Cluster Comput (2009) 12: 141–152
DOI 10.1007/s10586-009-0079-x

XenLoop: a transparent high performance inter-VM network loopback

Jian Wang · Kwame-Lante Wright · Kartik Gopalan

Received: 29 December 2008 / Accepted: 6 January 2009 / Published online: 17 January 2009

Abstract Advances in virtualization technology have focused mainly on strengthening the isolation barrier between virtual machines (VMs) that are co-resident within a single physical machine. At the same time, a large category of communication intensive distributed applications and software components exist, such as web services, high performance grid applications, transaction processing, and graphics rendering, that often wish to communicate across this isolation barrier with other endpoints on co-resident VMs. State of the art inter-VM communication mechanisms do not adequately address the requirements of such applications. TCP/UDP based network communication tends to perform poorly when used between co-resident VMs, but has the advantage of being transparent to user applications. Other solutions exploit inter-domain shared memory mechanisms to improve communication latency and bandwidth, but require applications or user libraries to be rewritten against customized APIs—something not practical for a large majority of distributed applications. In this paper, we present the design and implementation of a fully transparent and high performance inter-VM network loopback channel, called XenLoop, in the Xen virtual machine environment. XenLoop does not sacrifice user-level transparency and yet achieves high communication performance between co-resident guest VMs. XenLoop intercepts outgoing network packets beneath the network layer and shepherds the packets destined to co-resident VMs through a high-speed inter-VM shared memory channel that bypasses the virtualized network interface. Guest VMs using XenLoop can migrate transparently across machines without disrupting ongoing network communications, and seamlessly switch between the standard network path and the XenLoop channel. In our evaluation using a number of unmodified benchmarks, we observe that XenLoop can reduce the inter-VM round trip latency by up to a factor of 5 and increase bandwidth by a up to a factor of 6.

Keywords Virtual machine · Inter-VM communication · Xen

J. Wang · K. Gopalan (✉)
Computer Science, Binghamton University, Binghamton, NY, USA
e-mail: kartik@cs.binghamton.edu

J. Wang
e-mail: jianwang@cs.binghamton.edu

K.-L. Wright
Electrical Engineering, The Cooper Union, New York, NY, USA
e-mail: wright2@cooper.edu

1 Introduction

Virtual Machines (VMs) are rapidly finding their way into data centers, enterprise service platforms, high performance computing (HPC) clusters, and even end-user desktop environments. The primary attraction of VMs is their ability to provide functional and performance isolation across applications and services that share a common hardware platform. VMs improve the system-wide utilization efficiency, provide live migration for load balancing, and lower the overall operational cost of the system.

Hypervisor (also sometimes called the virtual machine monitor) is the software entity which enforces isolation across VMs residing within a single physical machine, often in coordination with hardware assists and other trusted software components. For instance, the Xen [1] hypervisor runs at the highest system privilege level and coordinates with

a trusted VM called Domain 0 (or Dom0) to enforce isolation among unprivileged guest VMs. Enforcing isolation is an important requirement from the viewpoint of security of individual software components. At the same time enforcing isolation can result in significant communication overheads when different software components need to communicate across this isolation barrier to achieve application objectives. For example, a distributed HPC application may have two processes running in different VMs that need to communicate using messages over MPI libraries. Similarly, a web service running in one VM may need to communicate with a database server running in another VM in order to satisfy a client transaction request. Or a graphics rendering application in one VM may need to communicate with a display engine in another VM. Even routine inter-VM communication, such as file transfers or heartbeat messages may need to frequently cross this isolation barrier.

In all the above examples, when the VM endpoints reside on the same physical machine, *ideally we would like to minimize the communication latency and maximize the bandwidth, without having to rewrite existing applications or communication libraries.* Most state of the art inter-VM communication mechanisms provide either application transparency, or performance, but not both. For example, the Xen platform enables applications to transparently communicate across VM boundary using standard TCP/IP sockets. However, all network traffic from the sender VM to receiver VM is redirected via Dom0, resulting in a significant performance penalty. To illustrate this overhead, columns 1 and 2 in Table 1 compare the performance of the original network communication path between two different machines across a 1 Gbps Ethernet switch versus that between two Xen VMs on the same physical machine (labeled "Netfront/Netback"). Flood Ping RTT refers to the average ICMP ECHO request/reply latency. Rows 2–5 use the *netperf* [12] benchmark. TCP_RR and UDP_RR report average number of 1-byte request-response transactions/sec. TCP_STREAM and UDP_STREAM report average bandwidth. Row 6 shows bandwidth performance using the *lmbench* [7] benchmark. One can see that in all cases, except TCP_STREAM, original inter-VM communication performance is only marginally better or even slightly worse than inter-machine performance, although one might expect a significantly better communication performance within the same machine.

To improve inter-VM communication performance, prior works [4, 5, 18] have exploited the facility of inter-domain shared memory provided by the Xen hypervisor, which is more efficient than traversing the network communication path via Dom0. With [4, 18], network applications and/or communication libraries need to be rewritten against new APIs and system calls, thus giving up user-level transparency. With [5], guests' operating system code needs to be modified and recompiled, giving up kernel-level transparency.

Table 1 Latency and bandwidth comparison

	Inter Machine	Netfront/ Netback	XenLoop
Flood Ping RTT (µs)	101	140	28
netperf TCP_RR (trans/sec)	9387	10236	28529
netperf UDP_RR (trans/sec)	9784	12600	32803
netperf TCP_STREAM (Mbps)	941	2656	4143
netperf UDP_STREAM (Mbps)	710	707	4380
lmbench TCP (Mbps)	848	1488	4920

In this paper, we present the design and implementation of a fully transparent and high performance inter-VM network loopback channel called XenLoop that permits direct network traffic exchange between two VMs in the same machine without the intervention of a third software component, such as Dom0, along the data path. XenLoop operates transparently beneath existing socket interfaces and libraries. Consequently, XenLoop allows existing network applications and libraries to benefit from improved inter-VM communication without the need for any code modification, recompilation, or relinking. Additionally, XenLoop does not require any changes to either the guest operating system code or the Xen hypervisor since it is implemented as a self-contained Linux kernel module. Guest VMs using XenLoop can automatically detect the identity of other co-resident VMs and setup/teardown XenLoop channels on-the-fly as needed. Guests can even migrate from one machine to another without disrupting ongoing network communications, seamlessly switching the network traffic between the standard network path and the XenLoop channel. Our current prototype focuses on IPv4 traffic, although XenLoop can be extended easily to support other protocol types.

