Netbus: A Transparent Mechanism for Remote Device Access in Virtualized Systems

Sanjay Kumar, Sandip Agarwala, Karsten Schwan
College of Computing, Georgia Institute of Technology
Atlanta, GA 30332
{*ksanjay, sandip, schwan*}*@cc.gatech.edu*

Abstract

Efficient and seamless access to local as well as remote devices is a desirable property in multiple settings, including blade-servers, datacenters, enterprises, and even in home-based, personal computing environments. New virtualization technologies developed for PC and server platforms are now making it possible to implement remote device access at a level of abstraction transparent to operating systems and their device drivers. This paper presents a new mechanism for transparent device remoting, resulting in a hypervisor-level abstraction termed *Netbus*. The Netbus software solution provides both (1) efficient and reliable access to networked devices, and (2) remote access to devices not directly attached to networks, an example being a disk locally present on a bladeserver node. Netbus-based device remoting also supports *virtual device migration*, *device hotswapping* and efficient device sharing. A Xen-based prototype implementation of Netbus demonstrates transparent device remoting for block and for USB devices, for both bulk and isochronous USB access methods. Within the same administrative domain, seamless access to these devices is maintained during VM migration and during device hotswapping. Experimental evaluations with microbenchmarks and with representative server applications exhibit comparable performance for Netbus-based remote vs. local devices.

1 Introduction

System-level virtualization technologies [16, 7] are becoming increasingly important, as evident from recent hardware support integrated into processor architectures like Intel's VT [4] and AMD's Pacifica [1] technologies. Virtual Machine Monitors (VMMs) (e.g., Xen [16] and VMWare [7]) support the creation and execution of multiple virtual machines (VMs) on the same platform, and they enforce the isolation properties necessary to make the underlying shared platform resources appear exclusive to each VM. Toward these ends, VMMs export virtual instances of physical resources to VMs and they offer secure methods for sharing them. For I/O devices, such methods include time-sharing, space-sharing, and exclusive use.

A common reason for virtualization is server consolidation in datacenters or cluster systems. A prerequisite for server consolidation is the efficient and dynamic migration of Virtual Machines [14]. A key requirement for efficient VM migration, that is, for VM migration without downtime for the services being run, is continuous and transparent access to its virtualized I/O devices. Unfortunately, current methods for I/O virtualization do not provide such continuity and transparency for locally attached devices. Instead, they must rely on other technologies to provide them, such as hardware-based methods like network attached disks.

This paper argues the importance of VMM-level support for continuity and transparency in accessing remote devices. By enabling seamless VM migration, as shown in Section 7.7, such support will make it easier to develop the load balancing methods envisioned for next generation datacenters. In blade servers, for example, it will permit a single PCI device to be shared transparently across multiple machines (i.e., software-based, multi-hosted PCI). This makes it possible to dynamically extend virtual machines across additional blades to better scale to new application demands [8]. More generally, it provides end user applications running on virtual machines with richer choices in device usage. A concrete example is the ability to play a movie or do backup on a thin client, from a DVD drive on a nearby machine, or to do so on a personal laptop without having to physically unplug and re-plug the drive. Similarly, a video IPod can allow remote access to its movie files via a simple networked 'docking station' running a thin client.

VMM-level support for access to remote devices provides complete *transparency* in accessing remote vs. lo-

cal devices, at a level of abstraction not visible to guest operating systems. Specifically, since the hypervisors or VMMs that control the hardware platform already virtualize the platform's physical resources, it becomes possible to extend their per-platform methods for device virtualization into *remoting* methods that make the physical locations of devices entirely transparent to guest operating systems.

This paper demonstrates the feasibility and efficiency of transparent device remoting (TDR). Toward that end, it describes and evaluates a new abstraction termed *Netbus*, implemented for the Xen hypervisor and evaluated on a cluster machine. Netbus offers the following additional degrees of flexibility to virtual machines and applications:

- *Transparency* – Netbus makes it possible to access remote devices without changes to guest operating systems or device drivers. Stated differently, devices appear on a remote machine as if they were present locally.

- *Seamless migration* – Netbus enables the seamless migration of virtual devices along with virtual machines, with protocols less complex than those being used now (e.g., without having to explicitly detach and attach devices).

- *Transparent remoting* – Netbus permits virtual machines to be migrated across potentially heterogeneous platforms, without new hardware solutions like those being considered for bladecenters [20] or like those earlier implemented for datacenter environments [3].

- *High performance* – for strongly networked systems, like those found in datacenters or even in offices and homes, Netbus offers levels of performance for remote devices similar to those seen for local devices, in terms of realized device bandwidths.

A concrete demonstration of these claims is a *virtual device migration* mechanism built using Netbus. The mechanism not only enables a VM to continuously access its IO devices, including remotely after migration, but also, to seamlessly hot-swap devices, to replace remote with local devices whenever indicated or necessary, without any noticiable downtime.

Netbus is creating opportunities for further improvements in virtualization technologies. At base level, it makes it easy for developers to associate remote devices with guest operating systems. This is done by enriching Xenbus, the low-level cross-domain communication facility present in the Xen open source hypervisor More interestingly, Netbus may be used to layer additional functionality on top of generic remote devices, so that different machines can have different views of such devices, similar to 'soft devices' [26]. An example is the association of functionality that establishes dynamic trust characteristics for remotely accessed devices [23]. More aggressive examples concern runtime device enhancements like dynamic function placement for QoS in data intensive computing [9]. Finally, our experimental evaluation of Netbus demonstrates the feasibility of providing remote access to devices and shows the benefits of virtual device migration and device-hotswapping.

Netbus currently targets LAN-centric environments with single administrative domain that offer strong connectivity, high levels of cross-machine network bandwidth and low network latency. Further, additional work and hardware support [2] are needed to create an implementation that can operate across multiple administrative domains. Stated more explicitly, the current Netbus implementation ignores issues like trust management (i.e., all participating machines trust each other), network timeouts and admission control failures.

The Netbus prototype described in this paper relies on the frontend/backend device driver solutions used in Xen [19] (termed Xenbus) and in other hypervisor implementations. Specifically, Netbus extends Xenbus to enable communication to devices located on different physical machines. It does so by 'rewiring' backend device drivers to reach out to remote machines and their backend device drivers rather than directly accessing local devices.

We summarize by briefly outlining this paper's contributions: (1) the Netbus architecture and its Xen implementation constitute a simple approach to transparently access remote devices. (2) Evaluations with both remote disk and remote USB devices demonstrate that this approach offers levels of performance comparable to those of existing non-transparent kernel level solutions. (3) Finally, a key advantage of transparency in device access is the ability to seamlessly and efficiently migrate virtual devices along with VMs across different machines, even for machines with insufficient local device configurations. When required, device hot-swapping may be used after VM migration, to realize fault containment and/or reduce network dependence.

2 Related Work

There are many options for accessing remote devices. At user-level, file-based access to remote disks is provided by network file systems like NFS [24]. Similarly, web services may be used to access remote devices. One drawback of such application-level solutions is their typically higher latency of device access compared to device-level solutions like Netbus. Another issue with these so-

lutions in virtualized environments is their inability to continue to operate with VM migration.

Network Block Devices (NBD) [1] and DRBD [2] provide access to remote devices at the block level. Without additional support, however, NBD-based servers also cannot be migrated. In comparison, the Netbus server is tightly coupled with the underlying virtualization layer, running in a special privileged VM that will never be migrated to other machines. The same is not true for solutions like NFS or NBD, because they run as application or kernel-level services in guest VMs.

