Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Unifying Virtual Drivers

Jon Mason
3Leaf Systems
jon.mason@3leafsystems.com
Dwayne Shows
3Leaf Systems
dwayne.shows@3leafsystems.com
Dave Olien
3Leaf Systems
dave.olien@3leafsystems.com

Abstract

Para-virtualization presents a wide variety of issues to Operating Systems. One of these is presenting virtual devices to the para-virtualized operating system, as well as the device drivers which handle these devices. With the increase of virtualization in the Linux kernel, there has been an influx of unique drivers to handle all of these new virtual devices, and there are more devices on the way. The current state of Linux has four in-tree versions of a virtual network device (IBM pSeries Virtual Ethernet, IBM iSeries Virtual Ethernet, UML Virtual Network Device, and TUN/TAP) and numerous out-of-tree versions (one for Xen, VMware, 3leaf, and many others). Also, there are a similar number of block device drivers.

This paper will go into why there are so many, and the differences and commonalities between them. It will go into the benefits and drawbacks of combining them, their requirements, and any design issues. It will discuss the changes to the Linux kernel to combine the virtual network and virtual block devices into two common devices. We will discuss how to adapt the existing virtual devices and write drivers to take advantage of this new interface.

1 Introduction to I/O Virtualization

Virtualization is the ability of a system through hardware and/or software to run multiple instances of Operating Systems simultaneously. This is done through abstracting the physical hardware layer through a software layer, known as a hypervisor. Virtualzation is primarily implemented through two distinct ways: Full Virtualization and Para-virtualization.

Full Virtualization is a method that fully simulates hardware devices through emulating physical hardware components in software. This simulation of hardware allows the OS and other components to run their software unmodified. In this technique, all instructions are translated through the hypervisor into hardware system calls. The hypervisor controls the devices and I/O, and simulated hardware devices are exported to the virtual machine. This hardware simulation does have a significant performance penalty when compared to running the same software on native hardware, but allows the user to run multiple instances of a VM (virtual machine) simultaneously (thus allowing a higher utilization of the hardware and I/O). Examples of this type of virtualization are QEMU and VMware.

Para-virtualization is a method that modifies the OS running inside the VM to run under the hypervisor. It is modified to support the hypervisor and avoid unnecessary use of privileged instructions. These modifications allow the performance of the system to be near native. However, this type of virtualization requires that virtual devices be exposed for access to the underlying I/O hardware. UML and Xen are examples of such virtualization implimentations. The scope of this paper is in considering only para-virtualized virtual machine abstractions; this is where virtual device implementations have proliferated.

To have better I/O performance in a virtual machine, the VMs have virtual devices exported to them and have the native device located in another VM or in the hypervisor. These virtual devices have virtual device drivers for them to appear in the VM as if they were native, physical devices. These virtual device drivers allow the data to be passed from the VM to the physical hardware. This hardware then does the required work, and may or may not return information to the virtual device driver. This level of abstraction allows for higher performance I/O, but the underlying procedure to perform the requested operation differs on each para-virtualzation implementation.

2 Linux Virtualization support

There are many Linux-centered virtualization implementations, both in the kernel and outside the kernel. The implementations that are currently in the Linux kernel have been there for some time and have matured over that time to be robust. Those implementations are UML, IBM's POWER virtualization, and the tun/tap device driver. Those implementations that are currently living outside the kernel tree are gaining popularity, and may one day be included in the mainline kernel. The implementations of note are Xen and 3leaf Systems virtualization (though many others exist).

2.1 In-tree Linux Virtualization support

2.1.1 User Mode Linux

User Mode Linux (UML) provides virtual block and network interfaces with a couple of predominant drivers.

Briefly, the network drivers recommended for UML are TUN/TAP. These are described in a later section. The virtual devices created can be used with other virtual devices on other VMs or with systems on the external network. To configure for forwarding to other VMs, UML comes with a switch daemon that allows user-level forwarding of packets. UML supports hot plug devices: new virtual devices can be configured dynamically while the system is up. As long as a multicast-capable NIC is available on the host, UML supports multicast devices and it is possible to multicast to all the VMs.