A snapshot of performance results for XenLoop in column 3 of Table 1 shows that, compared to original network data path, XenLoop reduces the inter-VM round-trip latency by up to a factor of 5 and bandwidth by up to a factor of 6. The XenLoop source code is publicly available [17].

The rest of this paper is organized as follows. Section 2 covers relevant background for network datapath processing and the shared memory facility in Xen. Section 3 presents the design and implementation of XenLoop. Section 4 presents the detailed performance evaluation of XenLoop. Section 5 discusses related work and Sect. 6 summarizes our contributions and outlines future improvements.

2 Xen networking background

Xen virtualization technology provides close to native machine performance through the use of *para-virtualization*—a technique by which the guest OS is co-opted into reducing the virtualization overhead via modifications to its hardware dependent components. In this section, we review the relevant background of the Xen networking subsystem as it relates to the design of XenLoop. Xen exports virtualized views of network devices to each guest OS, as opposed to real physical network cards with specific hardware make and model. The actual network drivers that interact with the real network card can either execute within Dom0—a privileged domain that can directly access all hardware in the system—or within Isolated Driver Domains (IDD), which are essentially driver specific virtual machines. IDDs require the ability to hide PCI devices from Dom0 and expose them to other domains. In the rest of the paper, we will use the term *driver domain* to refer to either Dom0 or the IDD that hosts the native device drivers.

The physical network card can be multiplexed among multiple concurrently executing guest OSes. To enable this multiplexing, the privileged driver domain and the unprivileged guest domains (DomU) communicate by means of a split network-driver architecture shown in Fig. 1. The driver domain hosts the backend of the split network driver, called *netback*, and the DomU hosts the frontend, called *netfront*. The netback and netfront interact using high-level network device abstraction instead of low-level network hardware specific mechanisms. In other words, a DomU only cares that it is using a network device, but doesn't worry about the specific type of network card.

Netfront and netback communicate with each other using two producer-consumer ring buffers—one for packet reception and another for packet transmission. The ring buffers are nothing but a standard lockless shared memory data structure built on top of two primitives—grant tables and event channels. Grant table can be used for bulk data transfers across domain boundaries by enabling one domain to allow another domain to access its memory pages. The access mechanism can consist of either sharing or transfer of pages. The primary use of the grant table in network I/O is to provide a fast and secure mechanism for unprivileged domains (DomUs) to receive indirect access to the network hardware via the privileged driver domain. They enable the driver domain to set up a DMA based data transfer directly to/from the system memory of a DomU rather than performing the DMA to/from driver domain's memory with the additional copying of the data between DomU and driver domain.

The grant table can be used to either *share* or transfer pages between the DomU and driver domain. For example, the frontend of a split driver in DomU can notify the Xen hypervisor (via the `gnttab_grant_foreign_access` hypercall) that a memory page can be shared with the driver domain. The DomU then passes a grant table reference via the event channel to the driver domain, which directly copies data to/from the memory page of the DomU. Once the page access is complete, the DomU removes the grant reference (via the `gnttab_end_foreign_access` call).

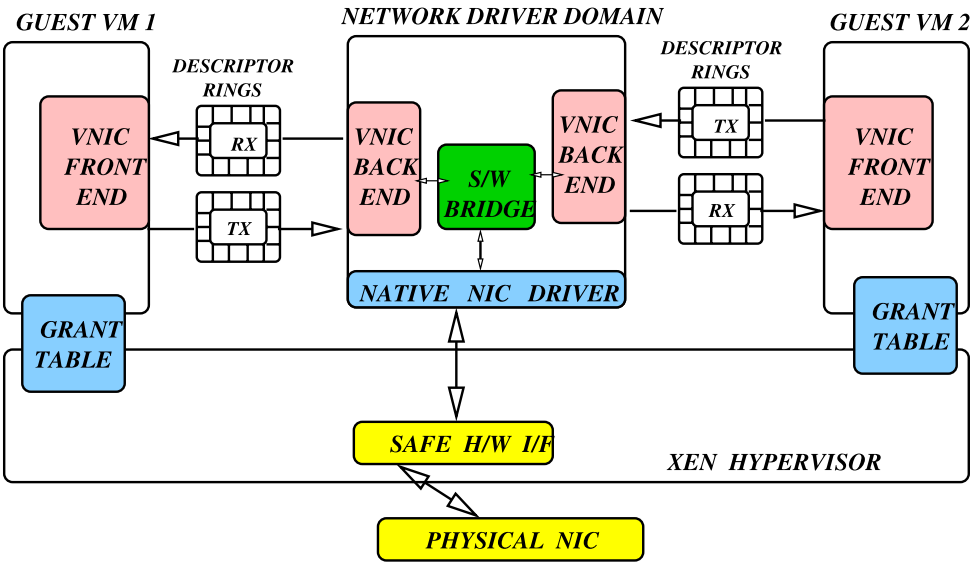


Fig. 1 Split Netfront-Netback driver architecture in Xen. Network traffic between VM1 and VM2 needs to traverse via the software bridge in driver domain

Such page sharing mechanism is useful for synchronous I/O operations, such as sending packets over a network device or issuing read/write to a block device.

At the same time, network devices can receive data asynchronously, that is, the driver domain may not know the target DomU for an incoming packet until the entire packet has been received and its header examined. In this situation, the driver domain first DMAs the packet into its own memory page. Next, depending on whether the received packet is small, the driver domain can choose to copy the entire packet to the DomU's memory across a shared page. Alternatively, if the packet is large, the driver domain notifies the Xen hypervisor (via the `gnttab_grant_foreign_transfer` call) that the page can be transferred to the target DomU. The DomU then initiates a transfer of the received page from the driver domain and returns a free page back to the hypervisor. Excessive switching of a CPU between domains can negatively impact the performance due to increase in TLB and cache misses. An additional source of overhead can be the invocation of frequent hypercalls (equivalent of system calls for the hypervisor) in order to perform page sharing or transfers. Security considerations may also force a domain to zero a page being returned in exchange of a page transfer, which can negate the benefits of page transfer to a large extent [8].

