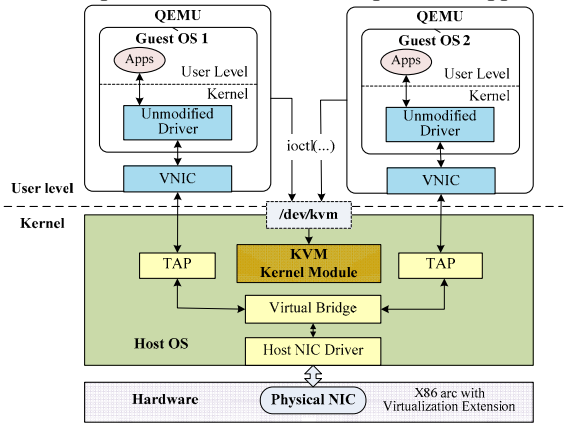

Figure 2. The architecture of KVM full-virtualized network I/O

2) Virtio based Para-Virtualized 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, as well as that between Linux kernel and user space QEMU. Therefore, Virtio, a para-virtualized I/O framework was introduced. Virtio avoids unnecessary I/O operations and reduces the number of data copies and context switches between the kernel and user space.

III. Overview of Design Choices

In this section we first describe the motivation and objectives for designing a shared memory based approach for residency-aware inter-VM communication. Then we present design guidelines and key issues for designing and implementing a high performance inter-VM communication protocol based on shared memory mechanisms.

A. Why 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 super efficient manner through shared memory since data shared between processes can be immediately visible to each other.

Compared with IPC, communication via TCP/IP takes longer time because data transfer 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, in KVM platforms, data transfer 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 and bypassing TCP/IP, we may gain a number of performance optimization opportunities: the number of data copies is reduced by shortening the data transmission path, unnecessary switches between VM and VMM are avoided by reducing dependency to VMM, and 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.

B. Functionality and Design Objectives

A shared memory based approach for inter-VM communication should be highly efficient, highly transparent and seamlessly agile in the presence of VM live migration and dynamic deployment. To support these design objectives, the following three core capabilities should be provided: (i) determining 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 mode of communication, automatically switching to the shared memory based channel, and (iii) the shared memory based inter-VM communication should be incorporated into the existing virtualized platform in an efficient and fully transparent manner [4-8, 10].

The design objective of high performance aims at improving network throughput by supporting VM residency-aware approach, which is capable of distinguishing communication between co-resident VMs from communication between VMs on separate physical hosts, so that the communication path between co-resident VMs is cut short and the communication overhead is reduced. The design objective of seamless agility calls for support of on-demand detection and automatic switching between local mode (co-resident inter-VM communication) and remote mode (remote inter-VM communication) to ensure that the VM live migration and the agility for VM deployment are retained. Finally, the design objective of the transparency of the shared memory based mechanism over programming languages, guest OS kernel and VMM ensures that there is no need to make code modifications, recompilation or re-linking to support legacy applications.

C. Implementation Layer in Software Stack

By supporting the shared memory based inter-VM communication mechanism to co-exist with the TCP/IP based inter-VM communication protocol, we can shorten the data transfer path and minimize the communication overhead between co-resident VMs. To achieve this, we need to intercept every outgoing data requests, examine and detect

whether the receiver VM is co-resident with the sender VM, and if so, redirect the outgoing data request to a co-resident VM communication protocol instead. To provide seamless agility, the switching between local mode and remote mode should be done automatically and transparently.

The shared memory based inter-VM communication protocol can be implemented in three alternative ways based on the choice of the implementation layer in the software stack where the shared memory channel is established: (i) user libraries and system calls layer (or layer 1), (ii) below system calls layer and above transport layer (or layer 2), and (iii) below IP layer (or layer 3). Implementation in different layers may bring different impacts on programming transparency, kernel-hypervisor level transparency, seamless agility and performance overhead.

