A High-Performance Inter-Domain Data Transferring System for Virtual Machines

Dingding Li, Hai Jin, Yingzhe Shao, Xiaofei Liao, Zongfen Han, Kai Chen
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
Email: hjin@hust.edu.cn

***Abstract*—Virtual machine technology can enhance server utilization and consolidation on an individual physical machine. In a growing number of contexts, many of which require high-performance networking inside of the system. Unfortunately, the development of virtualization technology is inclined to application isolation as yet, leads high communication overheads between virtual machines which are resident on the same physical machine, and also limits the achievable network performance. Some efforts have been paid to bridge the network I/O performance gap between the virtualization and native environment, but the amelioration is not satisfactory. In this paper, we propose a system called IDTS (*Inter-Domain Transferring System*) which uses the shared memory between different domains (also called hosted or guest OSes) to expand the communication bandwidth and shorten the latency for network applications running in the co-resident VMs. Evaluations are also presented to prove IDTS's availability and highly efficiency. We believe that IDTS is an effective solution for the inter-domain communication on Xen virtualization platform.**

***Index Terms*—virtualization, server consolidation, shared memory, latency, bandwidth**

I. INTRODUCTION

Virtualization technology brings the case of sharing the underlying physical machine resources between different virtual machines, each running its own operating system. The software layer providing the virtualization is called a virtual machine monitor or hypervisor. A hypervisor can run on bare hardware or on top of an operating system. The earliest adoption of VM technology is IBM VM/370 [1], which supported multiple concurrent virtual machines running on underlying hardware and each of concurrent virtual machines was believed that it was running natively on the IBM System/370 hardware architecture. Afterward, VM technology suffers a depression due to the price of hardware cheapening and performance of operating system upgrading. Until recently, it is begin experiencing a resurgence of popularity due to its many excellent features have been exploited, such as performance isolation, live migration and server consolidation [1].

Server consolidation [1] is the most attractive benefit for both enterprises and research institutes [3][4]. It makes original servers deployed on different physical platforms to be integrated into an individual physical machine: it is an approach to the efficient usage of computer server resources in order to reduce the total number of servers.

Despite the advances in virtualization technology, the overhead of network I/O virtualization can largely degrade the performance of network-intensive applications which have been consolidated on an individual physical machine. For example, a web server running in one guest VM needs to communicate with a database server running in another guest VM in order to satisfy a client transaction request, and these guest VMs running on the same physical machine are sharing the same network resource. Some works [1][2][6] have revealed that each interaction among the guest VMs needs to undergo costly interception and validation by the virtualization layer for security isolation and for data multiplexing and demultiplexing [17], which results in poor performance.

To solve above problems, many researchers have proposed some schemes based on shared memory across guest VMs. Nevertheless, the inefficient of network virtualization is only partially solved by using shared memory buffers. Some solutions only focus on improving spatial efficiency of memory usage, while others focus on throughput but they sacrifice a large portion of memory space.

This paper describes IDTS, a system that attempts to use static shared memory across different domains in order to provide a high performance inter-VM network loopback channel. IDTS borrows the idea of IPC (*Inter-Process Communication*) to fully bypass the virtualized network interface, and networking programs deployed among co-resident guest VMs can achieve high communication performance by leveraging IDTS and cost only a little sacrifice to the transparency of user-level applications. IDTS also supports the bidirectional communications for co-resident VMs and supplies friendly interface for upper applications which do not need much effort to modify their source codes.

The rest of this paper is organized as follows. Section II discusses the related works and background. The design and implementation details are described in section III. Section IV presents the performance evaluation of IDTS and analyzes some issues about inter-domain communications based on shared memory. Finally, we conclude our work in section V.

II. Related works and Background

As more and more important network intensive applications emerge, such as web servers, researchers always try to optimize the performance of network for satisfy the need of the applications. Druschel et al [7] try to increase network performance by caching fast buffers for OSIRIS network adaptor. Götz et al [8] implement a shared-memory-based transport for high-throughput data transfer in the L4 microkernel, whereas these methods are not applicable in VM environments.

Furthermore, performance overheads incurred by networking applications are much higher in VM environments in which I/O operations are relatively expensive. In Xen VMM [21], I/O device follows a split-driver model. Only IDD (*Isolated Device Domain*) has the privilege to access hardware using native device drivers, and all the other guest domains have to pass the I/O requests to the IDD to indirectly access physically devices. This model will definitely involve communication with VMM, which brings much latency for the system running. Although Menon et al [9][10] have boosted the performance of network virtualization in Xen, the improvement is not notable, compared with native environment.