3 Design and implementation

The two basic design objectives behind XenLoop are (a) user-level transparency and (b) significantly higher inter-VM communication performance than via netfront-netback. In this section, we describe the detailed design choices and tradeoffs in implementing XenLoop, justify our design decisions, and present implementation details.

3.1 XenLoop module in guest VM

Here we will discuss an overview of XenLoop architecture shown in Fig. 2 and will discuss each component in greater detail in the following subsections. Each guest VM hosts a self-contained XenLoop kernel module which inserts itself as a thin layer in the network protocol stack between the network layer and the link layer. The XenLoop module contains a guest-specific software bridge that is capable of intercepting every outgoing packet from the network layer in order to inspect its header to determine the packet's destination. Linux provides a netfilter [11] hook mechanism to perform this type of packet interception. The netfilter hook enables XenLoop to handle packets of different protocol types, though currently our protocol focuses on IPv4.

The XenLoop module also contains a mapping table that stores the identity, as [guest-ID, MAC address] pair, of every other guest VM within the same physical machine. This mapping table is dynamically populated using a soft-state domain discovery mechanism described later in Sect. 3.2.

Whenever two guest VMs within the same machine have an active exchange of network traffic, they dynamically set up a bidirectional inter-VM data channel between themselves using a handshake protocol. This channel bypasses the standard data path via Dom0 for any communication involving the two guests. Conversely, the Guests can choose

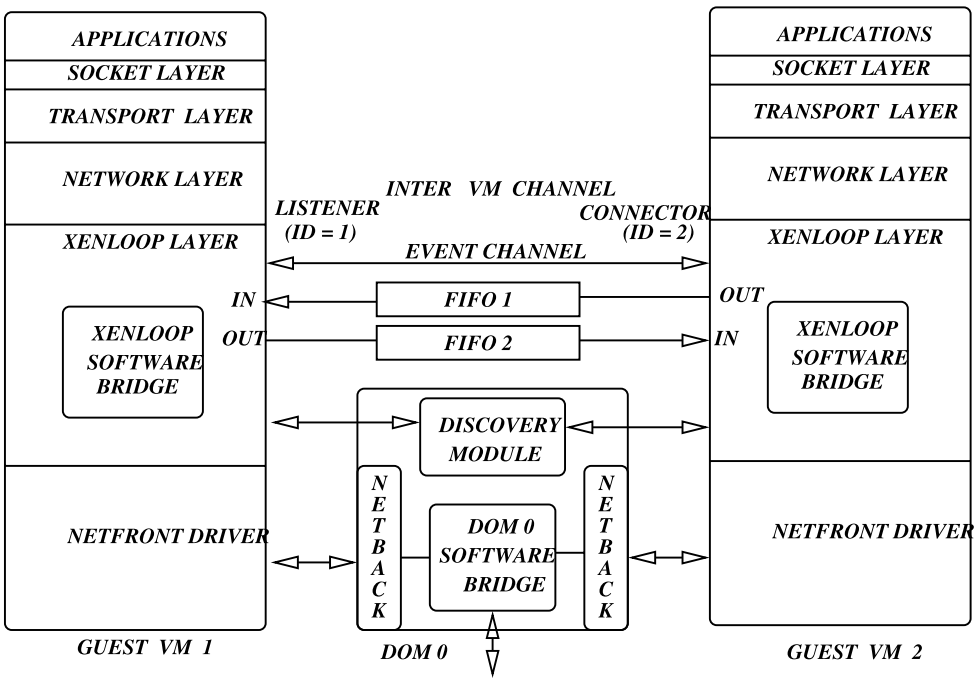


Fig. 2 XenLoop architecture showing XenLoop module in guest, the domain discovery module in Dom0, and the three components of inter-VM communication channel

to tear down the channel in the absence of the active traffic exchange in order to conserve system resources.

For each outgoing packet during data communication, the software bridge first inspects its destination address and resolves the layer-2 MAC address of the next-hop node. This resolution is done with the help of a system-maintained neighbor cache, which happens to be the ARP-table cache in the case of IPv4. The software bridge then looks up the mapping table to check if the next hop node is a guest VM within the same machine, in which case the packet is forwarded over the inter-VM data channel. If the FIFO is full, or the packet cannot fit into the available space, then the packet is placed in a waiting list to be sent once enough resources are available. If the next hop node is not a VM within the same machine, or if the packet size is bigger than the FIFO size, then the packet is forwarded using the standard netfront-netback data path via the driver domain.

3.2 Discovering co-resident guest VMs

In order to set up inter-VM channels with other guests, a guest needs to first discover the identity of co-resident guests. Similarly, to tear down a stale inter-VM channel, a guest needs to determine when the other endpoint no longer exists. Ideally, we want this to occur transparently without administrator intervention. To enable such transparent setup and teardown of inter-VM channels, XenLoop employs a soft-state domain discovery mechanism. Dom0, being a privileged domain, is responsible for maintaining XenStore—a store of key-value pairs representing different system configuration parameters, including information about each active guest VM. Whenever a new guest VM is created in a machine, or when it migrates in from another machine, new entries for that VM are created in XenStore to represent its state. Conversely, when a VM dies or is migrated away from a machine, its information in XenStore is destroyed. Only Dom0 is capable of collating the XenStore information about all active guests; unprivileged guest domains can read and modify their own XenStore information, but not each others' information.

In order to advertise its willingness to set up XenLoop channels with co-resident guests, the XenLoop module in each guest VM creates a XenStore entry named "xenloop" under its XenStore hierarchy (presently "/local/domain/<guest-ID>/xenloop"). A *Domain Discovery* module in Dom0 periodically (every 5 seconds) scans all guests in XenStore, looking for the "xenloop" entry in each guest. It compiles a list of [guest-ID, MAC address] identity pairs for all active guest VMs in the machine that advertise the "xenloop" entry and, by implication, their willingness to participate in setting up inter-VM channels. The Domain Discovery module then transmits an *announcement* message—a network packet with a special XenLoop-type layer-3 protocol ID—to each willing guest, containing the collated list of their [guest-ID, MAC address] identity pairs. Absence of the "xenloop" entry in XenStore for any guest leads to that guest VM's identity being removed from future announcement messages. The above mechanism provides a soft-state discovery design where only the guests that are alive and have an active XenLoop module participate in communication via inter-VM channels.