Typically, remote access services use the network stack in the corresponding OS to access remote devices. This dependency is not shared by our Netbus solution, since Netbus relies only (1) on hypervisor-provided inter-domain communications on the same platform and (2) on network stacks present in service VMs' driver domains which are not subject to the same failures or attacks experienced by guests.

Hardware techniques like RDMA-enabled NICs can be used to assist most software solutions, including Netbus, by their ability to directly and efficiently access memory on remote machines. Infiniband technologies can support remote devices on their interconnects with low latency, high bandwidth communication. In fact, Netbus can utilize these high-end interconnects when provided by the underlying hardware platform.

Datacenters commonly use network attached storage (NAS) and Storage Area Networks (e.g., Fiber Channel, iSCSI) to access remote disk devices. While these solutions provide adequate support for networked disks with their additional hardware, there are also some disks that are inherently not networked. For example, modern bladeservers use local disks (which may contain important data like OS binaries) on their blades, in addition to accessing networked disk devices. NAS and SAN solutions, therefore, cannot provide access to such disks remotely after VM migration. Netbus, in comparison, requires no additional hardware support and can deal with arbitrary non-networked remote devices.

Netbus architecture is similar to methods like USB/IP [18], which enables remote access to USB devices. One view of Netbus, in fact, is that it generalizes USB/IP to work with arbitrary devices that operate in virtualized systems and can support VM migration.

3 Platform Overview

As stated earlier, Netbus targets computing environments comprised of sets of machines with attached peripheral devices connected via a high bandwidth, low latency, reliable network, operating in a single administrative domain. Examples are clusters of workstations or server systems connected via gigabit Ethernet, blade servers with internal high performance system area networks, datacenters comprised of many such systems and networks, office or home environments with multiple computers and IO devices connected via a LAN. When used in such systems, Netbus affords administrators with substantial flexibility in how to structure or configure their systems. In blade servers, for example, some cabinets may be configured to be *device-less*, coupled with disk-heavy cabinets elsewhere. This can reduce per blade costs without limiting configuration flexibility in terms of where certain virtual machines may be run. An advantage in datacenter settings is the ability to upgrade to new hardware while still be able to use older devices on machines left over from a prior configuration, with potential performance penalties, but without having to use costly solutions for device interoperability like NAS, iSCSI etc. However, the current implementation of Netbus does not support *outsourced* solutions, since those may require dealing with multiple administrative domains or even with online trust management for remote machines and devices [22].

[1] http://nbd.sourceforge.net/
[2] http://www.drbd.org/

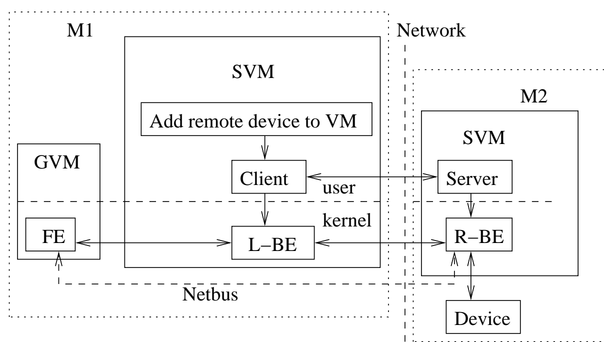

Figure 1: Netbus Software Architecture

4 Netbus Software Architecture

The Netbus abstraction is similar to that of channels in publish/subscribe systems [17], where a device offering remote services registers itself with Netbus and guest VMs desiring access subscribe to that registration. For performance reasons, however, Netbus then creates dedicated one-to-one connections between subscribers and registrants. So, in a case where a single remote device is shared by three guest VMs, there will be three different point-to-point connections between device and guests. The intent is, of course, to use Netbus to 'extend' the implementations of device interfaces for transparent access to remote devices. To understand how this can be

implemented, we next briefly digress to explain current systems' methods of I/O device virtualization.

Device virtualization typically involves running two stacks of device drivers: one in the guest VM (GVM), termed Frontend (FE) and another, termed Backend (BE), running in the VMM or more likely, in a special privileged VM, called a Service VM (SVM). SVM provides the management utilities and device drivers to the VMM. In fully virtualized systems, the GVM runs the entire device driver stack, while in the para-virtualized case, the driver stack is split: the GVM runs only the upper part of the split device driver stack, and the SVM runs the lower part of the split stack. Examples of both implementations are VMWare workstation [25, 21] and Xen [16] respectively. Netbus' current implementation uses Xen's para-virtualized system, but there are no inherent reasons why it cannot also be implemented for fully-virtualizing VMMs.

In para-virtualized systems, I/O virtualization involves two steps: (1) the export of virtual device interfaces by the BE driver in the SVM and (2) the use of these virtual interfaces by the GVM's FE driver. Current VMMs export these interfaces only to VMs running on the same host, but since these are virtual interfaces, there is no inherent reason to not also export these interfaces to remote VMs. This fact is exploited by the Netbus architecture, which uses a simple implementation of the publish/subscribe paradigm to export and import device interfaces across participating domains.

The following exposition of Netbus assumes a VMM using the split device driver stacks, as described above. FE and BE communicate with each other using VMM-provided inter-domain communication channels. The FE driver takes requests from the drivers in GVM and forwards them to BE. The BE forwards these requests to the IO device. The BE also notifies the FE about device events (connect, disconnect, interrupts etc.) and does multiplexing-demultiplexing between requests from multiple FEs (present in multiple VMs).

The software architecture of Netbus for FE/BE-based VMM implementations is depicted in Figure 1. It consists of utility applications (Netbus client and server) for control operations and of FE and BE drivers that communicate over the network to access the devices for which they are designed. For this configuration, Netbus extends the existing inter-domain communication channels with the knowledge that a device being accessed is potentially remote. This knowledge may be encoded in the VM configuration during VM creation, or it may be added later (e.g., for plug and play devices). In either case, to initialize a remote device, the Netbus server running in the SVM on host M2 having the device, exports it (i.e., publishes it) across the network to a designated host or a set of hosts if the device is to be shared. When a device is being added to guest VM G1 running on host M1, if the device is specified as remote, the Netbus client in SVM establishes a connection with the Netbus server and executes required authentication and authorization actions. When those succeed, client and server both inform their respective BEs about this connection information and henceforth, all communications between the BE drivers on both machines are carried out via this one-to-one connection. The BE makes the virtual device appear inside guest VM where it gets initialized by its respective driver. When the FE from the guest VM accesses the device by making requests to its corresponding local BE (LBE) driver, the LBE forwards this request to the remote host's BE (RBE). The RBE then makes the actual request to the device and returns the response to the LBE. The LBE in turn returns the response to the FE. The RBE also provides asynchronous device events (e.g., interrupts) to the LBE which forwards them to the FE. This mechanism effectively provides an abstraction of a *network bus* over which the FE can interact with a Remote BE (RBE).

Important attributes of this architecture include the following. First, although FE drivers are para-virtualized to enable them to talk to BE drivers, the Netbus architecture itself does not require FE drivers to be changed. Second, while the above description assumes a para-virtualized FE-BE mechanism for I/O device virtualization, the Netbus architecture itself is not restricted to such an environment. It can also be applied to a fully-virtualized case, where software I/O device emulation is used to provide virtual devices to guest VMs.

4.1 Implementation Issues

As evident from the experiments described in Section 7, even the simple implementation of Netbus described here has been shown to be useful for both disk and USB devices. We next discuss several implementation issues, addressed by our current work or to be addressed in the future.