1) User Libraries and System Calls Layer

One way to implement the shared memory based inter-VM communication protocol is to modify the standard user and programming interfaces in the user libraries and system calls layer. This approach is simple and straightforward. It introduces less switching overhead for crossing two protection barriers: from guest user level to guest kernel level and from guest OS to host OS. However, it exposes shared memory up to the user level applications running in guest VMs and fails to maintain the programming transparency. Thus, it is up to the application programmers to manually incorporate the shared memory based inter-VM communication optimization into their application systems.

2) Below System Calls Layer & Above Transport Layer

An alternative approach to implement shared memory based inter-VM communication is below system calls layer and above transport layer. Due to hierarchical structure of TCP/IP network stack, when data is sent through the stack, it has to be encapsulated with additional headers layer by layer in the sender node. When the encapsulated data reaches the receiver node, the headers will have to be removed layer by layer. Thus 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). This observation makes us conjecture that implementation in higher layer can potentially lead to lower latency and higher throughput of network I/O workloads. But if the shared memory channel is established in Layer 1, it is hard to maintain programming language transparency. From this perspective, layer 2 solutions are more attractive.

3) Below IP Layer

Another alternative method is to implement the shared memory based inter-VM communication optimization below IP layer. There are several advantages of choosing implementation in this layer: (i) since the interception is below the TCP/IP protocol stack, the existing TCP/IP features, such as reliability and so on, remain untouched; and (ii) third party tool, such as netfilter [14], is available to hook into the long TCP/IP network path to facilitate the implementation of functionalities such as packets interceptions. However, layer 3 is lower in the software stack. As mentioned, it potentially lead to higher latency due to higher protocol processing overheads, more number of data copies and context switches across barriers. Thus if alternative approaches for ensuring reliability and implementing functionalities similar with those provided by third party tools are available, to achieve better performance and programming transparency, layer 2 is preferred.

D. Design Choices for Seamless Agility

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 carried out automatically and adaptively in the presence of VM live migration and VM dynamic deployment (e.g., on-demand addition or removal of VMs).

1) Automatic Detection of Co-resident VMs

When a shared memory based inter-VM channel is to be set up or torn down, we need to verify if the communicating VMs are co-resident or not. Two methods are usually used to maintain co-residency information. The first one is called the static method, which is primarily used when the membership of co-resident VMs is preconfigured or collected manually by the administrator and is assumed not to change during the network operations afterwards. Thus, user applications are aware of the co-residency information. The second one is called a dynamic method, which provides automatic detection mechanisms for co-resident VMs and thus conveniently support VM live migration and dynamic VM deployment. The static method cannot detect the arrival or departure of VMs without administrator intervention.

There are two alternative approaches for implementing dynamic methods according to who initiates the process:

- The privileged domain or corresponding self-defined software entity periodically gathers co-residency information and sends it to 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 destruction, VM live migration into and out of a host, VM failure, etc.

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 co-resident VMs consistently. However, the period between two periodical probing operations needs to be configured properly. If the period is set longer than needed, it would bring delayed co-residency information. If it is too short, it might lead to unnecessary probing and thus consumes undesirable CPU cycles. The second approach is event-driven and synchronous. When a VM migrates out/in, the VM is expected to notify related VMs and to update the co-residency information. Thus, the updates are immediate upon the occurrence of the corresponding events. In comparison, the first approach periodically collects the status from co-resident VMs and thus introduces delayed update and some level of inconsistency. Even with the second approach, it is possible that co-residency information of multiple VMs changes concurrently. Thus, the consistency of the VM co-residency information should be maintained.

2) Transparent Switch between Local and Remote mode

To enable transparent VM live migration, we need not only to provide both local and remote mode of inter-VM communication channels, but also to support transparent

switch between the two types of inter-VM connections. Two tasks are involved in performing the switch: (i) to verify if the communicating VMs are co-resident, (ii) to find the proper point where and when to perform the transparent switch automatically. For the first task, the unique identity of every VM and the co-residency information are needed. [Dom ID, IP/Mac address] pairs can be used to uniquely identify domains. The VM co-residency membership is dynamically updated by automatic detection mechanism of co-resident VMs. As for the second task, the approach is to intercept the requests before setting up or tearing down connections or before the sender VM transfers the data.