The need to perform domain discovery announcements from Dom0 arises because Xen does not permit unprivileged guests to read XenStore information about other co-resident guests. Another alternative discovery mechanism, that doesn't require a Discovery Module in Dom0, could be to have each guest VM's XenLoop module to broadcast its own presence to other guests using special XenLoop-type self-announcement network messages. However this requires the software bridge in Dom0 to be modified to prevent the XenLoop-type broadcasts from leaving the local machine into the external network.

3.3 Inter-VM communication channel

The heart of XenLoop module is a high-speed bidirectional inter-VM channel. This channel consists of three components—two first-in-first-out (FIFO) data channels, one each for transferring packets in each direction, and one bidirectional event channel. The two FIFO channels are set up using the inter-domain shared memory facility, whereas the event channel is a 1-bit event notification mechanism for the endpoints to notify each other of presence of data on the FIFO channel.

FIFO design: Each FIFO is a producer-consumer circular buffer that avoids the need for explicit synchronization between the producer and the consumer endpoints. The FIFO resides in a piece of shared memory between the participating VMs. Each entry in the FIFO consists of a leading 8-byte metadata followed by the packet payload. To avoid the need for synchronization, the maximum number of 8-byte entries in the FIFO is set to 2^k, while the *front* and *back* indices used to access to the FIFO are m-bits wide, where $m > k$. In our prototype, $m = 32$ and k is configurable to any value up to 31. Both front and back are always atomically incremented by the consumer and producer respectively, as they pop or push data packets from/to the FIFO. It can be easily shown [2] that with the above design, we do not need to worry about producer-consumer synchronization or wrap-around boundary conditions. The situation when multiple producer threads might concurrently access the front of the FIFO, or multiple consumer threads the back, is handled by using producer-local and consumer-local spinlocks respectively that still do not require any cross-domain produce-consumer synchronization. Hence the FIFOs are designed to be lockless as far as producer-consumer interaction is concerned.

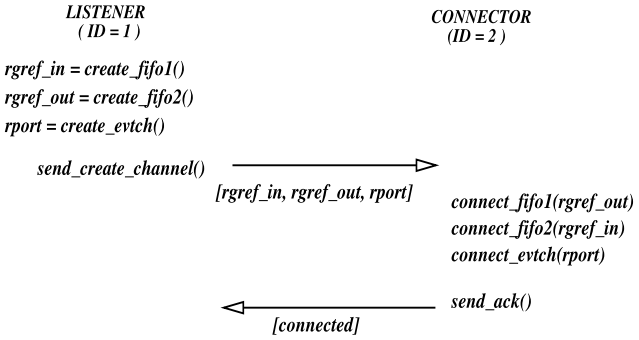


Fig. 3 Bootstrapping the Inter-VM channel. The guest with smaller ID acts as the listener and creates the shared FIFO pages and event channel, whereas the guest with larger ID acts as connector

Channel bootstrap: When one of the guest VMs detects the first network traffic destined to a co-resident VM, it initiates a bootstrap procedure to set up the two FIFOs and the event channel with the other endpoint. Figure 3 illustrates this bootstrap procedure. While the bootstrap is in progress, the network traffic continues to traverse via the standard netfront-netback data path. The bootstrap procedure is similar to a "client-server" connection setup. During bootstrap, the guest VM with the smaller guest ID assumes the role of "server", or the *listener*, whereas the other VM assumes the role of "client", or the *connector*. The identity of the listener VM's channel endpoint needs to be communicated to the connector VM using out-of-band XenLoop-type messages via the netfront-netback channel. The listener VM sets up shared memory pages for the two FIFOs and grants access to the connector VM to map these FIFO pages in its address space. The listener VM also creates an event channel endpoint to which the connector VM is permitted to bind. The listener then sends a *create channel* message to the connector, which contains three pieces of information—two grant references, one each for a shared descriptor page for each of the two FIFOs, and the event channel port number to bind to. The grant references for the remaining data pages of each FIFO are stored within the descriptor FIFO page. Upon receiving the *create channel* message, the connector VM maps the descriptor page for each FIFO, reads the grant references for remaining FIFO data pages from the descriptor page, maps the data pages as well to its address space, and binds to the event channel port of the listener VM. The connector VM completes the inter-VM channel setup by sending a *channel ack* message to the connector. To protect against loss of either message, the listener times out if the *channel ack* does not arrive as expected and resends the *create channel* message 3 times before giving up.

Data transfer: Once the inter-VM channel is bootstrapped, the network traffic between VMs can now be exchanged over this channel, bypassing the standard netfront-netback data path. Note that the distinction between the roles of listener and connector is only for the duration of channel bootstrap, and not during the actual data transfer. Both the endpoints play symmetric roles during the actual data transfer, acting as both senders and receivers of data. The XenLoop module in the sender intercepts all data packets on their way out from the network layer. If the packets are destined to a connected co-resident VM, the sender copies these packets onto its outgoing FIFO, which is conversely the incoming FIFO for the receiver. After copying the packet onto the FIFO, the sender signals the receiver over the event channel, which in turn asynchronously invokes a pre-registered callback at the receiver's XenLoop module. The receiver copies the packets from the incoming FIFO into its network buffers, passes the packets to the network layer (layer-3), and frees up the FIFO space for future packets.

Comparing options for data transfer: The above mechanism involves two copies per packet, once at the sender side onto the FIFO and once at the receiver from the FIFO. We purposely eschew the use of page-sharing or page-transfer mechanism employed by netback-netfront interface due to the following reasons. An alternative to copying data packets would be that the sender should explicitly grant the receiver permission to either share the packet's data page or transfer it. The sharing option requires one event channel notification to the receiver, one hypercall by the receiver to map the page, and another hypercall by the receiver to release the page. (Granting and revoking access permissions do not require a hypercall at the sender side since the grant table is mapped to the sender's address space.) The transfer option requires one event channel notification from the sender, one hypercall by the receiver to transfer a page, and another hypercall to give up a page in exchange to the hypervisor. Additionally, both sides have to zero out in advance the contents of any page that they share, transfer, or give up to avoid any unintentional data leakage. This is known within the Xen community to be an expensive proposition.