Connection and Communication Protocols:- As with Xenbus, Netbus shares pages between the FE and BE wherever facilitated by the underlying VMM, to reduce unnecessary data copying. While Netbus connections are established by client and server at application level, the connection information is passed down to the kernel-level LBE and RBE drivers. Thus, the data fast path is entirely in the kernel, and potential context switches between user and kernel levels are avoided.

The Netbus architecture is independent of the underlying network infrastructure. Its current implementation and evaluation use the TCP/IP protocol for cross-machine communication, but we are next experimenting with modern low latency interconnects, namely RDMA

and Infiniband using Sockets Direct Protocol (SDP).

Latency and Bandwidth:- Since network communications introduce additional latency in accessing remote devices, a guest driver may timeout on pending device requests. Such cases may require driver modification, but we have not observed them for the devices we have evaluated. These include bulk devices like IDE and USB disks and isochronous devices like USB cameras. Latency issues may also arise for devices used in sense-respond or real-time loops.

A more important issue is that isochronous devices reserve bandwidth prior to their operation and then they require such reservations to be honored. In such cases, insufficient network bandwidth may cause device malfunction. This is not an issue in strongly networked environments like blade servers, datacenters, or wired homes, but it is not clear to what extent solutions like Netbus generalize to wide area or wireless infrastructures. To better understand device operation under constrained network bandwidths, Section 7 of this paper determines the network conditions suitable for Netbus-based remote device access and operation. Here, we simply note that for the devices evaluated in our research, mechanisms like buffer caching [15] tend to reduce network bandwidth needs. Furthermore, future works on virtualized networks will likely address issues like congestion and isolation in the network.

Device Naming:- Netbus combines IP and device addresses to create unique device identifiers. More general naming schemes, like those based on device location [5] or the nature or content of the data produced by devices is an interesting topic for future work.

Device Discovery:- The current Netbus implementation discovers remote devices by having Netbus clients explicitly query administrator specified Netbus servers for their available devices. A more scalable implementation would use distributed discovery and directory services akin to the active directories developed in our own prior work [11].

Device Sharing, Failure, and Recovery:- Sharing remote devices is desirable for several reasons. For disks, for instance, device sharing can improve total throughput. For a camera device, sharing enables multiple VMs and hence, applications, to access the same camera images, a concrete example being a shared security camera.

Netbus supports sharing for the remote devices that permit it. Specifically, its RBE is designed to communicate with multiple LBEs and can multiplex and demultiplex access to the device by these VMs. Netbus checks for consistency across the access modes of different VMs sharing the device. For example, a disk partition can be shared only in read-only mode. Interesting future work concerns admission control for device sharing like those used for real-time systems [27], followed by the enforcement of desired QoS properties.

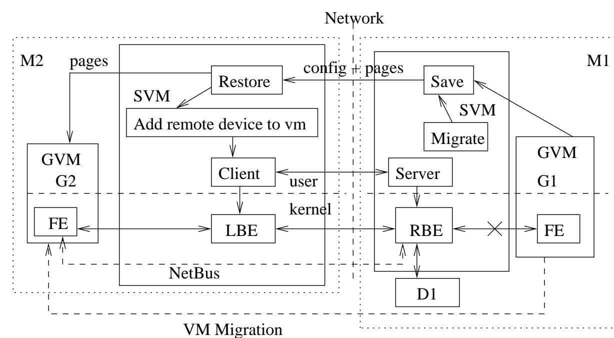

Figure 2: Virtual Device Migration

As stated earlier, Netbus does not currently distinguish device failures from network failures. This can cause problems when device drivers respond to failures with inappropriate handler actions. An example is a driver issuing a camera reset when the actual problem is network disconnection. The solution approach currently being investigated by us is one that associates multiple failure attributes with Netbus-based remote device accesses, then have successive drivers (i.e., first the RBE, then the GVM's driver) process these attributes. One interesting set of functionality requiring online failure handling is device hotswapping, for which initial results are presented in Section 7.

Residual Dependencies and Security:- Permitting VMs to access remote devices can lead to residual dependencies, and it can raise security concerns. Security can be addressed through authentication and authorization protocols during connection establishment, coupled with data encryption for remote device accesses. The viability of these approaches is apparent from the fact that off-chip encryption is already used in current server system hardware and supported by powerful crypto chips integrated into modern PC platforms like Intel's LT technology. Current research performed by our group is going beyond such static approaches to security, by developing online methods for trust monitoring and management [23]. By exploiting future multi- or many-core machines, methods like these can be used to continuously enhance Netbus to operate in infrastructures with dynamic levels of trust.

Netbus does not automatically deal with residual dependencies caused by VM migration. In fact, likely, such methods require semantic information beyond what is accessible at the low system levels at which Netbus operates. Instead, Netbus offers simple mechanisms that permit higher system levels to deal with such dependencies.

5 Using Virtual Device Migration

Netbus enables live VM migration [14] of GVMs, with continuous access to its devices without the need for special hardware solutions (e.g., iSCSI). This mechanism is termed as *virtual device migration* since logically the virtual device is also migrated along with the VM. It provides seamless access to a device throughout a VM's migration, e.g., to a local device before migration and the (then) remote device after migration.

5.1 Basic Mechanism

VM migration typically involves multiple steps, including freezing a VM, destroying its interfaces with the current VMM, creating a new VM on the destination machine, filling this VM's memory with the memory pages of the frozen VM, and finally, unfreezing the new VM. Live VM migration is a special case in which the freeze duration of the VM is kept very small. The method for live migration used in Xen3.0 is described in [14].

Virtual Device Migration is depicted in Figure 2. Assume that a guest VM named G1 is migrating from host machine M1 to host machine M2, while G1 is accessing a device D1 on M1. During VM migration, G1 is frozen and the FE-BE communication is suspended. During this suspension, BE (RBE) breaks its connection with the FE, but it does not break its connection with the device. On the destination host M2, the *restore* process creates a new VM G2, and its OS pages are filled from the suspended VM G1. G2 uses G1's configuration so that it exactly looks like G1. During G2's creation, however, its configuration is modified with respect to device D1, so that D1 appears as a remote device in G2. In this fashion, Netbus ensures the establishment of a valid communication channel between the two BEs (LBE and RBE). Next, just before the migrated VM G2 is un-paused on host M2, its FE establishes a connection with the local backend (LBE). The effect of this action is that the BE (RBE) on M1 connects to the LBE on M2, which is the new backend driver of the migrated VM. At this point, G2 is un-paused and migration completes, and communication between FE and LBE on M2 resumes. While all subsequent accesses to device D1 use Netbus, the entire process of virtual device migration is transparent to the guest VM.

5.2 Pending IO Transactions

A potential issue for virtual device migration is that there may be pending IO transactions at the time G1 is suspended. More precisely, there may be pending IO transactions in BE submitted by G1's FE. By the time these IO transactions complete, the VM has already migrated and the BE can't return the IO results to the FE. Transparency demands that we deal with these pending transactions. Several approaches are possible:

- **Discard pending IO transactions**– the BE driver on M1 can discard these IO transactions. This approach relies on the error recovery mechanism in the FE and in its upper level drivers. After migration, eventually the FE will realize the failure of the pending IO transactions and as part of error recovery, its retry of those IO operations will eventually succeed. While suitable for devices with highly resilient protocol stacks, e.g. the NIC device, the drawback of this approach is its reliance on robust error recovery in guest device drivers.