E. Multilevel Transparency

By multilevel transparency, we refer to three levels of transparency: user level transparency, OS kernel transparency and VMM transparency.

User level transparency refers to a key design choice regarding whether applications can take advantages of the residency-aware optimization without any modifications to the existing applications and user libraries. With user level transparency, legacy applications do not need to be modified to use the shared memory based local channel. User level transparency is usually one of the preferable end goals for software development, since it makes program development and management easier and simpler. To achieve this level of transparency, it is better to implement the inter co-resident VM communication in a layer lower than layer 1 such that no modification is made to the API and user libraries.

By OS kernel transparency, we mean that there is no modification to either host OS kernel or guest OS kernel, so that no kernel recompilation and re-linking are needed. With this feature, no customized OS kernel and kernel patches, and so forth, 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 to implement shared memory based inter-VM communication mechanism since kernel modules have dual advantages: they are compiled separately from OS kernel and thus recompiling and re-linking the kernel can be avoided, and they can be loaded at runtime without requiring system reboot.

VMM Transparency means no modification to VMM since modifying VMM is relatively more difficult and error prone than modifying OS kernel. To keep the stability of VMM and to maintain the independence between VMM and the guest OS instances, it is desirable to only use interfaces exported by the VMM, with VMM's code untouched.

IV. IMPLEMENTATION TECHNIQUES: A COMPARISON

In this section, we make comprehensive analysis and comparison in terms of the choice of implementation layer in software stack, fundamental functionalities, seamless agility, and multilevel transparency. Fig. 3 compares the architectural layout of existing representative approaches.

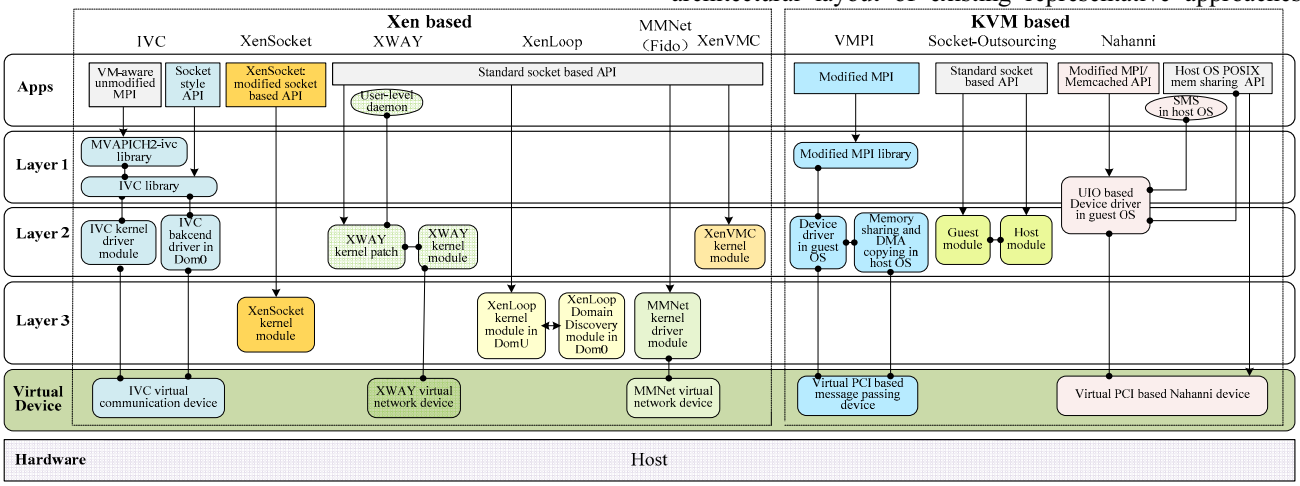

Figure 3. Architectural layout of co-resident VM communication mechanisms

A. User Libraries and System Calls Layer

We briefly describe three existing approaches implemented in this layer: IVC, VMPI and Nahanni.

IVC is designed for cluster-based HPC environment. It establishes the shared memory channel 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: (i) a user space VM-aware communication IVC library, supporting shared memory based fast communication between co-resident VMs, which provides socket style interfaces, (ii) a user space MVAPICH2-ivc library, which is derived from MVAPICH2, an MPI library over Infiniband, and (iii) a kernel driver, which 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. Evaluation demonstrates that in an environment with multi-core systems and Infiniband interconnects, IVC achieves comparable performance with native platforms.