UML's virtual storage solution consists of exporting a file from a filesystem on storage known by the host to the guest. That storage can be used directly through reads and writes or with a feature called IO memory emulation that allows a file to be mapped as an IO region. From the kernel address space the driver can use for user processes to mmap to their own address spaces.

All the guest reads and writes go through an API to the host and are translated to host reads and writes. As such, these operations are using the buffer cache on the host. This causes a disproportionate amount of memory to be consumed by the UMLs without limits as to how much they can utilize. As is characteristic to Linux, flushing modified buffers back to disk will be done when appropriate, based upon memory pressure in the host.

Another feature of UML virtual block device is supports "copy on write" (COW) partitions. COW partitions allow a common image of a root device to be shared among many instances of UML. Writes to blocks on the COW partition are kept in a unique device for that UML instance. This is more convenient than the Xen recommendation for using the LVM to provide COW functionality.

UML provides a number of APIs for file IO. These APIs hook into native versions largely unmodified Linux code. These APIs provides the virtual interface to the physical device on the host. Examples of these interfaces are `os_open_file`, `os_read_file`, and `os_write_file`.

2.1.2 IBM POWER virtualization

IBM's PowerPC-based iSeries and pSeries hardware provides a native hypervisor in system firmware, allowing the system to be easily virtualized. The user-level tools allow the underlying hardware to be assigned to specific virtual machines, and in certain cases no I/O hardware at all. In the latter case, one virtual machine is assigned the physical adapters and manages all I/O to and from those adapters. The hypervisor then exposes virtual devices to the other virtual machines. The virtual devices are presented to the OS via the system's Open Firmware device tree, and from the OS's perspective appear to be on a system bus [4]. Currently, the hypervisor supports virtual SCSI, ethernet, and TTY devices.

For the virtual ethernet device, the hypervisor implements a system-wide, VLAN-capable switch [4]. This enables a virtual ethernet device in one VM to have a connection to another VM via its virtual ethernet device. This virtual switch can be connected to the external network only by connecting a virtual ethernet device to a physical device in one of the VMs. In Linux, this would be done via the kernel level ethernet bridging code.

For the virtual SCSI device, the connection is based on a client-server model [4]. The SCSI virtual client device works like most SCSI host controllers. It handles the SCSI commands via the SCSI mid-layer and issues SCSI commands to those devices. The SCSI virtual server device receives all SCSI commands and is responsible for handling them. The virtual SCSI inter-partition communication protocol is the SCSI RDMA Protocol.

2.1.3 TUN/TAP driver

The tun/tap driver has two basic funtions, a network tap and a network tunnel. The network tap simulates a ethernet device, and encapsulates incoming data inside an ethernet frame. The network tunnel simulates an IP layer device, and encapsulates the incoming data in an IP packet. It can be viewed as a simple Point-to-Point or Ethernet device, which instead of sending and receiving packets from a physical media, sends and receives them from user-space programs [5].

This enables virtualization programs running in user space (for example UML) access to the network without needing exclusive access to a network device, or any additional network configuration or OS kernel changes.

2.2 Out-of-tree Linux Virtualization support

2.2.1 Xen

Xen provides an architecture for device driver isolation using a split driver architecture. The virtual device functionality is provided by front-end and back-end device drivers. The front-end driver runs in the unprivileged "guest" domain, and the back-end runs in a "privileged" domain with access to the real device hardware. For block devices, these drivers are called *blkfront* and *blkback*; and for the network devices, *netfront* and *netback*. On the front end, the virtual device appears as a physical device and receives IO requests from the guest kernel.

The front-end driver must issue requests to the back-end driver since it doesn't have access to physical hardware. The back-end verifies that the request is safe and issues it to the real device. The back-end appears to the hypervisor as a normal user of in-kernel IO functionality. When the IO completes, the back-end notifies the front-end that its data is ready. The front-end driver reports IO completions to its own kernel. The back-end is responsible for translating device addresses and verifying that requests are correct and do not violate isolation guarantees.

