A Fast and Transparent Communication Protocol for Co-Resident Virtual Machines

Yi Ren[1], Ling Liu[2], Xiaojian Liu[1], Jinzhu Kong[1], Huadong Dai[1], Qingbo Wu[1], Yuan Li[1]

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

[2]College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
[2]lingliu@cc.gatech.edu

Abstract—*Network I/O workloads are dominating in most of the Cloud data centers today. One way to improve inter-VM communication efficiency is to support co-resident VM communication using a faster communication protocol than the traditional TCP/IP commonly used for inter-VM communications regardless whether VMs are located on the same physical host or different physical hosts. Although several co-resident VM communication mechanisms are proposed to reduce the unnecessary long path through the TCP/IP network stack, to avoid communication via Dom0, and to reduce invocation of multiple hypercalls when co-resident inter-VM communication is concerned. Most state of the art shared memory based approaches focus on performance, with programming transparency and live migration support considered. However, few of them provides performance, live migration support, user-kernel-hypervisor transparency at the same time. In this paper, we argue that all three above aspects are fundamental requirements for providing fast and highly transparent co-resident VM communication. We classify existing methods into three categories by their implementation layer in software stack: 1) user libraries and system calls layer, 2) below socket layer and above transport layer, 3) below IP layer. We argue that the choice of implementation layer has significant impact on both transparency and performance, even for live migration support. We present our design and implementation of XenVMC, a fast and transparent residency-aware inter-VM communication protocol with VM live migration support. XenVMC is implemented in layer 2. It supports live migration via automatic co-resident VM detection and transparent system call interception mechanisms, with multilevel transparency guaranteed. Our initial 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.*

Keywords—inter-VM communication; shared memory; high performance; programming transparency; kernel transparency; live migration support

I. INTRODUCTION

Virtual machine monitor (VMM or hypervisor) technology enables hosting multiple guest virtual machines (VMs) on a single physical machine, while allowing each of the VMs running its own operating system. VMM is the software entity that runs at the highest system privilege level and coordinates with a trusted VM, called Domain 0 (Dom0), to enforce isolation across VMs residing on a single physical machine, while enabling each VM running under a guest domain ((DomU) with its own operating system. 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, lower ownership costs, to name a few. To date, VMM based solutions have been widely adopted in data centers, industry computing platforms and academic research and education infrastructures, including high performance computing (HPC) community [1-3].

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 host to share resources through time slicing and space slicing. Second, VMM technology introduces host-neutral abstraction by design, which treats all VMs as independent computing nodes regardless of where these VMs are located.

Although both design choices have gained some advantages, they also carry some performance penalties. First, VMM offers significant advantages over native machines when VMs co-located on the same physical host are non-competing in terms of computing and communication resources. But the performance of these VMs is significantly degraded compared to the performance of native machine when co-located VMs are competing for resources under high workload demands due to high overheads of switches and events in Dom0 and VMM [4]. Second, the communication overhead between VMs co-located on a single physical machine 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 host or from the VMs located on a different physical host. Several research projects [5-9] have demonstrated that. Linux guest domain shows lower network performance than native Linux [10], when an application running on a VM communicates with another VM. [6] showed that with copying mode in Xen 3.1, the inter-VM communication performance is enhanced to 1606Mbps but still significantly lagging behind compared to the performance on native Linux, especially for VMs residing on the same physical host. We observe from our experimental observation on Xen that long path through the TCP/IP network stack, inter-VM communication via Dom0, invocation of multiple hypercalls all contribute to the performance degradation of inter-VM communication. First, TCP/IP based network communication is not necessary when co-resident VMs communicate with one another, since TCP/IP was originally designed for inter physical machine communication via LAN/Internet. For instance, on Xen hypervisor enabled platforms, when the sender VM transfers data to the receiver VM residing on the same physical host, the packets have to unnecessarily go through the entire TCP/IP network stack. Furthermore, all the transmissions are redirected via Dom0, which invokes the page flipping mechanism through multiple hypercalls.

In order to improve the performance of network intensive applications running in virtualized computing environments, such as Web based systems, online transaction processing, distributed and parallel computing systems, we need to explore two levels of optimization opportunities. At lower level we need a fast and yet transparent communication protocol for co-resident VMs, and at the higher level, we need efficient co-location of applications running on co-resident VMs based on their resource consumption patterns to alleviate the resource competition among co-resident VMs. In this paper, we focus on techniques for the first type of performance optimization. Most of existing inter-VM communication optimization techniques argue that TCP/IP based network communication is not necessary when co-resident VMs communicate with each other and they differ from one another in at least two aspects [5-9]: First, different methods are proposed for implementing an alternative communication mechanism for co-resident VMs instead of TCP/IP to minimize the network latency. Second, different proposals make different degree of guarantee in terms of keeping the features of existing hypervisor, such as live migration support and the transparency of the proposed mechanism over existing software layers [11].

In this paper, we argue that the design of inter-VM communication mechanism for co-resident VMs to replace TCP/IP based network virtualization should be completely transparent to applications running on VMs in terms of both performance and operational interface. Concretely, a high performance and high transparency communication protocol for co-resident VMs should support the following three essential functional requirements: First, the co-resident communication protocol should be capable of distinguishing co-resident VMs from VMs located on separate physical hosts such that the communication path between co-resident VMs can be shortened to reduce or minimize the unnecessary communication overhead. Second, the co-resident VM communication protocol should provide seamless support of automatic switch between co-resident inter-VM communication (or local mode) and remote inter-VM communication (or remote mode) where sender VM and receiver VM are residing on different physical machines. This ensures that the important features of system virtualization, such as VM live migration, are retained. Third but not the least, the co-resident VM communication protocol should also maintain the same uniform VM interface for applications by keeping the transparency of the proposed communication mechanism over programming languages, OS kernel and VMM, such that legacy applications are supported in a seamless fashion and there is no need to make code modifications, recompilation, or re-linking.