One additional option to avoid the copy at the receiver would be to directly point the Linux network packet data structure `struct sk_buff` to the data buffer in the FIFO, and free the corresponding FIFO space *only after* the protocol stack has completed processing the packet. We also implemented this option and found that the any potential benefits of avoiding copy at the receiver are overshadowed by the large amount of time that the precious space in FIFO could be held up during protocol processing. This delay results in back-pressure on the sender via the FIFO, significantly slowing down the rate at which FIFO is populated with new packets by the sender. Thus we adopt a simple two-copy approach as opposed to the above alternatives.

Channel teardown: Whenever a guest VM shuts down, removes the XenLoop module, migrates, or suspends, all active inter-VM channels need to be cleanly torn down. The guest winding down first removes its "xenloop" advertisement entry in XenStore to forestall any new XenLoop

connections. It then marks all active FIFO channels as "inactive" in the shared descriptor pages, notifies the guest VMs at other endpoint over the respective event channels, and disengages from both the FIFO pages and the event channel. The other guest endpoints notice the "inactive" state marked in shared descriptor page of each FIFO and similarly disengage. The disengagement steps are slightly asymmetrical depending upon whether initially each guest bootstrapped in the role of a listener or a connector.

3.4 Transparently handling VM migration

XenLoop transparently adapts to the migration of VMs across physical machines. If two communicating VMs, that were originally on separate machines, now become co-resident on the same machine as a result of migration, then the Dynamic Discovery module on Dom0 detects and announces their presence to other VMs on the same machine, enabling them to set up a new XenLoop channel. Conversely, when one of two co-resident VMs is about to migrate, it receives a callback from the Xen Hypervisor, which allows it to delete its "xenloop" advertisement entry in XenStore, and gracefully save any outgoing packets or receive incoming packets that may be pending in all its inter-VM channels, before disengaging from the channel itself. The saved outgoing packets can be resent once the migration completes. The entire response to migration is completely transparent to user applications in the migrating guest, does not require application-specific actions, and does not disrupt any ongoing communications. On a related note, XenLoop responds similarly to *save-restore* and *shutdown* operations on a guest.

4 Performance evaluation

In this section, we present the performance evaluation of XenLoop prototype. All experiments were performed using a test machine with dual core Intel Pentium D 2.8 GHz processor, 1 MB cache, and 4 GB main memory. We deployed Xen 3.2.0 for the hypervisor and paravirtualized Linux 2.6.18.8 for the guest OS. Another Intel Pentium D 3.40 GHz machine, with 2 MB cache and 4 GB main memory, was also used to measure the native machine-to-machine communication performance over a 1 Gbps Ethernet switch. We configured two guest VMs on the test machine with 512 MB of memory allocation each for inter-VM communication experiments. Our experiments compare the following four communication scenarios:

- *Inter-machine*: Native machine-to-machine communication across a Gigabit switch.
- *Netfront-netback*: Guest-to-guest communication via standard netfront-netback datapath.
- *XenLoop*: Guest-to-guest communication via the XenLoop inter-VM communication channel.
- *Native loopback*: Network communication between two processes within a non-virtualized OS via the local loopback interface. This workload serves as a baseline comparison for other scenarios.

For test workloads, we use three unmodified benchmarks, namely `netperf` [12], `lmbench` [7], and `netpipe-mpich` [13–15], in addition to ICMP ECHO REQUEST/REPLY (flood ping). `netperf` is a networking performance benchmark that provides tests for both unidirectional throughput and end-to-end latency. `lmbench` is a suite of portable benchmarks that measures a UNIX system's performance in terms of various bandwidth and latency measurement units. `netpipe-mpich` is a protocol independent performance tool that performs request-response tests using messages of increasing size between two processes which could be over a network or within an SMP system.

4.1 Snapshot of microbenchmarks

Table 2 compares the measured bandwidth across four different communication scenarios using four benchmark generated workloads. We observe that across all cases, the improvement in bandwidth for XenLoop over the netfront-netback ranges from a factor of 1.55 to 6.19. Table 3 compares the measured latency across the four communication scenarios using four benchmark-generated request-response workloads. Compared to netfront-netback, XenLoop yields 5 times smaller ping latency, 3 times smaller latency with lmbench, 2.8 times higher transactions/sec with netperf TCP_RR, 2.6 times higher transactions/sec with UDP_RR, and 2.43 times smaller latency with netpipe-mpich. The latency performance gap of XenLoop against native loopback

Table 2 Average bandwidth comparison

	Inter Machine	Netfront/ Netback	XenLoop	Native Loopback
lmbench (tcp) (Mbps)	848	1488	4920	5336
netperf (tcp) (Mbps)	941	2656	4143	4666
netperf (udp) (Mbps)	710	707	4380	4928
netpipe-mpich (Mbps)	645	697	2048	4836

Table 3 Average latency comparison

	Inter Machine	Netfront/ Netback	XenLoop	Native Loopback
Flood Ping RTT (µs)	101	140	28	6
lmbench (µs)	107	98	33	25
netperf TCP_RR (trans/sec)	9387	10236	28529	31969
netperf UDP_RR (trans/sec)	9784	12600	32803	39623
netpipe-mpich (µs)	77.25	60.98	24.89	23.81

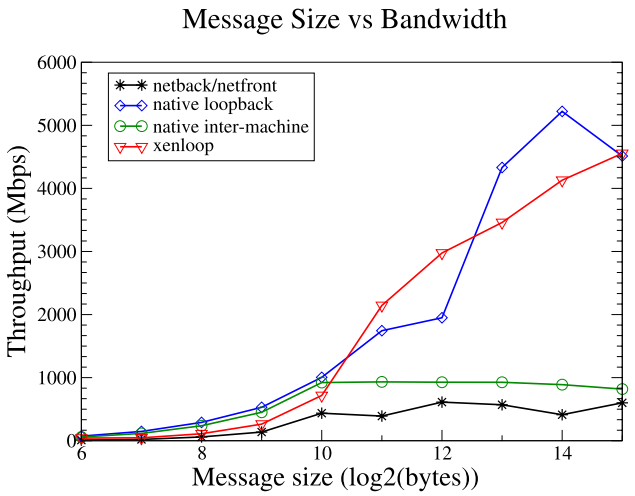


Fig. 4 Throughput versus UDP message size using netperf benchmark

is wider than in the bandwidth case, being worse by a factor ranging from 1.2 to 4.6. Also note that average latency for netfront-netback is either marginally better than inter-machine latency, or sometimes worse. This illustrates the large overhead incurred during small message exchanges across netfront-netback interface.