Xen accomplishes device virtualization through a set of clean and simple device abstractions. IO data is transferred to and from each domain via grant tables using shared-memory, asynchronous buffer-descriptor rings. These are said to provide a high-performance communication mechanism for passing buffer information vertically through the system, while allowing Xen to efficiently perform validation checks—for example, of a domain's credits. This shared-memory interface is the fundamental mechanism supporting the split device drivers for network and block IO.

Each domain has its own grant table. This data structure is shared with the hypervisor, allowing the domain to tell Xen what kinds of permissions other domains have on its pages. Entries in the grant table are identified by a grant reference, an integer, which indexes into the grant table. It acts as a capability which the grantee can use to perform operations on the granter's memory. This mechanism allows shared-memory communications between unprivileged domains. A grant reference also encapsulates the details of a shared page, removing the need for a domain to know the real machine address of a page it is sharing. This makes it possible to share memory correctly among domains running in fully virtualized memory.

Grant table manipulation, the creation and destruction of grant references, is done by direct access to the grant table. This removes the need to involve the hypervisor when creating grant references, changing access permissions, etc. The grantee domain invokes hypercalls to use the grant reference.

Xen uses event-delivery mechanism for sending asynchronous notifications to a domain, similar to a hardware interrupt. These notifications are made by updating a bitmap of pending event types, and optionally calling an event handler specified by the guest. The events can be held off at the discretion of the guest.

Xenstore is the mechanism by which these event channels are set up, along with the shared memory frame.

It is used for setting up shared memory regions and event channels for use with the split device drivers. The store is arranged as a hierarchical collection of key-value pairs. Each domain has a directory structure containing data related to its configuration.

The setup protocol for a device channel should consist of entering the configuration data into the Xenstore area. The store allows device discovery without requiring the relevant device structure to be loaded. The probing code in the guest should see the Xen "bus."

Once communications is established between a pair of front- and back-end drivers, the two can communicate by directly placing requests/responses into shared memory and then on the event channel. This separation allows for message batching, making for efficient device access.

Xen Network IO

As mentioned, the shared memory communication area is shared between front-end and back-end domains. From the point of view of other domains, the back-end is viewed as a virtual ethernet switch, with each domain having one or more virtual network interfaces connected to it.

From the reference point of the back-end domain, the network driver on the back end consists of a number of ethernet devices. Each of these has a connection to a virtual network device in another domain. This allows the back-end domain to route, bridge, firewall, etc. all traffic from and to the other domains using the usual Linux mechanisms.

The back end is responsible for:

- Validation of data. The back end ensures the front ends do not attempt to generate invalid traffic. The back end may look at headers to validate MAC or IP addresses, making sure they match the interface they have been sent from.
- Scheduling. Since a number of domains can share the same physical NIC, the back end must schedule between domains that can have packets queued for transmission, or that may have ingress traffic. The back end is capable of traffic-shaping or rate-limiting schemes. Logging/Accounting on the back end can be configured to track/record events.

  Ingress packets from the network are received by the back end. The back end simply acts as a demultiplexer, forwarding incoming packets to the correct front end via the appropriate virtual interface.

The asynchronous shared buffer rings described earlier are used for the network interface to implement transmit and receive rings. Each descriptor ring identifies a block of contiguous machine memory allocated to the domain. The transmit ring carries packets to transmit from the guest to the back-end domain. The return path of this ring carries messages indicating contents have been transmitted. This signals that the back-end driver does not need the pages of memory associated with that request.

To receive packets, the guest puts descriptors for unused pages of memory on the receive ring. The back end exchanges these pages in the domain's memory with new pages containing the received packet and passing back descriptors regarding the new packets in the ring. This is a zero-copy approach, allowing the back end to maintain a pool of free pages to receive packets into, delivering them to the associated domains after reading their headers. This is known as *page flipping*.

A domain that doesn't keep its receive ring filled with empty buffers will have dropped packets. This is seen as an advantage by Xen because it limits live-lock problems because the overloaded domain will stop receiving further data. Similarly, on the transmit path, it provides the application the feedback on the rate at which the packets can leave the system.