Shared memory buffers is an alternatively method to improve the performance of network virtualization via an interface, called grant tables [18], provided by Xen [5]. Grant tables are used for sharing and transferring page frames between the guest VMs, without requiring the cooperated domains to be privileged. Two operation modes can be performed via the grant table. The first mode allows Dom A to grant Dom B the right to access a specific page frame, but retaining ownership. The block front driver uses the mode to grant memory the right to access the block back driver, so that it may read or write as requested. The second mode allows Dom A to accept a transfer of ownership of a page frame from Dom B. The network front and back driver will use the mode for packet transmitting or receiving.

In practical terms, by leveraging the first mode of grant table, it is feasible to allocate static memory buffer for transferring application level data during the interaction period in a physical machine. The method also can help achieve the throughput of native environment. There are no extra overheads, for example, the invocation of hypercalls. However, the method is not novel.

Xenlog [11][12] is a system focused on protecting the flow path which carrying Domain U’s logging data. The flow path is created by VMM via establishing shared memory between the Domain 0 and Domain U, instead of the network stack, which will result in the integrity and security problems of the transferred logging data. Xenlog is a particular system focus on security, thus lacks the generality and availability to other applications. Moreover, the original intention of Xenlog also produces some flaws in its architecture, for instance, only unidirectional transmission supported.

XenSocket [13] is motivated by *System S* [23]; the goal of *System S* is to extract important information by analyzing voluminous amounts of unstructured and mostly irrelevant data, thereby needing a mechanism to exchange voluminous data rapidly across different process modules. XenSocket is a UNIX-domain-socket-like interface for inter-domain communication on the Xen, it replaces the page-flipping mechanism with a static circular memory buffer shared between domains. However, in order to archive high throughput, XenSocket uses 32 pages as shared circular buffer, and an extra page as shared descriptor page for each connection. As a result, the shared memory of VMM will be quickly exhausted. In addition, during connection initializing period, developers must manually provide grant table reference and event channel port, which makes very inconvenient for developers and users. Though XenSocket has a sockets-based interface, it is different from traditional view of socket due to its one-way communication mechanism. The parameters of XenSocket are also greatly different from those of traditional TCP/IP.

XWay [14] provides a transparent inter-domain transfer mechanism to almost all TCP oriented applications which are running above the hosted OSes by intercepting TCP socket calls beneath the socket layer. These applications require no modification to its source code. However, after being installed and used, we find that XWay had a complex setup course. Furthermore, XWay requires intrusive modifications to the implementation of network protocol stack in the core operating system since Linux does not seem to provide a transparent netfilter-type hooks to intercept message above TCP layer [14]. Some information also indicates that there is no support for automatic discovery of co-resident VMs on XWay, which means that researchers must explicitly and manually designate the domain IDs of Domain U which are co-resident VMs in the phase of installing and configuring XWay.

XenLoop [17] is a more transparent inter-VM network loopback channel than XWay, and requires no intrusive modifications to the Linux kernel and application's source code. It also supports the VM's live migration expediently without evident performance penalty. XenLoop is also easy to setup and uninstall. But there is a tradeoff between convenient and performance for XenLoop: XenLoop shows a poor performance when TCP_STREAM applications are running, that is inferior to other inter-domain mechanisms.

IVC [15] is an inter-VM communication library to support shared memory communication between distinct VMs on the same physical host in Xen environment. IVC has a general socket-style API and intends to be used in HPC applications. As the above systems, it is a one-way communication pipes between guest domains. VM migration is supported on IVC, though not fully transparently at user-space, by invoking callbacks into the user code so it can save any shared-memory state before migration gets underway.

From the low-level interface perspective, all above mechanisms' are based on grant table and treated as a producer-consumer circular buffer features FIFO. The design avoids the need for explicit synchronization between the producer and consumer endpoints [17]. In other respects, event channel, a 1-bit event notification me-

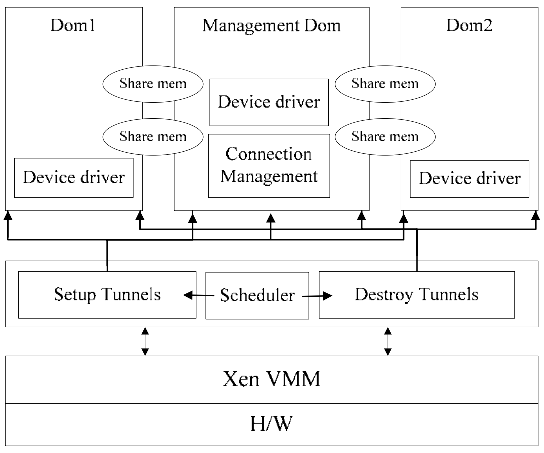

Figure 1. IDTS architecture

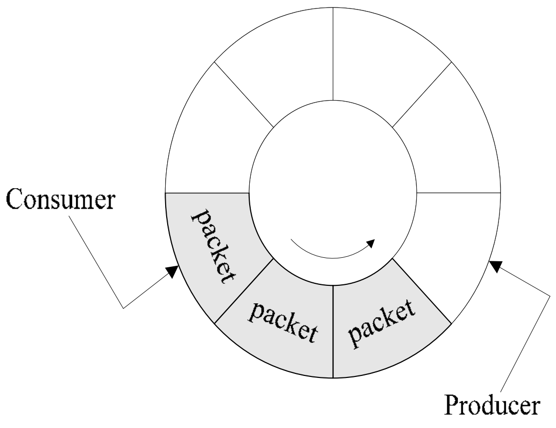

Figure 2. Shared memory

chanism for the endpoints to notify each other of presence of data on the FIFO channel, has been adopted by all mechanisms to facilitate the data transfer on shared memory.