- **Bringing the device into a quiescent state before migration** – In this approach the virtual device of the migrating VM is brought into a state where there are no pending IO operations to the backend driver, termed the *quiescent* state. To do this, during the suspend operation, the communication channel from FE to BE is suspended (but not in the other direction), so that the FE ceases to make additional requests to the BE, queuing them instead. In addition, suspend waits until all pending I/O operations are complete. At this point, the virtual device is in a quiescent state and can be migrated. During the resume operation on M2, FE first issues the queued requests to BE before resuming normal operation. A potential drawback of this approach is that the FE must wait for all pending transactions to complete and cannot make further requests to the device during that period. Nevertheless, our experience with live VM and device migration has shown that the approach is viable, causing only marginal increases in migration time.

The current realization of virtual device migration permits VMs to continue to operate while being migrated, at the cost of 'remoting' its devices. As stated earlier, a necessary improvement in this approach is to 're-wire' devices after migration, particularly when migration is caused by the need to stop using some host (e.g., due to impending failure) or to take it offline for maintenance. Device hot-swapping, explained in the next section, can be used to remove such inter-machine dependencies.

5.3 Device Hot-Swapping

Device hot-swapping permits a VM to dynamically re-wire its local/remote device connections while the device is in operation. This is particularly useful when after migration, a VM wishes to switch from the remote to a

local device to improve device throughput, remove network dependence or to shutdown the remote host. Transparent hot-swapping, however, requires that a 'similar' device be present locally (e.g., a disk with the same content as the original disk), implying the need to properly deal with device contents and state. Not surprisingly, this also involves bringing the device into a quiescent state as described in the previous section, switching the LBE connection from the RBE to the local device, and finally, resuming device operation. For devices like NICs, frequent hot-swapping is reasonable due to their small internal states. For disks and similarly state-rich devices, hot-swapping is likely to remain infrequent.

6 Netbus Prototype

6.1 Netbus Implementation in Xen

Xen VMM virtualizes I/O devices by splitting the device stack at the 'class-driver' level. Special 'class-drivers' are used in both dom0 (SVM) and domU (GVM). The 'class-driver' in domU and dom0 are the frontend (FE) and backend (BE) respectively. Concrete examples of class-level drivers are block, network, USB, etc.

The Netbus server is runs on the machine having the local device that exports it. The connection information for the remote device provided by the Netbus server is specified in the VM configuration and passed down to the respective BE drivers (RBE and LBE) using the *xenstore* utility. This connection information is maintained in a structure associated with the Netbus extension of Xenbus. This structure also contains generic remote device state and any device-specific states the BE may require. Every Netbus communication over the network, then, is preceded by a Netbus header. Each such header has a common and a device-specific part. The common part contains information common to all devices, e.g., request-response id, device id, length of the data following the header, etc. Examples of the device-specific parts appear later in this section.

Driver actions are linked to Netbus communications by having drivers register device-specific callback functions with Netbus, upon their receipt of connection information from the Netbus client and server. The following function call is used for this purpose:

*void register_netbus_device(struct xenbus_device *xenbus_dev, u64_t devid, void *dev_context, netbus_connect_callback *connect, netbus_disconnect_callback *disconnect, netbus_receive_callback *receive, netbus_xmit_callback *xmit, netbus_hotswap_callback *hotswap);*

Netbus communications and the execution of LBE and RBE callbacks are carried out by kernel-level, per-device threads. The receive thread, for instance, will first perform some Netbus header checks on the header, e.g., matching a request response id pair, and then forward the actual data contained in each request to the device-specific callback function for further processing. We next explain in more detail several specific Netbus devices implemented in our work.

Remote Virtual Block Device Access

Xen virtualizes block devices using block BE and FE drivers. Applications make block requests to the FE, which forwards them to the LBE. The LBE also maps the relevant block pages into dom0. For read requests, the LBE transmit the requests to the RBE via the Netbus transmit thread. For write operations, the LBE additionally sends the blocks to be written. The receive thread on the remote host forwards the request to the RBE, which issues block I/O request to the device. When the request completes, it sends the result to LBE along with any blocks read.

Remote USB Device Access

Xen virtualizes USB devices at port granularity, using USB BE and FE drivers. The USB ports are assigned to guest VMs, and any device attached to a port belongs to the respective VM. When a device is attached to a USB port, the root hub driver notifies the corresponding BE (RBE), which notifies the guest FE about the attach event through LBE. The guest USB driver initializes the remote virtual device via Netbus. The USB drivers in the guest make USB requests using USB Request Blocks (URBs), which the FE forwards to the RBE via the LBE. For write operations, it also sends the transfer buffer. For isochronous devices, it additionally sends an isochronous schedule to RBE. Upon URB completion, the RBE sends the results back to FE via the LBE, along with any transfer buffer.

6.2 Virtual Device Migration in Xen

Virtual device migration operates alongside live VM migration in Xen. The Netbus server is started on the local host M1 before any VM migrations take place. During VM migration, the *save* process on M1 sends G1's configuration and memory pages to M2 and *restore* process on M2 modifies this configuration to create a new VM G2. G2's configuration is exactly like that of G1 except that G2's device configurations are modified to reflect the correct location of devices. Hence the underlying Netbus implementation establishes the necessary communication channels between M1's BE (RBE) and M2's BE (LBE) as described in Section 5.1.

Just before the VM G1 is frozen, Xenbus is suspended, and all communication links between FE and BE are broken. The device FE is also suspended by calling their

'suspend' callbacks. As a result, it starts queuing the I/O requests by guest VMs rather than forwarding them to the backend. The FE also waits until it receives results for all the pending IO transactions. After the FE returns, the device is in a quiescent state, and the VM is ready to migrate. The VM is frozen and its final chunk of memory pages is moved to the VM G2 on machine M2. Just before VM G2 is resumed, new Xenbus connection is established between the FE and BE on M2. The VM G2 is started, and the FE is resumed by calling the resume callback. This callback re-makes all queued requests (queued during suspended state in G1) to the new BE. The resume callback returns and VM migration completes.

6.3 Hot-swapping of block devices

Device hot-swapping is administrator-controlled. For the disk hot-swapping implemented in our work, after VM migration, a new disk partition of same size as the VM's current disk size is created on the destination machine, and the disk is replicated onto this partition over the network. During the replication period, the VM continues to access its original disk remotely. Since disk virtualization works at the block level, we use block level replication. Disk replication time depends on the size of the disk and is done in phases, similar to the pre-copy phase of live VM migration [14]. Intelligent disk replication techniques [6] can be used to reduce overheads, but these are not the focus of this paper.

A hot-swapping command is issued after the disk has been replicated. Upon such a command, the LBE puts the device into a quiescent state by queuing the FE requests instead of forwarding them to RBE. It also waits for all pending IO operations to complete. The last phase of block replication is completed to address any changes made by the IO operations that were pending at the start of the quiescent state. At this point, the hot-swap happens and the virtual device is released from its quiescent state. Henceforth, the LBE makes IO requests to its local disk instead of sending them to the RBE. The disk stays unavailable to the VM during its quiescent period.