Existing methods based on Xen platform for reducing the performance overhead of co-resident VMs introduced by TCP/IP network stack are done primarily by shortening the long communication path through shared memory facilities [5-9]. Most of them demonstrate their optimization techniques by exploiting the generic facilities of grant tables and event channels provided by Xen hypervisor to show the throughput improvement of co-resident VM communications. We classify these approaches into three categories based on the different layers of the software stack where the implementation is exercised.

The first category of approaches implements the co-resident VM communication in the user library and system calls level. The most representative approach is IVC [7], which was specially designed for parallel high performance computing (HPC) protocols. IVC supports VM live migration. But it introduced a VM-aware MPI library, which is not programming transparent to application developers.

The second category of methods implements the co-resident VM communication in the software stack below socket layer and above transport layer, represented by XenSocket [5] and XWAY [6]. XenSocket proposed a new socket protocol such that the applications have to be rewritten to run on XenSocket, though it is more efficient for co-resident VM communication. XWAY is programming transparent. But its components are scattered across different parts of the operating system kernel. The guest operating system code needs to be modified, recompiled and re-linked, and thus XWAY offers better application transparency but less kernel transparency.

The third category of approaches is represented by MMNet [9] and XenLoop [8] and is implemented below the IP layer. These methods are implemented as self-contained kernel modules and are both application programming and operating system kernel transparent. Concretely, MMNet maps the kernel address space of one VM into the address space of its peer VM and relaxes memory isolation between VMs. XenLoop uses netfilter [12] hooks to intercept outgoing network packets below IP layer and supports fully transparent VM migration. Although implementing co-resident VMs at below IP layer could also facilitate the goal of achieving user-level, kernel-level and hypervisor-level transparency, it suffers from processing overhead at lower protocol layer [8].

In summary, for mechanisms implemented in layer 1, it is difficult to ensure its user level transparency since libraries and programming interfaces are usually modified. For below IP layer implementation, user level transparency is maintained, but the performance is potentially suffered due to lower protocol processing overheads experienced in each network I/O. To achieve both high transparency and high performance, layer 2 (below socket layer and above transport layer) is preferred. However, existing methods proposed in this layer either fails to provide user-level programming transparency or suffers from kernel level transparency by requiring modifying guest OS code, recompiling and re-linking guest OSs.

In this paper, we argue that residency-aware inter-VM communication mechanism should provide high performance and good user-kernel-hypervisor transparency at the same time. We first analyze and compare all three categories of existing shared memory based co-resident VM communication approaches and discuss key issues to be considered for designing a high performance and high user-kernel-hypervisor transparency solution for co-resident VM communication. Then we present our design of a transparent residency-aware fast inter-VM communication mechanism through XenVMC, the prototype implementation of our current solution. XenVMC is implemented below socket layer and above transport layer. It achieves transparency in VMM, operating system kernel and user programming level and fully supports VM live migration. Our initial experimental results show that our co-resident communication protocol achieves better throughput and less latency than TCP/IP based mechanisms. Furthermore, it offers better peak performance over existing solutions.

The rest of this paper is organized as follows. Section II introduces relevant background and terminology of VMM and its shared memory structures. Section III discusses several key issues for design and implementation of shared memory based co-resident VM communication mechanisms. Section IV presents our approach XenVMC, including design philosophy, implementation and evaluation. Section V concludes the paper.

II. XEN NETWORK ARCHITECTURE AND INTERFACES

This section gives a brief description of Xen network I/O architecture and its VM communication mechanism.

A. *Network I/O architecture*

Xen is a popular 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 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 (DomU). It is also allowed to access the control interfaces of the hypervisor.

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 typically is Dom0 or is a driver specific VM. The IDD hosts a backend network driver. And 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.

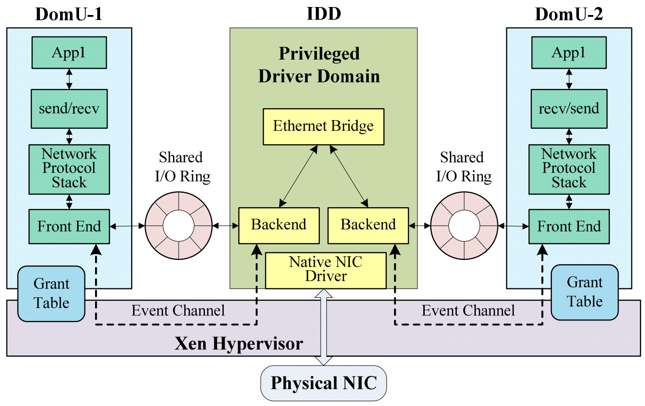

Figure 1. Xen network I/O architecture and interfaces

B. *Communication mechanisms among domains*

As shown in Figure 1, the shared I/O Ring buffers between the frontend and the backend are built upon grant table and event channel mechanisms provided by Xen. Grant table works as a generic mechanism to share pages of data between domains, which support both page mapping mode and page transfer mode. 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). XenStore is used by some of the existing projects as a basic mechanism to facilitate the tracing of dynamic membership changing of co-residency VMs. It 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. Dom0 can access the entire path, while each DomUs can access only its owned directories.

III. CO-RESIDENT VM COMMUNICATION: DESIGN CHOICES

A. *Why shared memory based approaches*

It is observed that with TCP/IP based network, the data will have to go through along the TCP/IP protocol stack. With Xen hypervisor involved, multiple switches between VM and VMM are incurred during the transfer. All these can lead to performance degradation. By bypassing TCP/IP and introducing shared memory based approaches for co-resident VM communication, we can reduce the number of data copies and avoid unnecessary switches between VM and VMM. Another advantage of using shared memory is the reduced dependency to VMM as the shared memory based communication among co-resident VMs bypasses the default TCP/IP network path, thus less hypercalls are used and is less dependent on Dom0, which is originally a potential performance bottle neck with TCP/IP network. 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. *Design Guidelines and Classification*

As mentioned in the Introduction section, there are three criteria for designing an efficient shared memory based residency-aware inter-VM communication mechanism. They are high performance of virtualized network, seamless agility for supporting VM live migration, multilevel transparency (user-kernel-hypervisor). We will discuss why these criteria should be met and the possible approaches to achieve these objectives. Given that Xen and KVM (or other similar VMM) differ to some extent in their concrete virtualization architecture design, we below mainly discuss the approaches and related work that are conducted based on Xen platforms.