As far as performance is concerned, XenLoop has the most performance lost, partially due to the overhead of registered netfilter hooks used for intercepting and scanning the head of packages from user-level application [10]. The performance cost of Linux networking is accounted by the netfilter [19] module. XWay, Xensocket and IVC eliminate these overheads by placing data directly into shared memory. Therefore they can obtain more bandwidth and less latency for network applications than XenLoop. However, transparency for application is paid for the tradeoff.

The other approaches only focus on the improvement of space efficiency. For example, Kloster et al [16] intends to reduce the number of physical pages required by VMs on the same physical machine through hashing mechanism to locate identical pages and transparently share these identical pages among VMs. However, its intention is to try to minimize memory usage instead of inter-domain communications.

III. Design and Implementation

IDTS supports bidirectional communications between domains by providing two shared memory tunnels without network stack. Every domain plays two roles in our system. It can be both served as a sender and a receiver at the same time. This dual-role ability provides much universality and flexibility to our system. IDTS is also designed for simplicity, high efficiency and scalability. Simplicity means that developers can use the interface of IDTS with fewer complexes than ever. We also pay attention to the space efficiency and time efficiency in the design of our system. Scalability is necessary to allow IDTS to support concurrent multiple applications at the same time. That means many applications can share a connection at the same time instead of creating a new one.

A. Overview

IDTS provides the following properties: (1) high performance for applications; (2) friendly interface for developers; (3) no intrusive change to the OS kernel source.

Figure 1 presents the system architecture of IDTS. There are five main components existing in the system: shared memory, which is allocated by one VM then granted to another VM who wants to communicate with the owner of the shared memory; device driver, encapsulates the detailed implementation for the shared memory and supplies some typically device operations for upper developers; connection manager, takes responsibility for maintaining the links between co-resident VMs; tunnel manager, a more abstractive structure than connection manager, and each tunnel contains two connection; scheduler, a simple schedule mechanism to further explore the performance of IDTS. More details will be discussed in following subsections.

IDTS requires the applications, who want to improve their network performance inside VM, to modify their relative source code minimally to bypassing the network stack entirely. The scheme has high network performance for these applications and friendly interface supplied via its standard device driver. All implementations in IDTS need not any alteration to OS kernel.

B. Shared Memory

Shared memories across guest domains can be used for inter-domain communications. In our system, we provide two shared memories to enable bidirectional communications. Each shared memory is a standard consumer-producer circular buffer. We use FIFO algorithm to schedule data in the shared memories. As shown in Figure 2, consumer pointer points to the next place to read, and producer pointer points to the next place to write. A data unit in the shared memories is comprised of two parts: the first part is metadata structure and the other is payload. The metadata structure consists of three members: first one is used to specify the length of the following data; second one is the identity number of the guest domain who sends the data; and the last one is used to tell the Domain 0 which applications these data should be delivered to. The second one is useful when there are multiple guest domains to communicate with Domain 0. The last one is used when applications want to share the tunnels with each other. It is noteworthy that the last two members can be neglected if single guest domain and application be involved in inter-domain communications.

C. *Device Driver*

There are two kinds of device drivers in our system: front-end driver and back-end driver, which are located in guest domains (also called user domains) and Domain 0 (or privilege domain) respectively.

1) *Front-end Driver*

We provide two interfaces to enable bidirectional communications in Xen, one for receiving data and the other for sending. If inter-domain connections have been created, the front-end driver can read or write the shared memory. In our prototype, we use *dev_read* interface to receive data from other domains, and use *dev_write* interface to send data to other domains. In order to support non-block I/O, we also implement the *dev_poll* interface.

The main function of *dev_read* interface is to copy data from shared memory to user space, and then notify the other domains that some data have been written in the shared memory with event channel, which has been initialized in the period of the system bootstrap. If the buffer ring in the shared memory is empty, the read thread will sleep until the other domain wakes it up when it is in blocked I/O mode, otherwise, it returns error message immediately.

During writing period, data are written to the shared memory, and another notification is used to inform the other domains that data is prepared for reading from shared memory with event channel mechanism. If the shared memory ring is full, the write thread will sleep until the other domain wakes it up when it is in blocked I/O mode, or else it also returns immediately.