VMPI is an Inter-VM 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. Both OS kernel and the hypervisor are extended and modified to implement VMPI. Experimental results show that VMPI achieves near native performance in terms of MPI latency and bandwidth. VMPI only supports a small subset of MPI API. Its scalability is limited since it does not support a varying number of co-resident VMs to communicate.

Nahanni provides inter co-resident VM shared memory API and commands for both host-to-guest and guest-to-guest communication on KVM platform. It is designed and implemented mainly in layer 1 and its interfaces are visible to user space 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 (User space I/O) device driver model. The shared memory region is allocated by host POSIX operations. It is mapped to QEMU process address space. The mapped memory can be used by guest applications by mapping it again to guest user space. Shared-Memory Server (SMS), a standalone host process running outside of QEMU is designed and implemented to enable inter-VM notification. Evaluation results shows that applications or benchmarks achieve better performance with Nahanni support. However, to take advantage of Nahanni, it is required to rewrite applications/libraries, to modify/extend existing applications/libraries. Nahanni does not support transparent switches between local and remote mode. VM migration is not fully supported.

B. Below System Calls Layer & Above Transport Layer

XWAY, XenVMC and Socket-outsourcing are three existing approaches implemented in this layer.

XWAY makes efforts to abstract all socket options and keeps user level transparency. XWAY modifies the OS kernel by patching it and intercepts TCP socket calls below system calls layer and above transport layer. Hierarchically, XWAY consists of three layers: switch, protocol and device driver. They are implemented as a few lines of kernel patch and a loadable kernel module. At the very first packet delivery attempt, the switch layer is used to determine if the receiver is co-resident or not. Then between TCP socket and local XWAY protocol, it transparently chooses which lower layer protocol should be called whenever a message is transmitted. The protocol layer conducts the tasks of data transmission via the device driver. The device driver plays 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 layers when necessary. Evaluation results show that under various workloads XWAY achieves better performance that native TCP socket by bypassing the long TCP/IP network stack and providing direct shared memory based channel for co-resident VMs.

XenVMC is a fast residency-aware inter-VM communication protocol with 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 in layer 2. XenVMC kernel module contains six sub modules: (i) Connection Manager is responsible for establishing or tearing down local connections between two VMs, (ii) Data Transfer Manager is responsible for data sending and receiving, (iii) Event Manager handles data transmission related notifications between the sender and the receiver, (iv) System Call Analyzer intercepts related system calls and analyzes them, if co-resident VMs are identified, it bypasses traditional TCP/IP paths, (v) VM State Publisher is responsible for announcement of VM co-residency membership modification to related guest VMs, (vi) Live Migration Assistant supports transparent switch between local and remote mode communication 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.

Socket-outsourcing enables inter co-resident VM communication by bypassing the network protocol stack in guest OSes. It 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. Socket-outsourcing supports standard socket API. It is user transparent. However, the VMM is extended to provide shared memory region between co-resident VMs, event queues for asynchronous notification between host module and guest module, as well as VM Remote Procedure Call (VRPC). The user level host module acts as a VRPC server for the guest module and 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. Using 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. No live migration support is provided.

C. Below IP Layer

We also describe three existing approaches in this layer: XenSocket, XenLoop and MMNet.

XenSocket provides a shared memory based one way co-resident channel between two VMs and bypasses the TCP/IP network stack when the communication is local. Most of its code is in layer 2 and is compiled into a kernel module. It is not binary compatible with existing applications. In XenSocket, there are two types of memory pages shared by the communicating VM peers: the descriptor page, used for control information storage, and the buffer pages, used for data transmission. They work together to form a circular buffer. When a connection is be established, 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 buffer and the receiver reads data from it. The connection is torn down from the sender side after data transfer to ensure the shared resources be 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. XenSocket does not support automatic detection of co-resident VMs and transparent switches between local and remote mode.