The ultimate goal of incorporating a fast co-resident VM communication mechanism into the TCP/IP based inter-VM communication protocol is two folds. First, when the sender VM and the receiver VM are co-resident on the same host, the data will bypass the long path of TCP/IP network stack and be transmitted via the shared memory (local mode); otherwise the data will be transferred through traditional TCP/IP network (remote mode). To achieve this capability, we need to be able to intercept every outgoing data requests, examine and detect whether the receiver VM is co-located with the sender VM on the same host, and if so, redirect the outgoing data request to a co-resident VM communication protocol instead. The switching between local and remote modes of inter-VM communications should be carried out automatically and transparently.

Existing approaches differ from one another in their choice of which layer in the software stack they propose to implement the above interception, detection and switching mechanisms. Thus we classify the existing work into three categories based on their implementation layer in the software stack: (i) user libraries and system calls layer (or layer 1), (ii) below socket 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.

a) User libraries and system calls layer

One way to implement interception, detection and redirection tasks is to modify the standard user and programming interfaces in the user libraries and system calls layer [7]. This category of approaches is often used in the HPC environment where MPI (Message Passing Interface) based communication dominates. Although this approach is simple and straightforward, it fails to maintain the programming transparency.

b) Below socket layer and above transport layer

An alternative approach to implementing an efficient co-resident VM communication is to perform interception, detection and redirection tasks below the socket layer and above the transport layer [5-6]. There are several reasons of why this alternative layer 2 solution may be more attractive. 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. Furthermore, when the encapsulated data reaches the receiver node, the headers will be removed layer by layer. However, if the data is intercepted and redirected in a higher layer, such as below socket layer and above transport layer, 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 believing that implementation in upper layer of TCP/IP network stack can potentially lead to lower latency and higher throughput of network I/O workloads.

c) Below IP layer

In addition to Layer 1 and Layer 2 approaches, another alternative method is to implement the interception, detection and redirection tasks below the IP layer [8-9]. MMNet and XenLoop are both implemented below IP layer. MMNet 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. It achieves better performance in the cost of relaxed memory isolation between VMs. Evaluations demonstrate that XenLoop reduces the inter-VM round trip latency by up to a factor of 5 and increases bandwidth by up to a factor of 6. XenLoop not only improves the performance, but also satisfies both the transparency and live migration support criteria. To intercept outgoing data, hooks in the data process chain are needed. XenLoop was implemented on top of netfilter, a third party tool, which already exports such hooks and provides development interface below IP layer.

In summary, implementing interception, examination and redirection data in higher layer of the protocol stack could potentially provide much better inter-VM communication performance due to low protocol processing overheads [8].

C. Seamless Agility for VM live migration support

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 in an automatic and adaptive fashion, in the presence of both VM live migration and on-demand addition or reduction of co-resident VMs.

a) Automatic co-resident VM detection

When inter-VM channels are to be set up, VMs need to identify whether the other peer is on the same physical host. Similarly, when the channels are to be torn down, VMs also need to know if the other peer is still there.

There are two methods to maintain VM co-residency information. One is static method, which is primarily used when the membership of co-resident VMs is preconfigured or collected by the administrator and is not supposed to change during network operations afterwards [6-7]. The second method is a dynamic one, which is usually implemented as automatic co-resident VM detection mechanisms. In the absence of dynamic method, VM live migration cannot be supported, since co-residency information is unpredictable because of the existence of VM live migration.

Automatic co-resident VM detection mechanism is expected to gather and update co-resident information in an efficient and precise way. There are two alternative approaches according to who initiates the process: (i) Dom0 periodically gather co-residency information and transmit it to VMs on the same host, (ii) VM peers advertize its presence/absence upon significant events, such as VM creation, migration in/out, destruction, etc. The two approaches have their advantages and disadvantages which we will discuss in Section IV.

b) Transparent switch between local and remote mode

To support automatic switch between local and remote mode, there are two issues to be considered:

- To identify whether the communication VMs are residing on the same physical machine or not.
- To find the proper point where and when to identify those information.

For the first issue, the unique identity of every VM and the co-resident information are needed. [Dom ID, IP/Mac address] pairs can be used to identify unique domains. Maintaining co-resident VMs within one list makes the identification easier. The co-resident membership information is dynamically updated by automatic co-resident VM detection mechanism.

As for the second issue, the approach is to intercept the requests before the setup and tear down of connections or before the transfer of outgoing data. Linux provides netfilter, a hook mechanism inside the Linux kernel. XenLoop uses netfilter to intercept every outgoing network packet below IP layer and inspect its header to determine the next hop node. Since netfilter resides below IP layer, its interception mechanism involves packet headers and longer path compared with alternative approaches below socket layer and above transport layer.

We argue that even though so far, there is no available third party tool below socket layer and above transport layer that provides similar mechanisms as netfilter does. Thus in order to support VM live migration, it is expected one needs to implement an interception mechanism from scratch. And the implementations are is expected to be transparent to user applications and without modification of OS kernel and VMM.

D. *Multilevel transparency*

In this section, we introduce the concept of multilevel transparency to further illustrate the requirements for developing efficient and scalable inter-VM communication mechanisms. By multilevel transparency, we refer to three levels of transparency: user level transparency, OS kernel transparency and VMM transparency.

User level transparency. With user level transparency, legacy network applications using standard TCP/IP interfaces do not need to be modified in order to use shared memory based optimized communication 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.

OS kernel transparency. OS kernel transparency means that there are no modification to either Host OS kernel or guest OS kernel, and no kernel recompilation and re-linking. 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.