In order to support non-block I/O, we implement *dev_poll* interface in front-end driver. It returns a mask which specifies whether a process can read from or write to shared memory immediately. If the shared memory is not full, the mask is set to *POLLIN* and *POLLRDNORM*, meaning that writers can write immediately. If the shared memory is not empty, the mask is set to *POLLOUT* and *POLLWRNORM*, meaning that readers can read immediately.

2) *Back-end Driver*

The design of back-end driver is more complex than front-end driver, since back-end driver should communicate with multiple front-end drivers. In order to read data which is sent by multiple guest domains, a schedule algorithm is needed to manage all the shared memory regions.

When back-end driver is diving into reading, it should select a guest domain whose shared memory has been written. Other operations are the same as those of front-end driver. Back-end driver will copy data from that shared memory and notify the corresponding guest domain. When an application in Dom 0 (also named Domain 0) wants to communicate with another application in Dom U (also named Domain U), it must specify the identity number of the Domain U being contacted with, and the back-end driver will refers to that domain's shared memory, then send data to the domain. At last, notification, which means there is data to read in the shared memory, is sent to that domain. We also support non-block I/O mode in back-end. The poll interface of back-end driver is designed similar with the front-end drivers. The main difference is: if any tunnel can read, the mask must be set to *POLLIN* and *POLLRDNORM*; and if any one tunnel can write, the mask must be set to POLLOUT and POLLWRNORM.

D. *Connection Manager*

It is likely that the privilege domain may communicate with multiple guest domains, so Domain 0 must maintain some related information for these connections. In our prototype, all the connections are organized with a bidirectional list, and each node in the list is a structure, describes a connection across domains. The main members of this structure include the identity number of guest domain, grant table reference of shared memories, event channel ports, the address of shared memories, and the head of bidirectional list. If a guest domain communicates with Domain 0, back-end driver will allocate a node in response. Since such node is fixed size, we can create a particular memory pool, named look-aside cache located in kernel space, to greatly improve the efficiency of our system. Memory blocks in the look-aside cache have the same size, and IDTS can allocate/reclaim a connection structure quickly from/to this memory pool when it needs. Finally, during unloading mode period, IDTS will free the look-aside cache, and the operating system reclaims the memory resource.

When a guest domain begins to connect with Domain 0, the back-end driver needs to check whether the connection has been established. If the back-end fails to locate the connection in the current list, it will allocate a new node from the memory pool immediately, and set this connection's initial status to *DISCONNECTED*. Then, increase the number of front-end, and the new connection node is added to the connection list. When inter-domain communication is finished, the previous connection will be deleted from connection list, and the memory allocated to that connection will be revoked.

E. *Tunnels Manager*

The function of tunnels manager is to create/destroy tunnels for inter-domain communication. Xenbus and Xenstore are involved during the two phases. Xenstore [20] is arranged as a hierarchical collection of key-value pairs. There is a hierarchical directory for each domain for containing data related to its configuration. Guest domains are permitted to edit, create and delete information in its sub-trees information in Xenstore. In IDTS, it is used to create shared memory regions and event channels for the split device drivers, and record information of connection status for the domains. Xenstore is something like a storage media. Grant table references, event channel ports and connection state of the two domains are all stored in it. Whereas Xenbus like a bus between Xenstore and the two domains who want to create or destroy a connection. It is used to read/write metadata from/to Xenstore. Metadata refers to grantable references, event channel ports, domain id, frontend path, connection state and so on. Interactions between Xenstore and domains are shown in Figure 3. In the following paragraph, we will show how the inter-domain tunnels are set up and destroyed in detail.

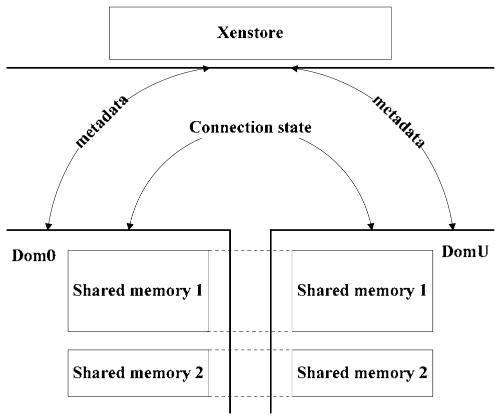

Figure 3. Interactions between Xenstore and domains

1) Setup Tunnels

Inter-domain tunnels must be created before communication, which uses shared memory regions. IDTS sets up two shared memory regions for each unidirectional connection. Each memory region is a number of continuous memory pages allocated by front-end from kernel space. Front-end then interacts with hypervisor to grant remote domain the right to access these memory regions, and two grant table references will be returned by hypercalls. Event channel [19], which is used to notify the remote domain, is also needed for inter-domain communication. IDTS also creates two event channels for the two domains and each event channel has a shared memory region. The creation of event channels like a client-server model. At first, guest domain acts as a server, requests front-end driver to allocate two event channel ports, and the corresponding interrupt numbers will be returned by hypervisor. Then, the guest domain will listen on the new allocated event channel ports, waiting remote domain to connect with itself. At the same time, each allocated event channel port is bounded with an interrupt handling procedure, which is used to process notifications sent by the remote domain. Finally, Xenbus device is called to write grant table references and event channel ports to Xenstore, where remote domain can access this information, and connection state switched to *CONNECTED*.