XenLoop provides fast inter-VM shared memory channels for co-resident VMs based on Xen memory sharing facilities. It keeps the feature of multilevel transparency. To utilize netfilter, 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 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. Every VM is uniquely identified by [guest-ID, MAC address] pairs. The bidirectional inter-VM 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 VMs. 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. Evaluations demonstrate that XenLoop increases bandwidth by up to a factor of 6 and reduces the latency by up to a factor of 5 compared with frontend-backend mode.

MMNet works together with Fido framework to provide shared memory based inter-VM communication optimization for co-resident VMs on Xen platform. Fido offers three fundamental facilities: a shared memory mapping mechanism, a signaling mechanism for inter-VM synchronization and a connection handling mechanism. Fido 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 security. Actually, it is designed for communicating between VMs that are trustable to each other, where the mapping of guest OSes' memory is acceptable. Built on Fido, MMNet achieves programming 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 the IP layer. MMNet provides near native performance and achieves much better performance than frontend-backend model.

D. Seamless Agility & Multi-level Transparency

Seamless agility refers to VM co-residency membership maintenance, automatic switches between local and remote channels, transparent VM live migration and dynamic live VM deployment support by the shared memory based inter-VM communication mechanism. Seamless agility features of representative approaches are summarized in Table I.

TABLE I. SEAMLESS AGILITY FOR VM LIVE MIGRATION AND VM DYNAMIC DEPLOYMENT SUPPORT

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

Xen based approaches utilize Xen Grant Table and 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 IVC, which modifies the VMM to enable VM live migration. In comparison, KVM based approaches, such as Nahanni, VMPI and Socket-outsourcing are all not VMM transparent. The existing support from QEMU/KVM for host-guest and guest-guest memory sharing is not sufficient. Thus current QEMU/KVM is modified or extended to provide such supports. Among all the representative related work, only XenLoop, MMNet and XenVMC keep the features of multilevel transparency. The multilevel transparency features are summarized in Table II.

TABLE II. MULTILEVEL TRANSPARENCY FEATURES

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

V. ADVANCED FEATURES

Most of the existing development work to date pays little attentions to advanced features such as reliability, security and stability. We argue that the next generation of a shared memory based fast communication mechanism for co-resident VMs should give full consideration on how to provide such advanced features.

Reliability. The shared memory based inter-VM communication for co-resident VMs should have the ability to perform and maintain expected functionalities in the case of failures. Thus mechanisms ensuring reliability, such as connection handling upon VM failures, pending data processing when VM migrates in/out, should be offered.

Security. A critical design issue for any mechanism that supports an external process or VM to access memory of another VM is security. Since information leakage is a severe concern for memory sharing between un-trusted VMs, a VM should not be able to access another VM's memory without permission unless the communicating VMs have mutual trust. Thus special care must be taken to ensure desirable degree of isolation among communicating VM peers.

Stability. From our experimental observations on existing open source shared memory based co-resident VM communication mechanism, we find that throughput or latency of data interchange turns unstable under certain circumstances and the mechanisms even fails to work in some boundary conditions [15]. Therefore, special concerns should be paid during the design of the shared memory based inter-VM communication mechanisms for co-resident VMs to ensure the stability. No matter whether the network protocol is TCP or UDP, the size of messages is extremely small or large, the arriving frequency of messages is normal or badly high, the number of co-resident VMs is large scale or not, the performance is expected to be reasonably stable and the system is supposed to operate normally.

VI. CONCLUSION

This paper makes three unique contributions. First, we present the core design guidelines and key issues for optimizing inter-VM communication using shared memory based mechanisms, including the choices of implementation layer in the software stack, seamless agility for VM live migration and VM dynamic deployment support, multilevel transparency. Second, we conduct a comprehensive analysis of representative state-of-the-art research efforts and implementation techniques based on the core design guidelines. Third, we also give a prospect of further requirements in advanced features such as reliability, security and stability. The research results reported in this paper not only serves as a comprehensive reference for developing the next generation of inter-VM communication optimization mechanisms, but also offers both cloud infrastructure providers and cloud service consumers an opportunity to further improve inter-VM communication efficiency in virtualized platforms.