VMM transparency. 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 above OS instances, it is desirable to only use interfaces exported by Xen, such as grant table, event channel and XenStore, with Xen hypervisor codes unmodified. We refer to this feature as VMM/hypervisor transparency.

To achieve better performance, multilevel transparency feature is often sacrificed [5-7]. To obtain the feature of OS kernel transparency, one feasible approach is to try to implement proposed inter-VM communication mechanism with non-intrusive and self-contained kernel modules.

IV. OUR APPROACH - XENVMC

In this section, we describe the design and implementation of XenVMC, a shared memory based fast inter-VM communication protocol, implemented below the socket layer and above the transport layer. Our design goal for XenVMC is three folds: high performance, high transparency and high agility. In this section we present the design principles, implementation details and the evaluation of XenVMC.

A. *Design principles*

We present the design principles of XenVMC in terms of functional and non-functional requirements.

The essential functional requirements for XenVMC design are two folds: First, we should support both local and remote inter-VM communication. The local mode of inter-VM communication is provided through a fast co-resident VM communication channel based on shared memory structures, while the TCP/IP protocol is used for the remote inter-VM communication. The second functionality of XenVMC is to support transparent switch between local model and remote mode of inter-VM communications.

To satisfy the first functional requirement, we need the mechanisms to fulfill a series of basic tasks, such as connection setup and tear down, data send and receive, event and message notification between any pair of VMs, no matter the VMs reside on the same physical machine or not. To meet the second functional requirement, we need to implement additional tasks, such as system calls analysis and interception, co-resident VM state information detection and update, live migration support. Both basic and additional tasks form the core components of the XenVMC solution.

The non-functional requirements of XenVMC design are three folds: high performance, high transparency and high agility. By high performance we aim at improving the throughput of inter-VM communication for co-resident VMs. By high transparency, we aim at ensuring multilevel transparency at user, OS kernel and hypervisor level. Namely, no modification to either user libraries or OS kernel and the hypervisor, and all basic as well as additional tasks are implemented as non-intrusive self-contained kernel modules. By high agility, we support dynamic addition or reduction of co-resident VMs.

B. *Implementation Considerations*

We have implemented XenVMC on Xen 3.4.2 with Linux kernel 2.6.18.8 in C. Excluding comment lines, the overall size of XenVMC code is about 2400 lines. XenVMC is compiled into a self-contained kernel module and makes no changes to both user libraries/interfaces and Linux kernel or Xen hypervisor. Figure 2 gives an overview of XenVMC system architecture. For presentation clarity, we only show one Guest OS to avoid redundancy. Multiple guest OSes can be deployed into the system and they have the same structure for inter-VM communication. Each guest OS hosts a non-intrusive self-contained XenVMC kernel module, which is inserted as a thin layer below the socket layer and above the transport layer.

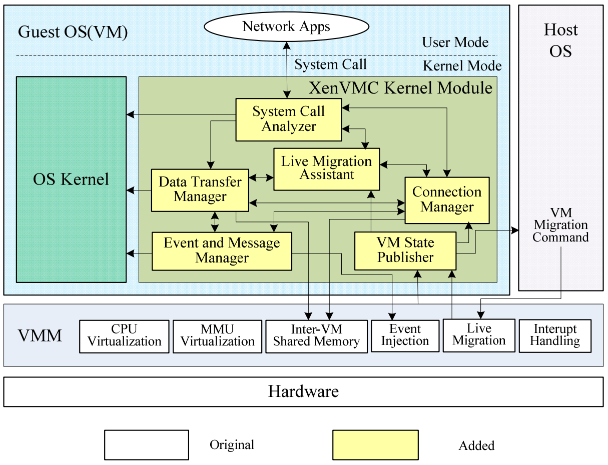

Figure 2. XenVMC architecture

In the subsequent sections we will describe in detail the sub modules of XenVMC kernel module, the automatic collection and update of co-resident VM information, the transparent system call interception, the setting up and tearing down of shared memory connections, the data transmission, as well as the VM live migration support in XenVMC.

1) Sub modules of XenVMC kernel module

XenVMC kernel module contains six sub modules as shown in Figure 2. They are Connection Manager, Data Transfer Manager, Event and Message 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. And it enables seeking local channels with socketfd or with <IP/DomID, port> pair. Data Transfer Manager is responsible for data sending and receiving. It supports both blocking and non-blocking mode via FIFO buffer. Event and Message Manager provides two types of communication mechanism for VMs. The event mechanism is based on Xen event channel. It handles data transmission related notifications between the sender and the receiver. While the message mechanism is implemented to enable notifications among VMs across the boundary of physical machines.

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 VMs upon VM creation, VM migration in/out, VM destruction, XenVMC module unloading/exit, etc. Live Migration Assistant maintains vms[], which will be illustrated later. It supports transparent switch between local and remote mode communication together with other sub modules. For example, it asks Data Transfer Manager to deal with pending data during the switch and sends requests to Connection Manager to establish/close local/remote connections.

2) Automatic VM co-residency information collection and update

To identify whether or not a pair of communicating VMs resides on the same physical machine and to support VM live migration, it is necessary to collect and update VM co-residency information automatically and on demand, without administrative intervention.

As mentioned in Section III, there are two alternative approaches for automatic co-resident VM detection. The first approach needs centralized management by Dom0. 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 period between two periodical probing operations needs to be configured properly. If the period is longer than needed, it would bring delayed co-residency information. However, if it is too short, it might lead to unnecessary probing and thus consumes undesirable CPU cycles. XenLoop employs the first approach. The second approach is event-driven. When a VM migrates out/in, the VM is expected to notify related VMs and update the co-residency information. Without the need to decide how long the probing period should be, the second approach is more precise, since co-residency information updates are immediate when corresponding events occur. But with the second approach, update results are not immediately visible before all the co-resident VMs are reached due to the advertizing/broadcasting latency. Moreover, since it is possible that the co-resident information of VMs on a single host changes concurrently, the consistency of the information should be maintained. We adopt the second approach based on the following observations:

- Events such as VM creation, VM migration in/out, VM destruction, XenVMC module unloading/exit do not happen frequently.
- The number of co-resident VMs is usually not large according to existing hardware constraints.