7 Evaluation

In this section, we quantify the overheads of Netbus implementation in Xen, measure the effects of network congestion and device sharing on the throughput and demonstrate its benefits during VM migration. The experiments reported in this paper use two hosts. These are Dell PowerEdge 2650 machines connected with a 1 Gbps gigabit Ethernet switch. Both hosts are dual 2-way HT Intel Xeon (a total of 4 logical processors) 2.80GHz servers with 2GB RAM running Xen3.0. Dom0 and DomU both run a paravirtualized Linux 2.6.16 kernel with a RedHat Enterprise Linux 4 distribution. Dom0 runs a smp kernel while domU a uni-processor kernel. Both hosts have EHCI USB host controllers, and we use a USB flash disk and a USB camera to evaluate remote USB device access. We use the default Xen CPU allocation policy, under which the first hyperthreads of all CPUs are assigned to dom0, and domUs share the second hyperthreads of all CPUs.

Experiments are divided into six sections. The first section measures the average throughput overhead of Netbus operations. The next two sections report the throughput of Netbus for block and USB devices, compared to that of their respective local device accesses. Throughput is measured with the Iozone [3] file io benchmarks for ext3 file systems. For block devices, we also measure the throughput of NBD access. Comparisons with USB/IP's throughput are not reported, because this functionality is not yet operational in Xen. We also show the effects of device sharing on Netbus throughput. The fourth section reports the latency overhead in accessing the devices using Netbus. In the fifth section, we measure the throughput characteristics of a remote disk with respect to the congestion in the network. A final set of experiments shows the effects of virtual disk migration on the RUBiS enterprise application. VM migration is applied to its database server VM, and performance measurements show the effect of migration on RUBiS clients, during VM migration and after it has completed. We also demonstrate the throughput benefits of device hot-swapping.

7.1 Netbus Overhead

The inherent overhead incurred by the Netbus implementation is derived from the need to encapsulate I/O blocks with additional control information. This includes the network header (TCP/IP and Ethernet, 40 + 16) bytes and the Netbus header bytes. The latter differ depending on the device being accessed remotely, since it also contains some device-specific information. For Block devices, the Netbus header is comprised of:

28 (request) + 4 (response) = 32 bytes.

On average, one request transfers 4 blocks of data over the network, where block size is 4096 bytes. Hence the overhead for block devices is:

((no of packets * network overhead) + Netbus overhead)/(total useful data transferred) = (5 * 56 + 32) / (4 * 4096) = 1.904%

For USB devices, the Netbus header consists of:

[3] http://www.iozone.org/

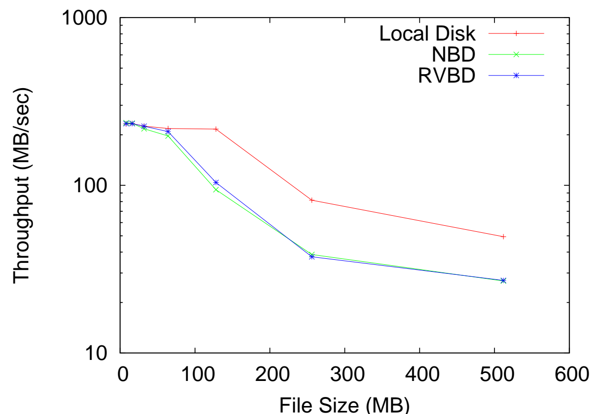

Figure 3: Write throughput of Block devices (record size=8MB)

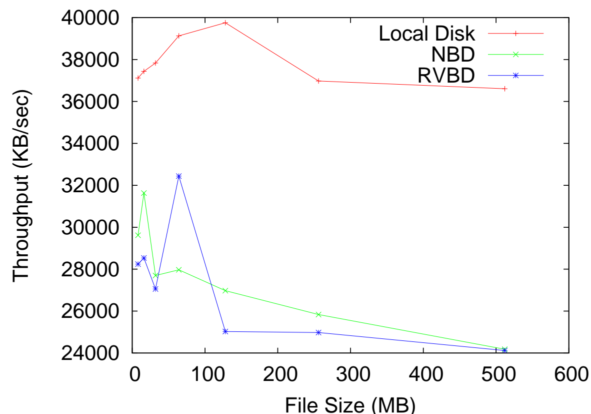

Figure 4: Write throughput of block devices w/o buffer caching (record size=8MB)

16 (request) + 16 (response) bytes.

USB devices transfer data in URB's transfer buffers. For isochronous transfers, there is an additional transfer of the isochronous schedule. Experiments with the USB bulk (disk) and isochronous (camera) has shown that on average, they transfer 3050 and 4400 bytes of data per URB request respectively. Hence the overhead for USB bulk devices is:

((no of packets * network overhead) + Netbus overhead)/(total useful data transferred) = (2 * 56 + 32) / (1 * 3050) = 4.721%

and the overhead for USB isochronous devices is

((no of packets * network overhead) + Netbus overhead)/(total useful data transferred) = (3 * 56 + 32) / (1 * 4400) = 4.545%

The straightforward computations in this section demonstrate that the additional bandwidth overheads incurred by Netbus encapsulation are quite low, never exceeding more than a few percent.

7.2 Block device performance

The I/O performance of Netbus primarily depends on available network bandwidth and on the manner in which devices exploit this bandwidth (e.g., with concurrent requests). To measure the throughput of remote virtual block devices (RVBDs), we booted a guest VM from a remote disk and ran Iozone benchmarks on it. The dom0 runs with 512 MB of ram while the guest VM runs with 256 MB. Read, write, buffered read and buffered write tests were performed. Iozone performs these tests for different file sizes and different record sizes. The results for the write tests are shown in Figure 3 for different file sizes with a record size of 8 MB. Throughput is shown on a logarithmic scale. The results for other tests and record sizes show similar patterns and for brevity, are not included here.

Throughput for file sizes less than 64MB is quite high, and it is similar for VBD, RVBD, and NBD. This is because of the buffer caching in the Linux kernel which caches the files in memory and so avoids disk access for subsequent file accesses. However, throughput sharply decreases for file sizes greater than 64 MB, because for bigger file sizes, the buffer cache cannot contain the file, which means that file contents must be pushed to the disk. Hence the actual disk I/O throughput starts to dominate. For the file size of 512 MB throughput of RVBD and NBD are similar, offering about 60% of the local VBD case.

Actual device I/O throughput without buffer caching are shown in Figure 4, on a normal scale. Throughput without caching is drastically lower than that with caching. The performance of NBD and RVBD remain similar for all file sizes, and show same relative performance to local VBD for other record sizes. For the record size of 8 MB, the throughput is about 65% of local VBD throughput. However, this drop in throughput can be significantly reduced by efficiently utilizing network bandwidth. The previous Iozone test did not issue sufficient number of simultaneous requests to maximize the disk throughput. Instead, it waits for requests to complete before issuing more. Hence, the additional request latency due to network delay causes throughput to drop significantly. To illustrate this point, we maximize the disk throughput for file size of 32 MB by increasing number of parallel Iozone executions. The result is shown is Figure 5. Clearly, as the number of parallel Iozone

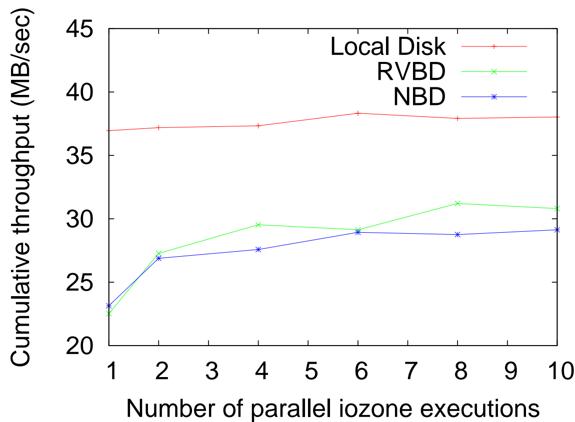

Figure 5: Effect on cumulative throughput of block devices with increasing number of Iozone executions