4.2 Impact of message and FIFO sizes

Figure 4 plots the bandwidth measured with netperf's UDP_STREAM test as the sending message size increases. Bandwidth increases for all four communication scenarios with larger message sizes. This is because smaller messages imply a larger number of system calls to send the same number of bytes, resulting in more user-kernel crossings. For

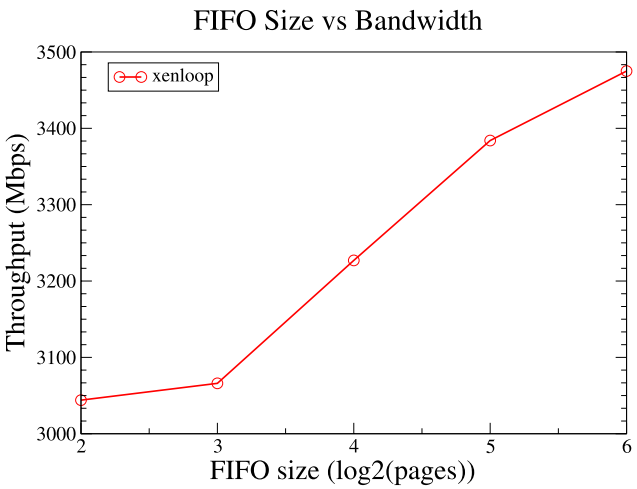


Fig. 5 Throughput versus FIFO size using netperf UDP bandwidth test

packets larger than 1 KB, XenLoop achieves higher bandwidth than both netfront-netback and native inter-machine communication. For packets smaller than 1 KB, native inter-machine communication yields slightly higher bandwidth than both XenLoop and netfront-netback due to significant domain switching and split-driver induced overheads for small packet sizes. Netback/netfront yields slightly lower bandwidth than native inter-machine communication across most message sizes. Beyond 1 KB packet size, neither native loopback nor XenLoop appear to be conclusively better than the other. Figure 5 shows that increasing the FIFO size has a positive impact on the achievable bandwidth. In our experiments, we set the FIFO size at 64 KB in each direction.

4.3 MPI benchmark performance

Next we investigate the performance of XenLoop in the presence of MPI applications using the netpipe-mpich benchmark in the four communication scenarios. Netpipe-mpich executes bandwidth and latency tests using request-response transactions with increasing message sizes to measure MPICH performance between two nodes. Figure 6 plots the one-way bandwidth variation and Fig. 7 plots the latency variation with increasing message sizes. These results validate our observations in earlier experiments. XenLoop latency and bandwidth performance is significantly better than netfront-netback performance, which in turn closely tracks the native inter-machine performance. XenLoop latency in this experiment also closely tracks the native loopback latency, whereas XenLoop bandwidth follows the native loopback trend at a smaller scale.