Based on these observations, we conjecture that the response time of the event driven approach is expected to be more precise and the update on the co-residency information is immediate, without consuming undesirable amount of time for periodical event probing. Given that the total number of co-resident VMs is usually not so large (0-100), the disadvantage of the event-driven approach, such as delayed visibility of update results due to advertising and broadcast latency, can be neglected.

Before illustrating how to gather and update VM co-residency information automatically with the second approach, we first introduce vms[], an array maintained by every guest VM. vms[] stores not only co-residency information, but also data of local channels. As shown in Figure 3, every item in vms[] represents one of the co-resident VM on the physical host (excluding the guest VM itself) uniquely identified by its <Dom ID, IP> pair and points to a hash table. This hash table stores all the communication VM peers of the specific co-resident VM, with each item in the hash table indicates one of such communication VM peers and points to a one-way linked list of local channels between the two peer VMs. Every connection in XenVMC system is represented by struct conn_t, which consists of a list of fields, including lport (local port), rport (remote port), sender_t, recver_t, etc. the struct sender_t contains mainly a pointer to the sender VM, a wait queue of processes waiting to write data, a pointer to allocated shared virtual memory, the offset for data writing, etc. recver_t is similar to sender_t in struct. All these parameters are defined as atomic_t to allow parallel accesses from multi users.

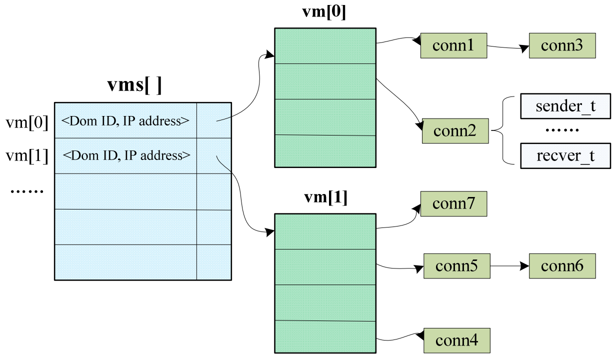

Figure 3. vms[] maintained by every guest VM

Co-resident VMs and their related local channels information are well organized in vms[]. When one guest OS is created or migrated in a new host, vms[] of this guest OS will be initialized. When events such as creation, migration in/out or destruction of other co-resident VM occur, vm[] will be updated. When a guest VM is about to migrant out of current physical machine, vms[] of this VM will be deleted. And the events and corresponding actions are handled in an atomic fashion. When one of above events occurs, all the other VMs on the same physical machine will be notified so that vms[] can be correctly updated. vms[] is maintained by every guest OS instead of Dom0. The reason is that co-residency and local channel information are frequently accessed data and inter-domain access indicates more latency.

3) transparent system call interception

To support transparent switch between local and remote mode, what is most important is to find hook point and how to hook into the communication path. As discussed in section III, for the sake of efficiency, it is desirable to implement the mechanism between socket layer and transport layer in stead of below IP layer. However, Linux does not seem to provide a transparent netfilter-like hook to intercept messages above IP layer. So the problem turns to be how to implement similar hooks based on existing system mechanisms. Since system calls mechanism itself provides a universal entry into kernel for all the requests, it is feasible to transparently intercept every network related system call and analyze it from the context whether the VM peers are on the same physical machine, then bypass TCP/IP path if they are co-resident VMs. We design and implement the interception mechanism on software interrupt based system call mechanism.

The basic design objectives behind the kernel transparent system call interception mechanism are: (i) to intercept network related system calls and (ii) be able to judge local from remote mode and switch between them without user intervention (iii) no modification, no re-compilation and re-linking of kernel codes. And the basic ideas behind the design are: (i) to hook into software interrupt based system call processing path, (ii) to replace existed corresponding system call handlers with self defined system call handlers, (iii) to implement in self-contained XenVMC kernel module.

We set hooks into system by introducing self defined system call handlers, replacing existing handler addresses with those of self defined handlers, and recovering the address when XenVMC module is unloaded from guest OS or when it is identified that its communication VM peer is remote. In order to replace system call handlers with self defined ones, the entries in system call table is to be modified transparently to OS kernel. However, for security reasons, the memory pages in which the system call table is residing are read only. Therefore, we modify the 17th bit in register cr0 to 0 (writable) before the replacement, and recover it when self defined system call handler returns or the communication VMs are located on separate physical machines.

The replacement is done in an action-on-event pattern. As a kernel module, there are three significant “events” for XenVMC: being loaded, initialized or exits (unloaded). The entry of system calls is passed into XenVMC as a parameter when the event “XenVMC is loaded” occurs. And the addresses of the original handlers are saved and replaced with user defined ones in XenVMC_init(). The addresses of original handlers are recovered in XenVMC_exit(). User defined system call handlers are responsible to identify from the context whether current communication is local or not. If it is local, shared memory based approach is activated. Otherwise, it recovers the addresses of original handlers.

All the above functionalities are implemented in XenVMC module. There is no modification to either OS kernel or VMM code. And either the switch between original system call handler and user defined handler or that between local and remote mode is transparent to users.

4) Setting up and tearing down share memory based connection

When one of the guest VMs detects the first network traffic to a co-resident VM, it initiates the procedures of local connection set up as the sender. First, it initializes struct conn_t, assigns memory pages as data buffer and writes data into it. Then the receiver initiates itself by initializing recver_t and asks the sender to establish the connection, with dom ID attached. Accordingly, the sender initializes sender_t in a similar way. Then the function gnttab_grant_foreign_access provided by Xen grant table is activated to share assigned memory pages with the receiver. And the sender calls HYPERVISOR_event_channel_op to create event channel. And dom ID of the sender, evtchn_port and references of shared memory pages are sent to the receiver by Event and Message Manager. After receiving these parameters, the receiver binds to the event port of the sender and maps the shared memory pages to its address space and reads data from the shared memory. Then it notifies the sender the connection set up procedure is finished. Figure 4 illustrates the process.