When back-end is informed that front-end is ready, it will respond to front-end's connection request immediately. First, it calls Xenbus device to read grantable references and event channel ports from Xenstore. Second, back-end driver will call hypervisor to map front-end's two memory regions into its memory space. Third, hypervisor is involved to allocate two inter-domain event channel ports. Then, back-end which acts like a client will connect to front-end by binding its event channel ports to front-end. At the same time, another two interrupt handling procedures, which are used to process front end's notifications, are spawned for back-end.

2) Destroy Tunnels

When communication is completed, inter-domain tunnels must be destroyed in order to avoid memory leakage or some security issues. First, back-end drivers must unmap the two shared memory regions, free inter-domain event channel ports, and terminate the corresponding interrupt. Then back-end switches its status to *DISCONNECTED*. When front-end driver finds that the status of back-end driver has changed to *DISCONNECTED*, it will revoke back-end's privilege of accessing those shared memory regions, free listening ports, and terminate interrupt handling procedures which was binding with the event channels. Finally, frontend also switches its status to *DISCONNECTED*.

F. Scheduler

Beside that have mentioned above, there are many other factors contributing to high performance when communication occurred across domains coexisting on same physical platform, such as domain scheduling and resource allocation. In order to further exploit higher performance in IDTS, we propose a method which can automatically allocate CPU resource according to the real-time workload in guest domains.

In most cases, memory bandwidth depends on the system bus width when processes communicate with each other in native Linux. Memory bandwidth is also limited by the system bus width when considering inter-domain communication. When there are several guest domains communicate with others simultaneously, each domain would try their best to contend for the finite bandwidth resource. So to a great extent, resource allocation strategy can affect the bandwidth competition.

However, since the complexity of the communication situation is obvious, it is difficult to make the resource allocation scheme completely clear. IDTS just implements a coarse-granularity scheduling algorithm, which can dynamically allocate idle CPU resource to these domains that contain multiple tunnels to the other domains.

When the event of creating tunnels is delivered to the hypervisor, IDTS just activate the coarse-granularity schedule policy. Then the scheduler scans the interior of system, identifying domains that just increasing the number of tunnels. If scheduler find the corresponding domains, it will assign an idle CPU resource to the domains.

When the event of destroying tunnels is delivered to the hypervisor, IDTS would launch the scheduler again to revoke the idle CPU resource.

It is important to note that the CPU resource is not infinite, so the tunnel creation to a domain cannot infinitely be conducted. If the number of IDTS' tunnels is being increasing, system will suffer a denial-of-service attack.

To allocate/revoke CPU resource to/from work-intensive domain, IDTS just simply modifies the value of the parameter *vcpu* in the configure file of these domains. The simple scheduler policy is a performance boosts for some given scenarios (detailed in section IV) and is a proof-of-concept for our goals.

IV. Evaluation

In this section, we present the details of our experiment environment and then give some evaluations on IDTS.

A. Evaluation Evironment

In our experiments, we use dual Xeon 1.6GHz CPU with 4MB L2 cache for each, and 4GB of main memory. Domain 0 is configured with dual physical CPUs and