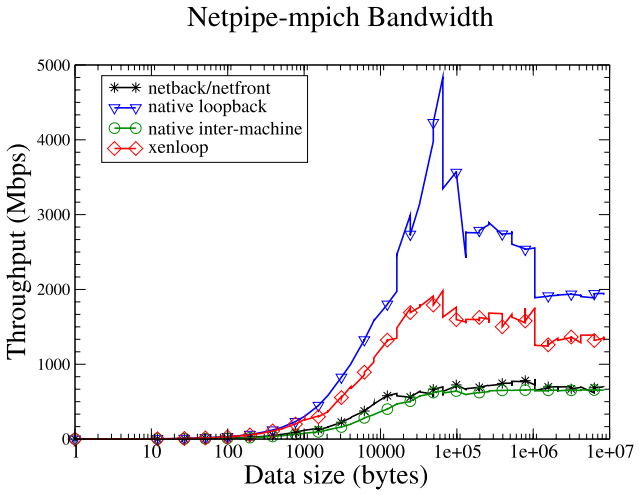


Fig. 6 Throughput versus message size for netpipe-mpich benchmark

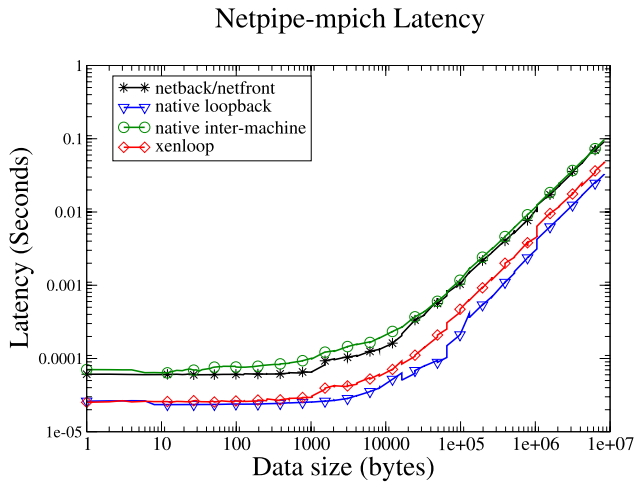


Fig. 7 Latency versus message size for netpipe-mpich benchmark

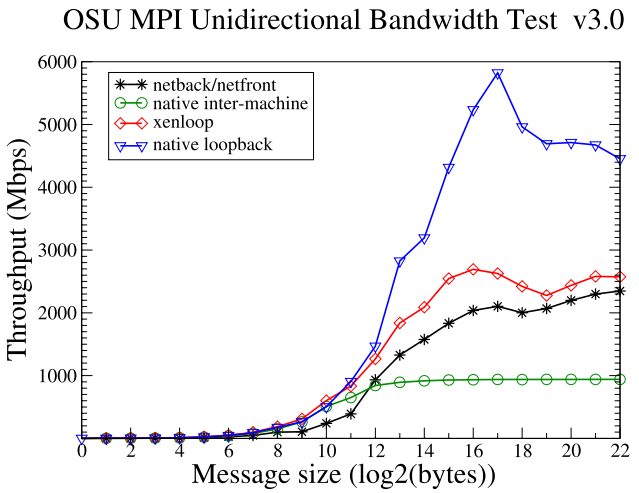


Fig. 8 Throughput versus message size for OSU MPI Uni-direction benchmark

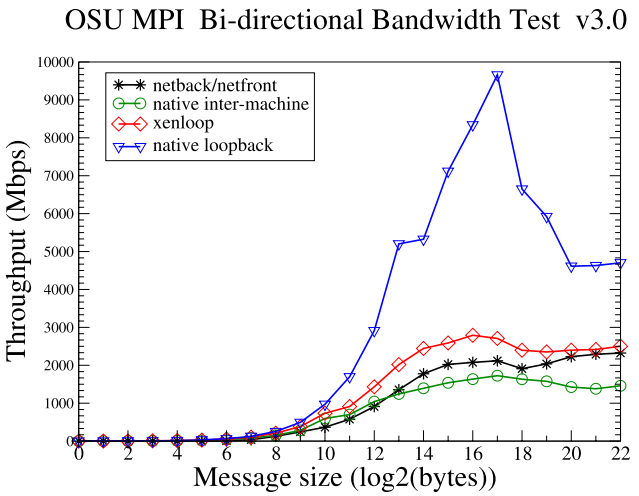


Fig. 9 Throughput versus message size for OSU MPI Bi-direction benchmark

4.4 OSU MPI benchmark performance

We also investigate the performance of XenLoop using the OSU MPI benchmark in the four communication scenarios. The bandwidth tests measures the maximum sustainable aggregate bandwidth by two nodes. The bidirectional bandwidth test is similar to the unidirectional bandwidth test, except that both the nodes send out a fixed number of back-to-back messages and wait for the response. Figure 8 plots the one-way bandwidth variation with increasing message sizes, Fig. 9 plots the two-way bandwidth variation, and Fig. 10 plots the latency variation. These results show the same performance trend as the previous MPICH test. We can see that XenLoop does much better job than native inter-machine and netfront-netback when the message size is smaller than 8192. This can be understood by observing that large-sized messages fill the FIFO very quickly and subsequent messages have to wait for the receiver to consume the older ones.

4.5 VM migration performance

In this section, we demonstrate that after migration guest VMs can dynamically switch between using XenLoop channel and the standard network interface depending on whether they reside on the same or different machines respectively. In our experiment, originally the two VMs are on different machines, then one of them migrates and becomes co-resident on the same machine as the second VM, and then again migrates away to another machine. During the migration, we run the netperf request-response latency benchmark. Figure 11 shows the number of TCP request-response transactions per second observed between two

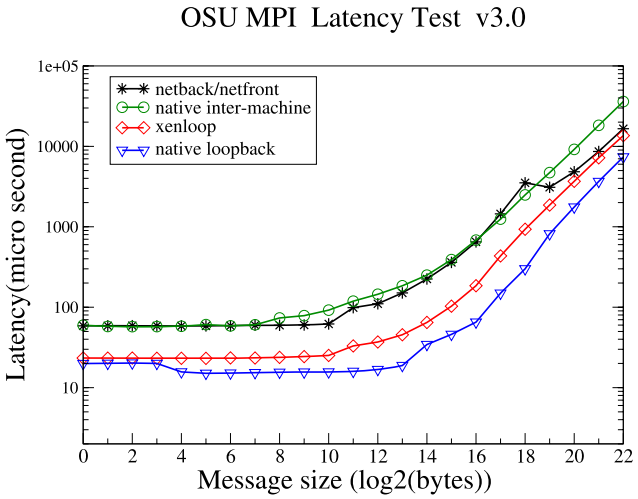


Fig. 10 Latency versus message size for OSU MPI Latency benchmark

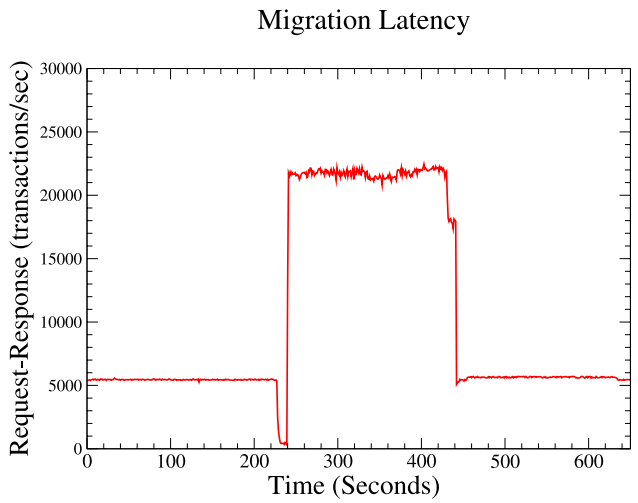


Fig. 11 Request response transactions/sec during migration

guest VMs during this migration process. Originally, the two guests have a transaction rate of about 5500 transactions/sec on separate machines, which translates to and average round trip latency of 182 µs. Then, as the VMs migrate together, detect that they are co-resident, and establish the XenLoop channel, the transaction rate increases to about 21000 transactions/sec, i.e. the average latency drops to 47.6 µs. The reverse is observed once the VMs separate again.

5 Related work

Most state-of-the-art inter-VM communication mechanisms provide either transparency or performance, but not both. As mentioned earlier, the Xen [1] platform enables applications to transparently communicate across the VM boundary using standard TCP/IP network sockets. However, all network traffic from the sender VM to the receiver VM is redirected via the netfront-netback interface with Dom0, resulting in significant performance penalty. There have been recent efforts [8, 9] to improve the performance of the standard netback-netfront datapath in Xen, though not targeted towards co-resident VM communication in particular.

Prior research efforts, namely XenSockets [18], IVC [4], and XWay [5] have exploited the facility of inter-domain shared memory provided by the Xen hypervisor, which is more efficient than traversing the network communication path via Dom0. In all these approaches, however, inter-VM communication performance is improved at the expense of some user or kernel-level transparency.

XenSockets [18] is a one-way communication pipe between two VMs based on shared memory. It defines a new socket type, with associated connection establishment and read-write system calls that provide interface to the underlying inter-VM shared memory communication mechanism. User applications and libraries need to be modified to explicitly invoke these calls. In the absence of support for automatic discovery and migration, XenSockets is primarily used for applications that are already aware of the co-location of the other VM endpoint on the same physical machine, and which do not expect to be migrated. XenSockets is particularly intended for applications that are high-throughput distributed stream systems, in which latency requirement are relaxed, and that can perform batching at the receiver side.