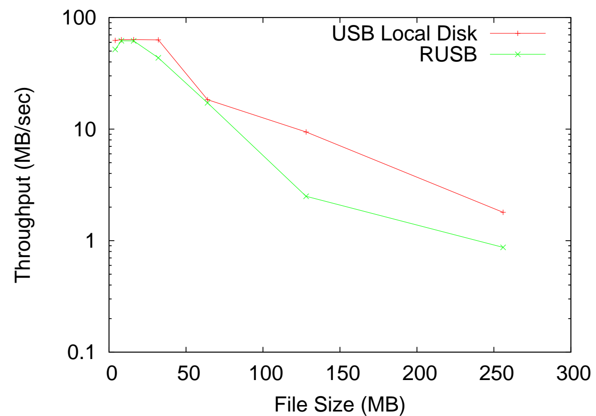

Figure 6: Write throughput of Netbus USB devices (record size=4MB)

executions increases, cumulative RVBD throughput increases and then saturates because it reaches the maximum Netbus throughput for block devices. Local VBD, however, does not show any significant increase because it has already reached its maximum throughput. Maximum throughput of RVBD is about 75% of the local VBD. The throughput of NBD is less than those of VBD and RVBD because the nbd-server runs at the application layer, as opposed to VBD and RVBD which run in the kernel. Hence RVBD can maximize its throughput if the applications better utilize the network bandwidth.

We conclude that RVDB offers performance comparable to that of local devices, in part due to optimization techniques like buffer caching. This clearly demonstrates the utility of the proposed Netbus solution to disk access. Furthermore, Netbus can replace NBD without significant performance penalties, thereby making it a software solution with which guest VMs can remain entirely unaware of device location. This is key to its utility for VM migration.

7.3 USB performance

Remote USB bulk device experiments use a USB flash disk with the same experiment setup as in the previous section. The USB disk (a 1GB Sandisk cruzer micro) is attached to the USB port of a machine, and it appears as a local USB disk to the guest VM on the other machine.

Figure 6 depicts the throughput of USB vs. RUSB disks, using a logarithmic scale. The results are from the Iozone benchmark's write test with a record size of 4 MB. Results are very similar to those seen for block devices. For small file sizes, throughput is similar for USB and RUSB disks because of buffer caching. For larger file sizes, throughput decreases sharply because of the actual device I/O. For the RUSB case, the decrease in throughput is higher. For the file size of 256 MB, RUSB throughput is approximately only 48% of USB throughput. Iozone throughput without buffer caching shows patterns similar to those of block devices (see Figure 4) and for brevity, they are not included here.

The decreased throughput seen in these experiments can again be remedied by utilizing network bandwidth more efficiently, as discussed in Section 7.2 by running multiple Iozone in parallel. The Linux USB storage driver does not support multiple outstanding URB requests. Instead, it issues an URB to the USB FE and waits for it to complete before issuing another one. To measure maximum throughput, therefore, we again run multiple simultaneous Iozone benchmarks. The results of using multiple Iozone are similar to those in Figure 5 and again, are not included here for brevity. To optimize sequential I/O throughput for RUSB bulk devices, a reasonable improvement is for the USB storage driver to provide support for multiple outstanding SCSI requests.

An alternate set of measurements demonstrate the generality of the Netbus solution. They measure the throughput of a USB camera as an example of an isochronous device operating over Netbus. We use a Logitech QuickCam web-camera and measure its throughput for both local and remote access. The results are shown in Figure 7. Since this camera only supports two small image sizes, its bandwidth requirements are modest. As a result, this isochronous device performs fairly well for Netbus compared to local access. The throughput for Netbus is about 83% of local throughput for image size 320x240 and about 90% for image size 640x480.

The differences in throughput reductions experienced by disks, USB bulk devices, and USB isochronous devices for Netbus access demonstrate the fact that actual

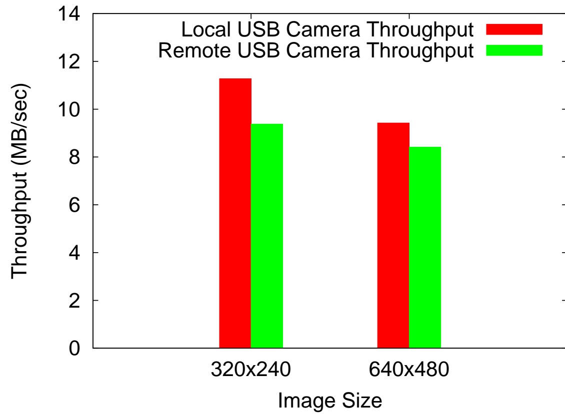

Figure 7: Throughput of remote USB camera using Netbus

Netbus performance depends on the combination of three principal factors: (1) network bandwidth, (2) the applications' and device drivers' ability to efficiently utilizing network bandwidth, and (3) actual device throughput.

7.4 Sharing of block devices

Sharing IO devices is encouraged by the Netbus solution, in part because such sharing will increase device throughput (limited by total network and device throughput). This section discusses throughput attained with a disk device shared between multiple VMs. The disk is shared in read-only mode, and Iozone test is run from the sharing guest VMs to measure maximum disk throughput without caching. Individual, per VM and cumulative disk throughput are measured as the number of VMs is changed.

Not surprisingly, IO device sharing tends to improve throughput. Further, the throughput characteristic of RVBD sharing is similar to the one shown in Figure 5 and is not shown here for brevity. The reason for this similarity is that running single Iozone from multiple guest VMs is loosely equivalent to running multiple Iozone from a single guest VM, particularly for an io-bound application like Iozone.

7.5 RVBD and RUSB latencies

This section contains detailed latency measurements when accessing remote vs. local devices. The purpose is to measure the latencies experiences in the various stages of accessing remote devices and then, determine techniques for decrease these latencies. To minimize TCP/IP buffering latency, we use the TCP NODELAY socket option.

	Frontend	Netbus	Backend	Total
Local VBD	1.485	n/a	7.036	8.521
RVBD	1.473	3.578	9.242	14.293
NBD	1.253	4.127	9.334	14.714

Table 1: Latency incurred by various components in accessing block devices (milliseconds)

	Frontend	Netbus	Backend	Total
Local USB	1.012	n/a	3.367	4.379
RUSB	1.473	3.778	3.056	8.307

Table 2: Latency incurred by various components in accessing USB devices (milliseconds)

End to end latency encompasses the entire time between the FE sending a request to the device and the FE receiving the device's response. We divide this latency into stages and calculate the time spent in every stage. In Xen, the stages are Frontend (time spent by the FE driver in GVM), Netbus (time spent by the Netbus processing and network propagation delay), and Backend (time spent by BE processing and actual block I/O (for block devices) or USB I/O (for USB devices)). For NBD, we have reported the time spent by the guest NBD driver under Frontend, NBD server under Backend, and network processing under Netbus. Every stage is measured individually, but we report the combined latency in both the forward (towards the device) and the backward (towards the FE) directions for each stage. Latency is measured using the sched_clock() function which gives synchronized time in the guest VM as well as in dom0. To remove the effects of large latencies due to infrequent system activity, we sample the latency for 100 transactions to the device and compute the average.