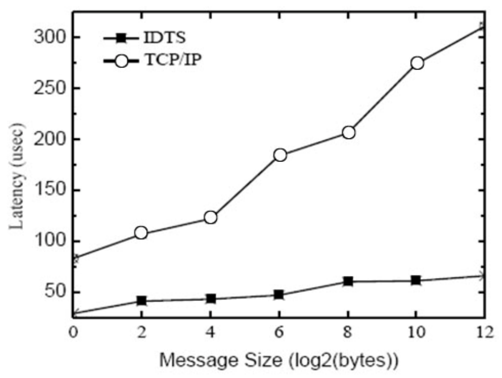

Figure 4. Round trip latency

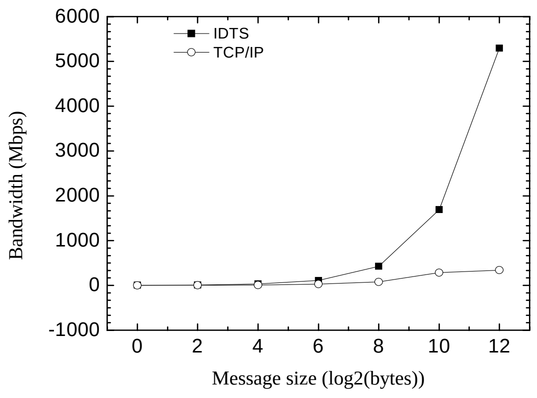

Figure 5. Bandwidth of domain 0 to Domain U

3.5GB of memory. Guest domain is configured to use one virtual CPU and 512MB of memory. Both domains are running Fedora Core 7 GNU/Linux. Hypervisor we developed is based on Xen 3.1.0 and the kernel of the two domains is paravirtualized Linux 2.6.18-xen. Our experiments compare the performance of IDTS with traditional TCP/IP approach. The target of our experiment is to validate that IDTS is more efficient than traditional TCP/IP approach in virtual environment. We evaluate the IDTS's performance from two-folds: request-reply and bulk data transferring. For the first metric, low latency is expected, whereas the second metric, high bandwidth is expected.

B. Round Trip Latency

We measure round trip latency in this section. In order to confirm that IDTS is more efficient than traditional TCP/IP approach, we compare the latency of IDTS with that default TCP/IP method in virtual environment. The impact of the message size is also considered in this section. We conduct each experiment for 100 times, and then calculate the average value for the latency. Figure 4 shows that IDTS achieves a much higher performance than those of TCP/IP based schemes in virtual environments. The average latency of IDTS is in the scope between 29µs to 66µs, while that of TCP/IP in the same scope is between 83µs to 311µs. It is obvious that IDTS has reduced round trip latency by a factor of 4 generally.

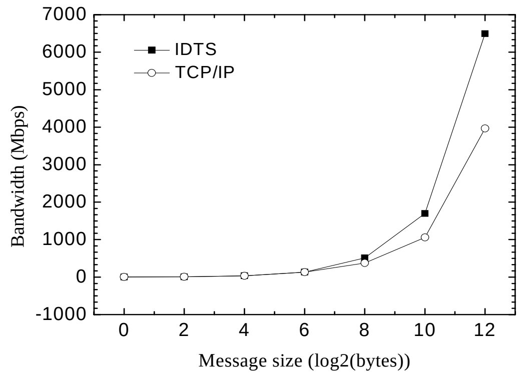

Figure 6. Bandwidth of Domain U to Domain 0

We also find that with the increasing of message size, both TCP/IP and IDTS experiences a performance decline. However, the performance loss of IDTS is not as prominent as TCP/IP. It is implied that the message size has little impact on our system. The result means IDTS can well suit for applications which are sensitive to latency.

C. Bandwidth

Bandwidth is another important performance metric for network application. In this evaluation, a process sends a bulk of messages to remote domain where another process waits to receive data in a non-blocking mode. If the process in the remote domain finds that data have arrived in the shared memories, it will send an acknowledgment back to the sender. The bandwidth is calculated as the ratio of total volume of message transferred to the corresponding lapsed time. According to above tests, we also conduct the experiment for 100 times for the sake of accuracy, and take the average value as evaluated bandwidth. In the following experiments, we test bandwidth both for data transferred from guest domain to privileged domain and from privileged domain to guest domain. We conduct these two similar experiments in the hope of finding whether bandwidth is affected by the direction of data transferring. The tested results are shown in Figure 5 and Figure 6.

Figure 5 shows the bandwidth when data is transferred from Domain 0 to Domain U, and Figure 6 shows the corresponding bandwidth when data is transferred from Domain U to Domain 0. Obviously, from the two experiments, we can conclude that IDTS has a higher throughput than that of traditional TCP/IP approach. The bandwidth of IDTS is approximately 5 times higher than that of TCP/IP in virtual environment. Furthermore, both IDTS and TCP/IP experience a high throughput when data is transferred from Domain U to Domain 0. That is due to memory pages are in their own memory space, and it does not need to interact with hypervisor to map these memory pages.

D. Scalability

In a certain sense, scalability is closely related with security. For example, a domain has established many connections with other VMs: each connection consists of two

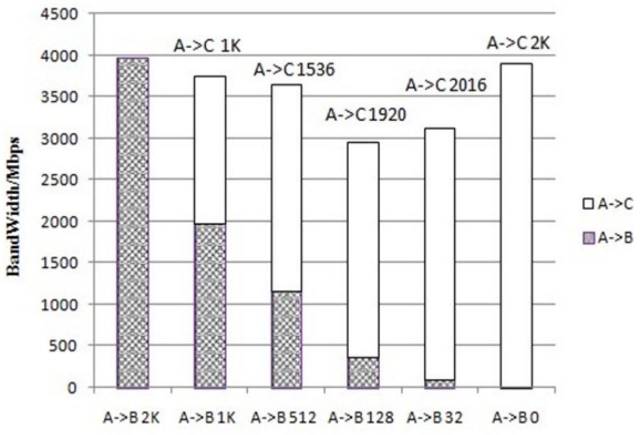

Figure 7. The bandwidth of DomA to DomB and DomC, we vary the send size data individually for per-connection, but the sum of send data size of all connection in one case is constant during the process, is 2048 bytes.