IVC [4] is a user level communication library intended for message passing HPC applications that provides shared memory communication across co-resident VMs on a physical machine. Unlike XenSockets, IVC does not define a new socket type, but provides a socket-style user-API using which an IVC aware application or library can be (re)written. VM migration is supported, 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. Authors also use IVC to write a VM-aware MPI library called MVAPICH2-ivc to support message passing HPC applications. IVC is beneficial for HPC applications that are linked against MVAPICH2-ivc or that can be modified to explicitly use the IVC API.

XWay [5] provides transparent inter-VM communication for TCP oriented applications by intercepting TCP socket calls beneath the socket layer. Available information indicates that VM migration is a work-in-progress as of date and there is no support for automatic discovery of co-resident VMs. XWay also requires extensive 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 messages above TCP layer.

In other application specific areas, XenFS [16] improves file system performance through inter-VM cache sharing.

HyperSpector [6] permits secure intrusion detection via inter-VM communication. Prose [3] employs shared buffers for low-latency IPC in a hybrid microkernel-VM environment. Proper [10] describes techniques to allow multiple PlanetLab services to cooperate with each other.

6 Conclusions

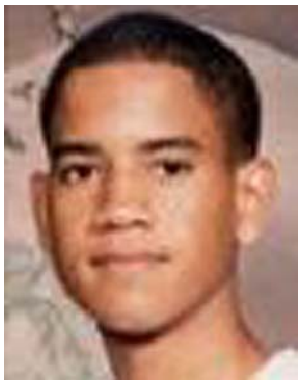
There is a growing trend toward using virtualization to enforce isolation and security among multiple cooperating components of complex distributed applications. Such application arise in high performance computing, enterprise, as well as desktop settings. This makes it imperative for the underlying virtualization technologies to enable high performance communication among these isolated components, while simultaneously maintaining application transparency. In this paper, we presented the design and implementation of a fully transparent and high performance inter-VM network loopback channel, called XenLoop, that preserves user-level transparency and yet delivers high communication performance across co-resident guest VMs. XenLoop couples shared memory based inter domain communication with transparent traffic interception beneath the network layer and a soft-state domain discovery mechanism to satisfy the twin objectives of both performance and transparency. XenLoop permits guest VMs to migrate transparently across machines while seamlessly switching between the standard network data path and the high-speed XenLoop channel. Evaluation using a number of unmodified benchmarks demonstrates a significant reduction in inter-VM round trip latency and increase in communication throughput. As part of future enhancements, we are presently investigating whether XenLoop functionality can implemented transparently between the socket and transport layers in the protocol stack, instead of below the network layer, without modifying the core operating system code or user applications. This can potentially lead to elimination of network protocol processing overhead from the inter-VM data path.

Acknowledgement We'd like to thank Suzanne McIntosh and Catherine Zhang from IBM Research for helpful interactions and discussions regarding their implementation of XenSockets [18].

References

1. Barham, P., Dragovic, B., Fraser, K., Hand, S. et al.: Xen and the art of virtualization. In: SOSP, Oct. 2003
2. Chisnall, D.: The Definitive Guide to the Xen Hypervisor, 2nd edn. Prentice-Hall, Englewood Cliffs (2007)
3. Hensbergen, E.V., Goss, K.: Prose i/o. In: First International Conference on Plan 9, Madrid, Spain, 2006
4. Huang, W., Koop, M., Gao, Q., Panda, D.K.: Virtual machine aware communication libraries for high performance computing. In: Proc. of SuperComputing (SC'07), Reno, NV, Nov. 2007
5. Kim, K., Kim, C., Jung, S.-I., Shin, H., Kim, J.-S.: Inter-domain socket communications supporting high performance and full binary compatibility on Xen. In: Proc. of Virtual Execution Environments, 2008
6. Kourai, K., Chiba, S.: HyperSpector: Virtual distributed monitoring environments for secure intrusion detection. In: Proc. of Virtual Execution Environments, 2005
7. McVoy, L., Staelin, C.: lmbench: portable tools for performance analysis. In: Proc. of USENIX Annual Technical Symposium, 1996
8. Menon, A., Cox, A.L., Zwaenepoel, W.: Optimizing network virtualization in Xen. In: Proc. of USENIX Annual Technical Conference, 2006
9. Menon, A., Santos, J.R., Turner, Y., Janakiraman, G.J., Zwaenepoel, W.: Diagnosing performance overheads in the Xen virtual machine environment. In: Proc. of Virtual Execution Environments, 2005
10. Muir, S., Peterson, L., Fiuczynski, M., Cappos, J., Hartman, J.: Proper: privileged operations in a virtualised system environment. In: USENIX Annual Technical Conference, 2005
11. Netfilter. http://www.netfilter.org/
12. Netperf. http://www.netperf.org/
13. Snell, Q.O., Mikler, A.R., Gustafson, J.L.: NetPIPE: a network protocol independent performance evaluator. In: Proc. of IASTED International Conference on Intelligent Information Management and Systems, 1996
14. Turner, D., Chen, X.: Protocol-dependent message-passing performance on Linux clusters. In: Proc. of Cluster Computing, 2002
15. Turner, D., Oline, A., Chen, X., Benjegerdes, T.: Integrating new capabilities into NetPIPE. In: Proc. of 10th European PVM/MPI conference, Venice, Italy, 2003
16. XenFS. http://wiki.xensource.com/xenwiki/XenFS
17. XenLoop Source Code. http://osnet.cs.binghamton.edu/projects/xenloop.html
18. Zhang, X., McIntosh, S., Rohatgi, P., Griffin, J.L.: Xensocket: a high-throughput interdomain transport for virtual machines. In: Proc. of Middleware, 2007

Jian Wang graduated from Beijing University of Posts and Telecommunications with an M.S. in Computer Science and worked at Sun Microsystems before joining the Ph.D. program at Binghamton University. His current research work is on inter-VM communication mechanisms in virtual machines.



Kwame-Lante Wright is an Electrical Engineering student at The Cooper Union for the Advancement of Science and Art. He joined the XenLoop project through a Research Experience for Undergraduates (REU) program hosted by Binghamton University and funded by the National Science Foundation (NSF).

Kartik Gopalan is an Assistant Professor in Computer Science at Binghamton University. He received his Ph.D. in Computer Science from Stony Brook University (2003), M.S. in Computer Science from Indian Institute of Technology at Chennai (1996), and B.E. in Computer Engineering from Delhi Institute of Technology (1994). His current research interests include Resource Virtualization, Wireless Networks, and Real-Time Systems.