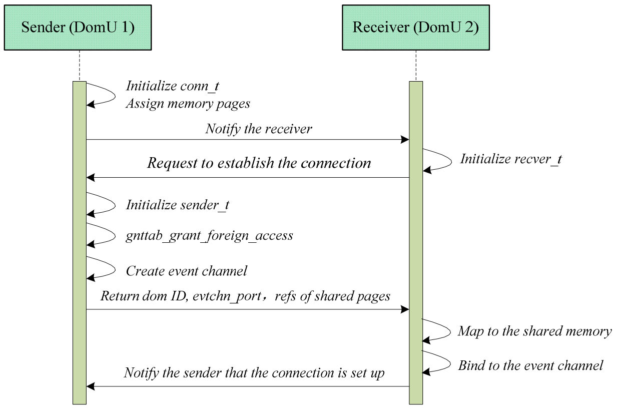
Sender (DomU 1)
Receiver (DomU 2)
Initialize conn_t
Assign memory pages
Notify the receiver
Request to establish the connection
Initialize recver_t
Initialize sender_t
gnttab_grant_foreign_access
Create event channel
Return dom ID, evtchn_port，refs of shared pages
Map to the shared memory
Bind to the event channel
Notify the sender that the connection is set up

Figure 4. Setting up a shared memory based connection

Whenever one guest VM migrates out, suspends, shuts down, destructs, or unloads the XenVMC kernel module, all of its shared memory based connections are expected to be torn down. Additionally, VM State Publisher of its co-resident VMs needs to update co-residency information by refreshing the vms[]. As shown in Figure 5, in order to tear down a connection, if the initiator is the sender, first it needs to stop data sending and notify the receiver to begin the process. Then the receiver stops data receiving. This is guaranteed by obtaining access lock to struct recver_t, which indicates that all the receiver threads have stopped getting data. And the connection is switched to traditional TCP/IP mode if necessary by recovering the addresses of corresponding system call handlers. After that, shared memory pages are unmapped and event channel is closed. Then struct recver_t is freed. After that, the sender is notified. Then the connection is switched to TCP/IP mode if necessary. If there are any unsent data, then transfer the remained data. Finally, event channel is closed by the sender, share memory pages and therefore struct sender_t are freed. After the connection is torn down, struct conn_t is freed and vms[] are updated.

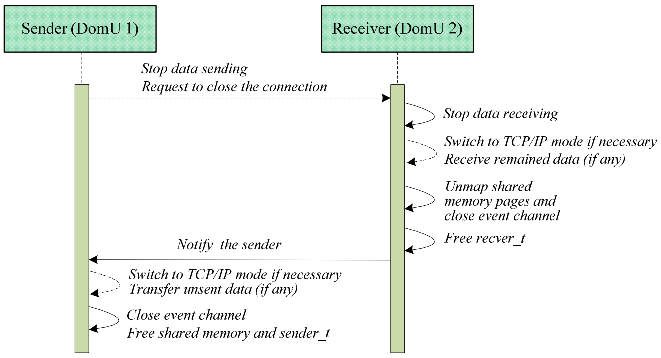
Sender (DomU 1)
Receiver (DomU 2)
Stop data sending
Request to close the connection
Stop data receiving
Switch to TCP/IP mode if necessary
Receive remained data (if any)
Unmap shared
memory pages and
close event channel
Free recver_t
Notify the sender
Switch to TCP/IP mode if necessary
Transfer unsent data (if any)
Close event channel
Free shared memory and sender_t

Figure 5. Tearing down a shared memory based connection

5) *Data transmission*

Once the connection is established, it is possible for VM peers to exchange network traffic. In this section, we first describe the structure of FIFO buffer. Then we introduce steps of VM co-residency identification and local channel lookup. Based on these mechanisms, we illustrate how to handle data sending and receiving in XenVMC system.

a) *FIFO buffer*

The buffer between sender VM and receiver VM for local data transfer is structured as a FIFO circular queue, as illustrated in Figure 6. It is built based on Xen grant table mechanism. Given any guest VM, its sending FIFO buffer and receiving one are separated since the VM can be either the data producer or the consumer. The producer writes data into the buffer and the consumer reads data from it in an asynchronous way. Xen event channel is used for them to notify each other when data is ready or the buffer is empty, etc. The transferred data is allowed to vary in size from time to time. The exact length is specified in the head attached before the data. The thread safety is guaranteed by spinlock facilities on struct sender_t and recver_t.

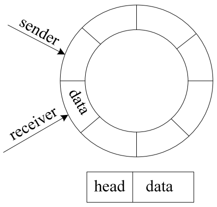

Figure 6. FIFO buffer

b) *Co-residency identification and local channel lookup*

Before data is transferred, if the sender and the receiver are on the same physical machine, it is necessary to look up local connection info via variable vms[] to determine if a new local connection needs to be set up. For example, when a new VM is added to the current host machine, the local channel needs to be established. This is done in two steps. First, Data Transfer Manager obtains <IP, port> pairs of the pair of communicating VMs from input parameters. Then it tries to find the VM peer in local vms[] via the IP address of the other VM. If there is no match, then it switches to TCP/IP mode since the two communicating VMs are not on the same physical machine. Otherwise, there is a matched VM, which indicates the two VMs are on the same host. With IP addresses, it is easy to locate the information of a VM in the specific hash table and to find out if the local connection between a pair of VMs has been established via lport and rport. Note that upon addition of a new VM to a physical host machine, this VM will finally be added to vms[] of all the other co-resident VMs.

c) *Data sending and receiving*

When system call for sending or receiving data is intercepted by the XenVMC module, the original handler is replaced with XenVMX self defined system call handler. Whether it is data sending or receiving, the first few steps in self defined system call handler is similar. First, it identifies if the two VMs are on the same physical machine by looking up the matching VM in vms[]. If the two VMs are not co-resident, then it recovers the original system call handler and switches to traditional TCP/IP communication. Otherwise, it determines if local channel, implemented as the shared memory based connection between the two VMs, already exists. The remaining processing steps are different for data sending and receiving.