ACKNOWLEDGMENT

The first author's research is partially supported by grants from National Advanced Technology Research and Development Program under grant NO. 2011AA01A203, National Nature Science Foundation of China (NSFC) under grant NO. 60633050 and 61103015, Young Excellent Teacher Researching and Training Abroad Program of China Scholarship Council (CSC). The second and third authors are partially supported by USA NSF CISE NetSE program and CrossCutting program, an IBM faculty award and a grant from Intel ISTC on Cloud Computing.

REFERENCES

[1] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao. "Who is your neighbor: net I/O performance interference in virtualized clouds," IEEE Transactions on Services Computing. IEEE Computer Society, Los Alamitos, CA, USA. Jan, 2012.

[2] W. Huang, M. Koop, Q. Gao, and D.K. Panda. "Virtual machine aware communication libraries for high performance computing", Proc. of the 2007 ACM/IEEE conference on Supercomputing (SC '07). ACM New York, NY, USA. Article No. 9. 2007.

[3] X. Zhang, S. McIntosh, P. Rohatgi, J. L. Griffin. "XenSocket: A high-throughput interdomain transport for virtual machines," Proc. of the ACM/IFIP/USENIX 2007 International Conference on Middleware (Middleware '07). Springer-Verlag New York, Inc. New York, NY, USA.pp. 184-203, 2007

[4] K. Kim, C. Kim, S. Jung, H. Shin, and J. Kim. "Inter-domain socket communications supporting high performance and full binary compatibility on Xen,"Proc. of the fourth ACM SIGPLAN/SIGOPS international conference on virtual execution environments (VEE '08). ACM New York, NY, USA. pp. 11-20, 2008.

[5] J. Wang, K. Wright, and K. Gopalan. "XenLoop: A transparent high performance inter-VM network loopback," Proc. of the 17th International Symposium on High Performance Distributed Computing (HPDC '08). ACM New York, NY, USA. pp. 109-118, 2008.

[6] P. Radhakrishnan, and K. Srinivasan. "MMNet: an efficient inter-vm communication mechanism," Proc. of Xen Summit. Boston, June 2008.

[7] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson, "Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances",Proc. of the 2009 conference on USENIX Annual technical conference.USENIX Association Berkeley, CA, USA..pp. 25–25, 2009.

[8] Y. Ren, L. Liu, X. Liu, J. Kong, H. Dai, Q. Wu, and Y. Li,"A Fast and Transparent Communication Protocol for Co-Resident Virtual Machines," Proc. of 8th IEEE International Conference on Collaborative Computing (CollaborateCom2012). October 14 - 17, Pittsburgh, PA, USA. 2012.

[9] F. Diakhaté, M. Perache, R. Namyst, and H. Jourdren, "Efficient shared memory message passing for inter-VM communications," 3rd Workshop on Virtualization in High-Performance Cluster and Grid Computing (VHPC'08), as part of Euro-Par 2008. Springer-Verlag Berlin, Heidelberg. pp. 53-62, 2008.

[10] H. Eiraku, Y. Shinjo, C. Pu, Y. Koh, and K. Kato., "Fast Networking with Socket-Outsourcing in Hosted Virtual Machine Environments," Proc. of ACM symposium on applied computing (SAC '09). ACM New York, NY, USA. pp.310-317, 2009.

[11] A. C. Macdonell, "Shared-Memory Optimizations for Virtual Machines," PhD thesis, Department of Computing Science, University of Alberta, 2011.

[12] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in Xen," Proc. of the annual conference on USENIX '06 annual technical conference (ATEC'06). USENIX Association Berkeley, CA, USA, pp. 15-28, 2006.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization,"Proc. of the nineteenth ACM symposium on Operating systems principles (SOSP '03). ACM New York, NY, USA. pp. 164-177, December 2003.

[14] Netfilter,http://www.netfilter.org/.

[15] Q. Zhang, L. Liu, and Y. Ren, "Co-Resident Inter-VM Communication: Performance Measurement and Analysis", CERCS Technical Report, Georgia Institute of Technology , USA, Feb 2013.