Table 1 shows the latencies incurred in various stages of accessing local VBD, RVBD, and NBD devices, respectively. The time spent in Frontend, Netbus, and Backend is approximately similar for RVBD and NBD devices since they follow similar code path. It is apparent from Table 1 that the increase in RVBD latency is due to additional time spent in the Netbus stage and the increased latency in Backend stage. The latter is due to the additional copying involved in transferring data over the network.

Table 2 shows the latency incurred in various stages of accessing local USB and RUSB devices respectively. The USB devices show similar behavior as the block devices in Table 1. The increase in IO latency for RUSB case is due to the Netbus processing and extra copying of data over the network.

Table 2 shows the latency incurred in various stages of accessing local USB and RUSB devices, respectively. The USB devices show similar behavior as the block

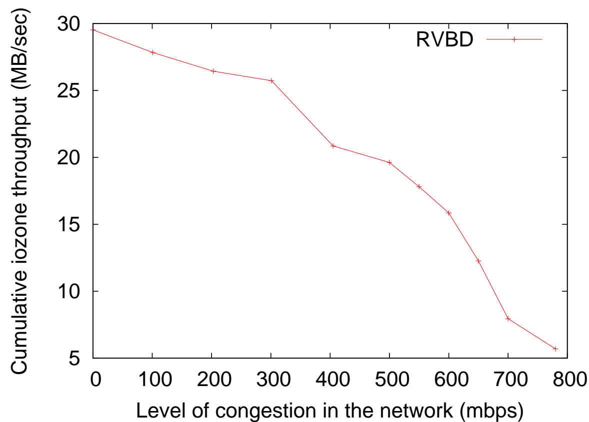

Figure 8: Effect of network congestion on RVBD throughput

devices in Table 1. The increase in IO latency for the RUSB case is again due to Netbus processing and the extra copying of data over the network.

We see that for both block and USB devices, the majority of IO access latency is incurred in servicing an actual IO request by the device and in case of remote devices, by the network (protocol stack processing and propagation) delay. The time taken by the device is characteristic of the device and cannot be improved by software solutions. The latency induced by the network, however, can be reduced by using low latency communication technologies like RDMA, Infiniband, etc.

7.6 Effects of congestion in the network

A potential issue with transparent device remoting is device malfunction due to excessive network delays. One cause for such delays in local area networks is congestion. To simulate such congestion, we use *iperf* utility to flood the network with UDP traffic from the dom0 of both of the machines used in our Iozone benchmark tests. Iozone is used for the write test, with file size of 32 MB and record size of 8 MB. RVBD throughput is measured as the level of congestion increases in the network. The results are shown in Figure 8.

RVBD throughput drops almost linearly as congestion is increased, till 500 mbps, and then drops more rapidly. Before 500 mbps, the main reason for throughput reduction is the increase in network latency because of packet queuing in the switch. After 500 mbps, the lack of network bandwidth itself starts to contribute to the drop in Iozone throughput. It is, however, noteworthy that there is no evidence of disk operations timing out or device malfunction. This shows that the guest block driver is tolerant to network congestion.

7.7 Virtual Device Migration and Hot-Swapping

A strong advantage of Netbus is the ability to migrate devices while they are being used. We evaluate the effects of such migration with a multi-tier web application that heavily accesses a disk, and we show how the VM running the application can be migrated seamlessly without any disconnection from the device and hence, without any significant degradation in its performance.

Application Description. Many enterprise applications are constructed as multi-tier architectures, with each tier providing its own set of services [10]. A typical e-commerce site, for instance, consists of a web server at the front-end, a number of application servers in the middle tier, and database servers at the backend [12]. In this environment, it may be desirable to migrate one or more components in a tier to another physical machine for performance (load balancing, etc.) or for maintenance (hardware upgrades, applying software patches, etc.). Our experiments with live migration in multi-tier applications uses the RUBiS open source online auction benchmark [13]. It implements core functionalities of an auction site like selling, browsing and bidding.

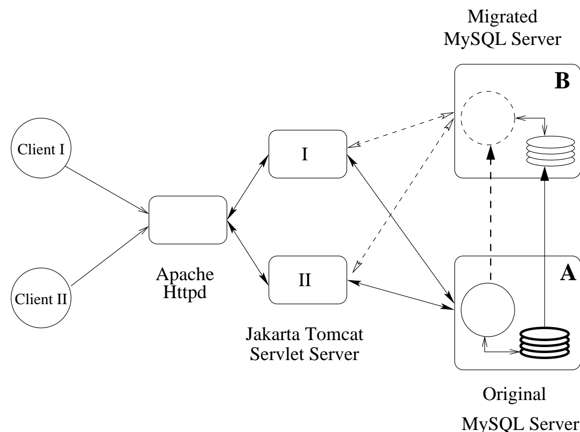

Figure 9: MySQL Server migration in RUBiS

Figure 9 shows the basic setup and the live migration of a MySQL server from node A to node B. The workload is generated using *httperf* running on two separate client machines. Each of the httperf instances create 30 parallel sessions that issue user registration requests to the RUBiS web-server. The web-server forwards the requests to the two application servers, which in turn communicate with the MySQL server backend. The MySQL server runs in a guest VM and to satisfy client requests, heavily accesses a database of size more than 6 GB which is stored on a local disk. In order to demonstrate the seamless migration of block devices, at some point during the experiment, the guest VM running

RUBiS' MySQL server is migrated from host A to host B. For this migration, we evaluate the change in throughput seen by the clients. After migration, we replicate the disk used by the MySQL server from host A to a disk on host B and perform disk hot-swapping. We use *dd* utility to do block level disk replication. The results are shown in Figure 10.

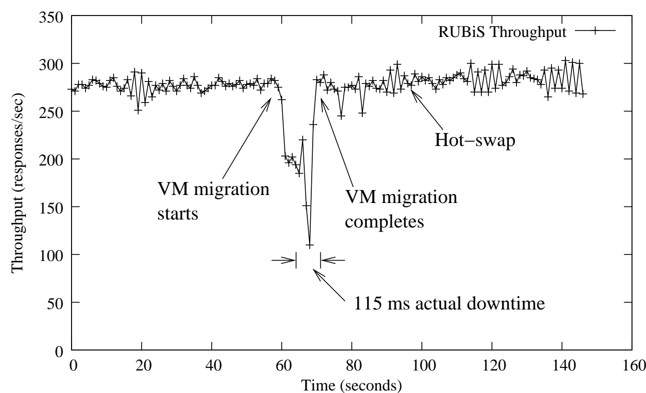

Figure 10: Effect on RUBiS throughput due to MySQL Server migration

Figure 10 shows the drop in the total throughput (measured in response/sec) due to this migration. Note the change in performance in the *push phase*, when the memory pages of the guest VM running MySQL server are copied to node B. This phase lasts for about 6 seconds and during this phase the throughput drops by about 28%. In the *stop-and-copy phase*, throughput further goes down significantly. This phase lasts for approximately 115 milliseconds. This includes the wait time for the disk to go into quiescent state which is about 7 ms. This wait time is actually dependent on the number of pending block requests which was on average 6 in this test. Hence latency overhead of virtual block device migration is only about 6%. We observe that because of virtual device migration, the database server seamlessly accesses the disk remotely. However, we were not able to observe any significant effect of disk hot-swapping on throughput. This is because of the buffer caching effects (discussed in Section 7.2) which cause the throughput for remote disk access (the interval between "Migration completes" and "Hot-swap" in Figure 10) to be same as the throughput for local disk access (before VM migration). Interestingly, the throughput doesn't drop during hot-swapping in spite of device quiescent state. This is because the device quiescent period lasts only about 3 ms which is not noticed by the clients.