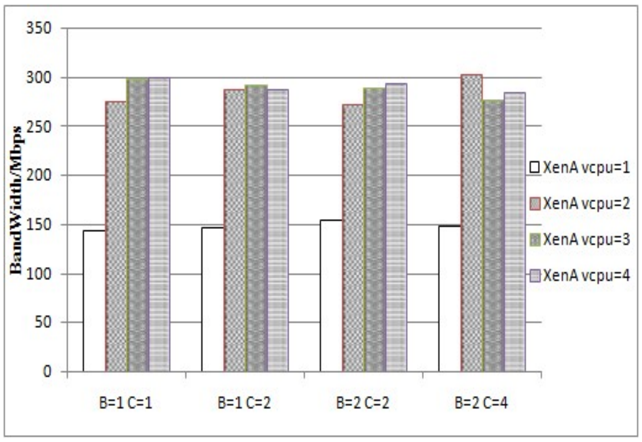

Figure 8. The bandwidth when more vcpus are scheduled to the domains for inter-domain communications, and the size of package sent is 64 bytes (by default, the number of vcpu assigned to a domain is 1)

first-in-first-out (FIFO) data channels and one bidirectional event channel, whereas the resource upper bound is not infinite. So if the number of VMs who have the connections with the domain exceeds a threshold value, the system will suffer from unexpected behaviors, including denial-of-service attacks. Fortunately, VMM or hypervisor can eliminate the risk naturally due to the isolation of VM technology. So the simple solution is to reboot the domain and reset the application running again. But, during the reset proceeding, the service will be disrupted, so some policies should be applied for whole system stability. For example, we should limit the number of connections or provide a gracefully resource sharing management strategy.

In order to evaluate the scalability of IDTS, we boot three guest domains, called DomA, DomB and DomC respectively. We deploy netperf [22] on these domains: DomA acts as client to sends data to the other domains, and two servers are represented by DomB and DomC. During the testing period, we vary the data size to observe the variation in IDTS's total bandwidth. It is worthy to note that the sent data size for all connections in one case is constant during the process, and is about 2048 bytes. From Figure 7, we can see the tiny oscillation has occurred when the data size is changed. The black bar represents a connection from DomA to DomB, and the white bar represents a connection from DomA to DomC. The left bar indicates that there is only one tunnel in the system: DomA to DomB. The right bar indicates that the tunnel from DomA to DomC exists, and their bandwidths all achieve the best value about 4Gb/s. But a little deviation is appeared in the two cases due to the non-determinism and complex of system kernel. The middle of the four bars represents four different cases respectively according to the ratio of sent data size. Each case of the four bars contains the mixture of two tunnels: we tune the proportionality of factor to sent data size per-connection correspondingly for one case, and keep the sum of the send data size invariable for per-case.

The result shows the individual bandwidth of per-tunnel is changed regularly according to its ratio of sent data size in each case, but the total bandwidth is decreased irregularly, especially the fourth one. We have investigated the reason, and concluded that the situation is induced by the event channel mechanism in IDTS: as the number of tunnel is increased, the behavior of the event channel becomes more complex, which brings side-effect to the bandwidth increasing.

To further explore the potential of scalability of IDTS system, we optimize the performance of IDTS by scheduling more VCPU resource to the domains involved in the inter-domain communications. Figure 8 illustrates the idea: when DomA sends data to DomB and DomC simultaneously, two tunnels are created and each tunnel contains two connections. When an extra VCPU resources are assigned to DomA (just modify the parameter “vcpu=1” to “vcpu=2” in DomA’s configure files), the bandwidth of the tunnel is doubled. When the number of VCPUs assigned to DomA continues increasing, the benefit is not significant. We have the following conclusions: for the sender, if the number of VCPUs is less than the number of tunnels, the bandwidth will be bounded by the CPU resource. Contrarily, if the number of tunnels is less than the number of assigned VCPU, the improvement of performance can be neglected; but for the receiver, more CPU resource assigned is unavailing on increasing the bandwidth.

We have not yet fully explored the scalability problems of IDTS in all scenarios, as its implementation is still in an early phase. The above experiments for IDTS’s scalability are not enough. More details will be posted in our future work.

V. CONCLUSION AND FUTURE WORKS

IDTS is an inter-domain communication scheme based on shared memory between different guest VMs, which can offer high performance for network applications with a little sacrifice for their transparency. IDTS also supplies bidirectional communication channels, which can make the IDTS universal for security applications, high performance computing applications, etc.