Data sending. For data sending, if no corresponding local channel exists in vms[], then Data Transfer Manager calls Connection Manager to initialize struct conn_t and insert it into vms[]. If unused space in FIFO buffer is enough for data to be sent, then it copies the data into FIFO buffer and notifies the receiver via event channel. Otherwise, for blocking I/O mode, the sender waits for the receiver to copy data from the buffer until the receiver notifies the sender. Struct sender_t is updated to mark the available space in FIFO buffer. If one of two locally connected VMs is about to migrate during data sending, the communication will be switched to TCP/IP mode.

Data receiving. For data receiving, if no corresponding local channel exists in vms[], then if there are any pending data remained from previous TCP/IP network traffic, the receiver receives those data via TCP/IP network. After finishing pending data handling, struct conn_t is initialized and inserted into vms[]. Then the receiver identifies if there is enough data ready in FIFO buffer for reading. If yes, it copies data from the buffer and notifies the sender that data has been received. Otherwise, for blocking I/O mode, the receiver waits for the sender to copy data into the buffer before reading; for non-blocking I/O mode, it copies data from the buffer. Struct recver_t is updated to record data received. When VM live migration is about to happen, the communication mode will be switched to remote mode.

6) *Live migration support*

XenVMC kernel module receives a callback from the Xen hypervisor when one of the two co-resident VM peers is going to migrate. By design, XenVMC handles the live migration transparently. If two VM peers originally are co-resident, then Live Migration Assistant stops them from data sending and receiving via FIFO buffer, saves the pending data, and tears down the current local channel. Meanwhile, the original system call handler is reinstalled so that TCP/IP mode of inter-VM communication will be used after the migration. Also the VM about to migrate out will delete its own vms[] and send broadcasts to its co-resident VMs, who will update co-residency and local channel information by modifying the value of their own vms[]. Once the VM is migrated out, TCP/IP network is used to send pending data and traffic afterwards. If two VMs originally are on separate physical machines and become co-resident after the migration, then the automatic co-resident VM detection mechanism on the destination host will discover the newly added VM , and this new VM addition will be broadcasted and local channel is established. vms[] of this VM and vms[] of its co-resident VMs are reconstructed or updated. The entire process of live migration is transparent to user applications, without modifying Linux kernel and Xen hypervisor.

C. Evaluation

XenVMC improves the performance of traditional TCP/IP based frontend-backend mode of inter-VM communication for co-resident VMs. A typical TCP/IP communication goes through long processing path, triggers more switches between Xen hypervisor and guest OSes, and more data copy operations. With XenVMC, the performance of inter-VM communication is improved by enabling co-resident VMs to communicate via fast shared memory channels, which use shorter communication path, require less context switches and data copy operations. In this section, we present some initial experimental results for evaluating the performance of XenVMC approach and comparing it with the traditional TCP/IP network based frontend-backend approach.

All experiments were conducted on a machine with Intel(R) Core(TM) i5 750 @ 2.67GHz processor and 4GB memory. All the data reported was run on Xen 3.4.2 and Linux 2.6.18.8. We run each test for 7 times and report the average, with the highest and the lowest values removed before computing the average. We configured two CentOS 5.5 guest VMs on the above test machine with one vCPU and 512M memory assigned for each VM. We use Netperf 2.4.5 as performance benchmark. The comparison is primarily done over the following two scenarios:

- XenVMC: Inter Guest VM communication via mechanisms provided by XenVMC kernel module.
- TCP frontend-backend: Inter Guest VM communication via traditional virtualized frontend-backend network data path.

We have compared and analyzed the experimental results of above two approaches in terms of both network throughput and network latency.

1) Network throughput

We use Netperf TCP_STREAM test to evaluate the throughput of XenVMC and TCP frontend-backend respectively. When a connection is established between a netserver and a netperf client, the netperf client begins to send data of specified size to the netserver continuously in 10 seconds duration. Figure 7 shows the throughput measurement with varying size of messages.

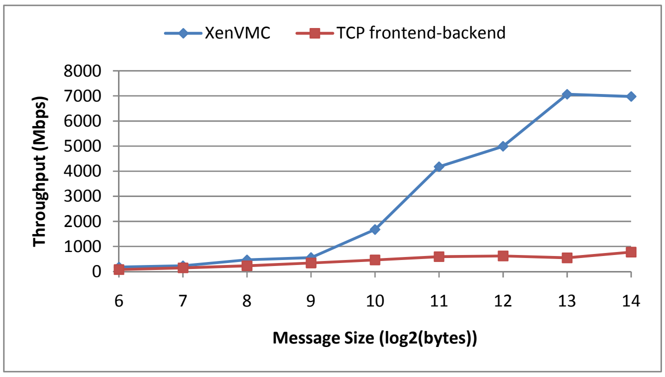

Figure 7. Throuput comparison of XenVMC vs. TCP frontend-backend

The experiments show that XenVMC achieves better network throughput over TCP frontend-backend when message size exceeds 512 bytes. Compared to TCP frontend-backend approach, XenVMC improves inter-VM communication throughput by up to a factor of 9 when the message size is larger than 512 bytes. This is because when the same size of data is being transferred, larger message size indicates less control information exchanged between sender VM and receiver VM, and less switches between user space and kernel space.

2) Network latency

We use Netperf TCP_RR test to evaluate network latency of XenVMC and compare the results with that of the TCP frontend-backend approach. Upon establishing the connection between a netserver and a netperf client, the netperf client begins to send requests to the netserver, which sends RESPONSEs back after receiving REQUESTs. Such round trip of communication performs in a repeated manner. Figure 8 shows the measurement of the network latency with varying message sizes.