To measure the throughput effects of device hot-swapping on disk access, we again used Iozone to measure the disk throughput (without any caching) before VM migration, after VM migration and after disk hot-swapping. Iozone is run inside guest VM to measure the disk throughput as seen by the guest VM. The result is shown in Figure 11.

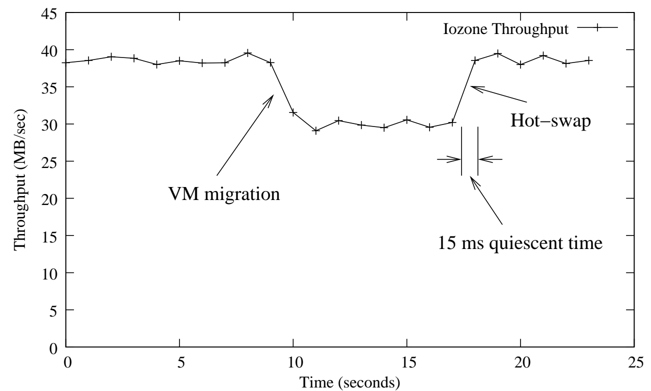

Figure 11: Effect on Iozone throughput due to VM migration and disk hot-swapping (file size = 32MB, test=write, record size = 8MB)

Figure 11 clearly shows the throughput benefits of doing device hot-swapping. The throughput level of Iozone drops after VM migration and is restored after device hotswapping. The number of pending io requests at the start of quiescent state are approximately 30 in this test and the measured downtime for the disk because of this state is 15 ms.

These results show that for complex applications like RUBiS, virtual device migration via Netbus is not only seamless and transparent to guest VMs but also performance transparent. This property is attained without the needs for any special hardware. In addition, for IO-intensive applications, reductions in throughput due to migration can be corrected using Netbus-based device hot-swapping.

8 Conclusions and Future work

This paper presents a software abstraction for accessing remote devices transparently in virtualized environments, termed Netbus. Netbus also supports the seamless migration of virtual devices needed for live VM migration.

When using a prototype implementations of Netbus in Xen for remote access to block and USB devices, the performance seen is similar to that of of local devices accesses. This is the case during normal device operation, due to buffer caching. Netbus performance is also very competitive when drivers are designed to issue multiple simultaneous device requests or when multiple applications simultaneously access the device.

Device remoting with Netbus makes it possible to implement the seamless (i.e., not visible to guest operating

systems) migration of a virtual device. This functionality can then be used to implement methods for VM migration that when applied to complex applications like multi-tier web services, cause little or no effects on application behavior and performance. In addition, for IO-intensive applications, transparent device remoting can be supplemented with device hot-swapping, to remove negative performance effects due to remote device access.

Netbus remains under active development. Current efforts include implementing intelligent device sharing and designing suitable mechanisms for recovering from network failure. We are also exploring the security and trust issues involved in remote device access.

9 Acknowledgement

We would specially like to thank Harry Butterworth from IBM Corporation for sharing his USB virtualization code which was used to implement remote USB virtualization. Kiran Panesar of Intel Corporation helped initiate and provided initial guidance for this research, and Himanshu Raj participated in the design discussions and implementation efforts underlying this work.

References

[1] AMD Pacifica virtualization technology. www.amd.com.

[2] Grand research challeges in information systems. http://www.cra.org/reports/gc.systems.pdf.

[3] Hp utility data center. `http://h71028.www7.hp.com/enterprise/cache/259009-0-0-0-121.aspx`.

[4] Intel Virtualization Technology Specification for the IA-32 Intel Architecture. ftp://download.intel.com/technology/computing/vptech/C97063-002.pdf.

[5] Jndi overview: An introduction to naming services. `http://www.javaworld.com/javaworld/jw-01-2000/jw-01-howto.html`.

[6] Steeleye technologies: Data replication. `http://www.steeleye.com/products/ext_mirroring.html`.

[7] The VMWare ESX Server. http://www.vmware.com/products/esx/.

[8] Virtualization with the hp bladesystem, `http://h71028.www7.hp.com/enterprise/downloads/virtualization%20on%20the%20HP%20BladeSystem.pdf`.

[9] AMIRI, K., PETROU, D., GANGER, G. R., AND GIBSON, G. A. Dynamic function placement for data-intensive cluster computing. In *USENIX 2000 Annual Technical Conference, General Refereed Papers Track* (June 2000), pp. 307–322.

[10] ARON, M., SANDERS, D., DRUSCHEL, P., AND ZWAENPOEL, W. Scalable Content-aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 2000 Annual Usenix Technical Conference* (San Diego, CA, June 2000).

[11] BUSTAMANTE, F., WIDENER, P., AND SCHWAN, K. Scalable directory services using proactivity. In *Proceedings of Supercomputing* (2002).

[12] CARDELLINI, V., CASALICCHIO, E., COLAJANNI, M., AND YU, P. S. The state of the art in locally distributed Web-server systems. In *ACM Computing Survey, 34(2):263311* (June 2002).

[13] CECCHET, E., MARGUERITE, J., AND ZWAENEPOEL, W. Performance and Scalability of EJB Applications. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)* (November 2002), pp. 246–261.

[14] CLARK, C., FRASER, K., HAND, S., HANSEN, J., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA.* (May 2005).

[15] DAHLIN, M., WANG, R., ANDERSON, T. E., AND PATTERSON, D. A. Cooperative caching: Using remote client memory to improve file system performance. In *Operating Systems Design and Implementation* (1994), pp. 267–280.

[16] DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., PRATT, I., WARFIELD, A., BARHAM, P., AND NEUGEBAUER, R. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2003).

[17] EUGSTER, P., FELBER, P., GUERRAOUI, R., AND KERMARREC, A.-M. The many faces of publish/subscribe. *ACM computing Surveys 35*, 2 (June 2003).

[18] HIROFUCHI, T., KAWAI, E., FUJIKAWA, K., AND SUNAHARA, H. Usb/ip - a peripheral bus extension for device sharing over ip network. In *USENIX 2005 Annual Technical Conference, FREENIX Track* (Apr. 2005).

[19] KEIR FRASER AND STEVEN HAND AND ROLF NEUGEBAUER AND IAN PRATT AND ANDREW WARFIELD AND MARK WILLIAMSON. Reconstructing i/o. Tech. rep., University of Cambridge, Computer Laboratory, August 2004.

[20] LEE, G., AND CHRISTO, S. Extending pci express with advanced switching. http://www.eecatalog.com/interconnect/article.php?article=4.

[21] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GÖTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, Dec. 2004).

[22] MISLOVE, A., POST, A., HAEBERLEN, A., AND DRUSCHEL, P. Experiences in building and operating epost, a reliable peer-to-peer application. In *Proceedings of EuroSys* (April 2006).

[23] RAMESH VISWANATH, M. A., AND SCHWAN, K. Harnessing non-dedicated wide-area clusters for on-demand computing. In *Proceedings of Cluster '05* (2005).

[24] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.* (Portland OR (USA), 1985), pp. 119–130.

[25] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 1–14.

[26] WARFIELD, A., HAND, S., FRASER, K., AND DEEGAN, T. Facilitating the development of soft devices. In *USENIX 2005 Annual Technical Conference, General Track* (Apr. 2005).

[27] XUAN, D., BETTATI, C. L. R., CHEN, J., AND ZHAO, W. Utilization-based admission control for real-time applications. In *International Conference on Parallel Processing* (2000), pp. 251–.