IDTS's high performance derives from the static shared memory, which is allocated by the grant table mechanism. Compared with traditional TCP/IP approach, experimentations results show that IDTS outperforms regular network communication scheme in round trip latency by a factor of 4, and bandwidth is increased by 5 times at most. The scalability of IDTS is also mentioned and investigated in our paper, and a basic optimization method is proposed and proven. Additionally, due to the bypassing network stack, using IDTS to communicate between virtual processes deployed on an individual physical machine can benefit the security issue.

In the future, we will still focus on some unsupported functions based on the current IDTS system, such as live migration, memory saving, and multicast.

ACKNOWLEDGMENT

This work is supported in part by National 973 Basic Research Program of China under grant No.2007CB310900, Program for New Century Excellent Talents in University with No. NCET-08-0218, Hubei Natural Science Foundation under grant No.2007ABD009 and Wuhan Chengguang Plan with No.200850731350.

REFERENCES

[1] J. R. Creasy, “The Origin of the VM/370 Time-Sharing System”, *IBM J. Research and Development,* Vol.25, No.5, pp.483-490, 1981.

[2] C. Clark, K. Frase, P. Hand, P. Hansen, P. Jul, C. Limpach, P. Pratt, and P. Warfield, “Live Migration of Virtual Machines”, In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation,* Boston, MA, pp.273-286, April 2005.

[3] M. Rosenblumand and T. Garfinkel, “Virtual Machine Monitors: Current Technology and Future Trends”, *IEEE Computer,* pp.39-47, May 2005.

[4] P. M. Chen and B. D. Noble, “When virtual is better than real”, In *Proceedings of Hot Topics in Operating Systems,* pp.133-138, Aug 2001.

[5] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, I. Warfield, P. Barham, and R. Neugebauer, “Xen and the art of virtualization”, In *Proceedings of the ACM Symposium on Operating Systems Principles*, pp.164-177, October 2003.

[6] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, “Reconstructing I/O”, *Technical Report UCAM-CL-TR-596*, University of Cambridge, *UK*, 2004.

[7] P. Druschel, L. L. Peterson, and B. S. Davie, “Experiences with a high-speed network adaptor: A software perspective”, In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, New York, NY, USA, ACM Press, pp.2–13, 1994.

[8] S. Götz, “Synchronous communication using synchronous IPC primitives,” *Diploma thesis, System Architecture Group, University of Karlsruhe, Germany*, May 20

[9] A. Menon, A. Cox, and W. Zwaenepoel, “Optimizing network virtualization in Xen”, In *Proceeding of the 2006 USENIX Annual Technical Conference*, Boston, Massachusetts, USA, pp.15-28, June 2006.

[10] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and J. W. Zwaenepoe, “Diagnosing performance overheads in the Xen virtual machine environment”, In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, Chicago, Illinois, USA, pp.13-23, June 2005.

[11] N. A. Quynh, R. Ando, and Y. Takefuji, “Centralized Security Policy Support for Virtual Machine”, In *Proceedings of the 20th Conference on Large Installation System Administration conference (LISA’06), USENIX, Washington D.C, USA*, pp.79-87, December, 2006.

[12] N. A. Quynh and P. Takefuji, “A Central and Secured Logging Data Solution for Xen Virtual Machine”, In *Proceedings of the 24th IASTED International Multi-Conference Parallel and Distributed Computing and Networks*, Innsbruck, Austria, pp.218-224, February 2006.

[13] X. L. Zhang, S. Mointosh, P. Rohatgi, and J. L. Griflin, “Socket: A high-throughput interdomain transport for virtual machines”, In *Proceedings of Middleware 2007,* Nov, 2007.

[14] 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*, pp.11-20, Mar 2008.

[15] H. Wei, M. J. Koop, Q. Gao, and D. K. Panda, “Virtual Machine Aware Communication Libraries for High Performance Computing”, In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing,* Reno, NV, pp.1-12, Nov 2007.

[16] J. F. Kloster, J. Kristensen, and A. Mejlholm, “Efficient memory sharing in the Xen virtual machine monitor”, *Technical report, Aalborg University*, Jan 2006.

[17] W. Jian, K. L. Wright, and K. Gopalan, “XenLoop: A Transparent High Performance Inter-VM Network Loop-back”, In *Proceedings of the 17th international symposium High performance distributed computing*, Boston, Massachusetts, USA, pp.109-118, June 2008.

[18] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, “Safe hardware access with the Xen virtual machine monitor”, In *Proceeding of 1st Workshop on Operating System and Architectural Support for the on demand IT Infrastructure (OASIS)*, Oct 2004.

[19] J. R. Santos, Y. Turner, and G. Janakiraman, “Bridging the Gap between Software and Hardware Techniques for I/O Virtualization”, In *Proceedings of the 2008 USENIX Annual Technical Conference*, Boston, Massachusetts, pp.29-42, June 2008.

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

[21] Xen project: Xen interface manual. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/interface/interface.html.

[22] Netperf. http://www.netperf.org/.

[23] Umbrella System S.
http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.index.html