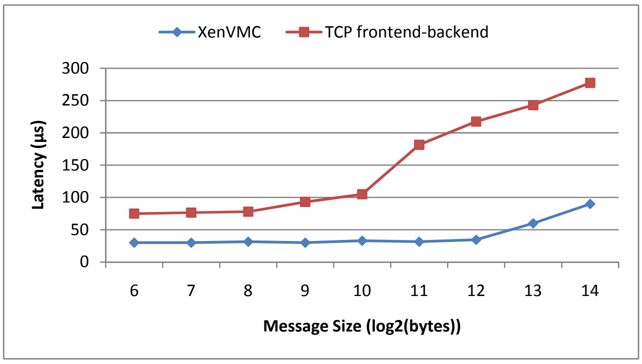

Figure 8. Latency comparison of XenVMC vs. TCP frontend-backend

As shown in Figure 8, XenVMC is more efficient than TCP frontend-backend for all message sizes, and its latency is only 16%-40% of the latency of TCP frontend-backend. Furthermore, when the message size is less than 1024 bytes, the results of both approaches remain relatively stable and the lantecy of TCP frontend-backend is 2.5-3 times of the latency of XenVMC. When the message size exceeds 512 bytes ($\log_2(10)$), the latency of TCP frontend-backend increases obviously, while the latency of XenVMC increases slightly. This set of experimental results show that XenVMC improves the network latency by up to a factor of 6 for varying message sizes.

The initial experimental evaluation shows that XenVMC achieves higher throughput and lower latency as expected, compared with traditional TCP frontend-backend approach. This is because XenVMC spends less time across the network protocol stacks for co-resident VM communication with throughput improvement by up to a factor of 9 and latency reduction by up to a factor of 6.

In contrast to XenLoop [8], which is implemented below IP layer by utilizing netfilter, XenVMC is implemented below socket layer and above transport layer. Thus we conjecture that XenVMC can deliver higher throughput and much improved latency compared with the improvement shown in XenLoop. One of our ongoing efforts is to conduct experimental comparison of XenVMC and XenLoop to better understand the performance impact of implementing shared memory based solution in different software stack through quantitative measures.

V. Conclusion

Traditional TCP/IP frontend-backend virtualization introduces unnecessary overhead for inter-VM communication due to long path through the protocol stack and Dom0-centralized communication mode. One way to address this problem is to use shared memory based approaches to optimize the performance when the VMs are co-resident on the same physical machine. However, implementing the shared memory based inter-VM communication channel at different layer in the software stack may lead to different quality and quantity of performance gains in terms of throughput/latency, user-kernel-hypervisor multilevel transparency and seamless agility.

In this paper, we give an overview of the state of art shared memory based techniques for inter-VM communication on Xen platform. We have also discussed key issues for designing and implementing an efficient residency-aware inter-VM communication mechanism. Based on the analysis, we classify existing shared memory based mechanisms into three categories according to their implementation layer in the software stack. We argue that the implementation layer may have critical impact on transparency, performance, as well as seamless agility in the context of VM additions and live migration support. We present XenVMC, a fast and transparent residency-aware inter-VM communication protocol, implemented below socket layer and above transport layer. We argue that to ensure user level transparency and to avoid unnecessary overhead introduced by lower level protocol processing, it is best to implement co-resident VM communication channel below the socket layer and above the transport layer. This ensures seamless VM live migration via automatic co-resident VM detection, transparent system call interception, and multilevel transparency. Our initial experiment results show that XenVMC improves co-resident inter-VM communication throughput by up to a factor of 9 and reduces the inter-VM communication latency by up to a factor of 6. We have implemented XenVMC to support TCP based inter-VM communication.

Our research continues along several dimensions. For example, we are currently working on experimental comparison of XenVMC with XenLoop to better understand the performance indications when implementing a shared memory co-resident inter-VM connection at different layers in the software stack, especially comparing XenVMC, implemented above IP layer and below socket layer, with XenLook implemented below IP layer. In addition, we are currently extending XenVMC to support UDP based communication as another orthogonal approach to speed up co-resident inter-VM communications.

Acknowledgment

The first author's research is partially supported by grants from National Advanced Technology Research and Development Program under grant NO. 2011AA01A203, National Science and Technology Support Program under grant NO. 2011BAH14B02, National Nature Science Foundation of China (NSFC) under grant NO. 60633050, Young Excellent Teacher Researching and Training Abroad Program of China Scholarship Council (CSC). The second author is 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] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis. “Virtualization for high-performance computing,” ACM SIGOPS Operating Systems Review. ACM, New York, NY, USA. vol. 40, issue 2, April 2006, pp. 8-11.

[2] L. Youseff, R. Wolski, B. Gorda, and C. Krintz. “Evaluating the performance impact of Xen on MPI and process execution for HPC systems,” in VTDC '06 Proceeding of the 2nd International Workshop on Virtualization Technology in Distributed Computing, IEEE Computer Society Washington, DC, USA, 2006.

[3] A. Gavrilovska, S. Kumar, H. Raj, K. Schwan, V. Gupta, R. Nathuji, R. Niranjan, A. Ranadive, and P. Saraiya. “High performance hypervisor architectures: virtualization in HPC systems,” 1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt), in conjunction with EuroSys 2007, Lisbon, Portugal, Mar. 2007, pp. 1–8.

[4] 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.

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

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

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

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

[9] P. Radhakrishnan, and K. Srinivasan. “MMNet: an efficient inter-vm communication mechanism,” in Proceedings of Xen Summit, Boston, June 2008.

[10] A. Menon, A. L. Cox, and W. Zwaenepoel. “Optimizing network virtualization in Xen,” in ATEC '06 Proceedings of the annual conference on USENIX '06 annual technical conference. USENIX Association Berkeley, CA, USA, 2006. pp. 15-28

[11] J. Wang. “Survey of state-of-the-art in inter-VM communication mechanisms,” Research Proficiency Report, available at http://www.cs.binghamton.edu/~jianwang/papers/proficiency.pdf, Sept 2009.

[12] Netfilter. http://www.netfilter.org/.

[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,” in SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles,December 2003. ACM New York, NY, USA. pp. 164-177.