Flow control on the rings is managed by an independent mechanism from the flow of data on the transmit/receive rings. In this way the ring is divided into two message queues, one in each direction.

Xen Block IO

All disk access uses the virtual block device interface. It allows domains access to block storage devices visible to the block back-end device. The virtual block device is a split driver, like the network interface. A single shared memory ring is used between the front and back end for each virtual device. This memory ring handles all IO requests and responses for that virtual device.

Many storage types can be exported by the back-end domain for use by the front end—various network-based block devices such as iSCSI, or NBD, as well as loopback and multipath devices. These devices get mapped to a device node on the front end in a static way defined by the guest's startup configuration.

The ring used by block IO supports two message types, read and write. For a read, the front end identifies the device and location to read from and attaches pages for data to be copied into. The back end acknowledges completed reads after the data is transferred from the device into the buffer, typically the underlying physical device's DMA engine. Writes are analogous to reads, except data moves from the front end to the back end.

Xen IO Configuration

Domains with physical device access (i.e., driver domains) receive access to certain PCI devices on a limited basis, acquiring access to interrupts and bus address space. Many guests attempt to determine the PCI configuration by accessing the PCI BIOS. Xen forbids such access and provides a hypercall, `physdev_op` to set/query configuration details.

2.2.2 3leaf Virtualization

3leaf Systems provides virtual storage and network interfaces built on top of typical Linux APIs. Their devices span physical machine boundaries so that front-end drivers can be on one system and the back-end drivers hosted where the physical devices reside.

Their front-end drivers communicate to the back-end drivers through a transport-agnostic interface that lends itself to running on any number of transports that meet some minimal set of requirements.

Front-end network devices look like a normal ethernet interface for the purposes of the front-end application/user. Such devices have their own MAC addresses randomly generated. Egress packets get wrapped with header information before being passed to the lower layers of the transport. This header is used for demultiplexing. The corresponding virtual NIC on the back end passes the packet to the bridge, where the packet gets forwarded to its recipient.

The storage front end looks like a SCSI device, usually having a SCSI device backing the back-end driver. The front-end registers with the block device layer so the SCSI mid-layer can pass off requests with scatterlists. These get wrapped with header information before being passed to the lower layers of transport. This header is used for demultiplexing on the back end where the request is forwarded to the SCSI device. Completions come back to the back end where they are wrapped with the header information for the return trip through the transport to the front end. The front-end driver forwards the completion to the SCSI midlayer.

The 3leaf stack can manage multiple devices, and hotplug events, but can entail software queuing at different levels, especially the networking stack. Interactions between front and back ends are mitigated through thoughtful use of scatter-gather lists to chain requests. 3leaf services do not rely on an operating system being fully or para-virtualized; it is more of a distributed IO services mechanism. In some ways the hardware where the back-end runs is analogous to a hypervisor, whereas the front-end systems can be many and serve as the guests in the model of the virtual IO services surveyed here.

3Leaf virtual storage virtualization supports a number of useful features for providing diskless front-end clients with controlled, high-availability, high-performance access to storage. It also includes tools for assisting centralized provisioning to these distributed clients.

Following is a list of capabilities supported by this storage virtualization implementation. Each capability will be followed by a brief description of its implementation.

Features supported include:

- Multiple redundant access paths to storage. A typical configuration has two or more back-end servers, each with redundant paths to storage on a fibre channel SAN. The fail-over and fail-back between redundant fibre channel paths on the back-end redundant paths is managed by the Linux device mapper multipathing. A front end then has paths to storage through two or more back-end servers. The front-end and back-end storage software manages fail over and fail back between back-end servers.
- Block or SCSI storage devices. The virtual storage devices can appear on the front-end clients as either block or SCSI devices.
- Name persistence. The Linux hotplug software also includes mechanisms that can be used give

storage devices persistent names. For example, these names can be based on the UUID of the physical storage device. This mechanism requires some intervention on the front-end client to establish these mappings. The hotplug mechanism is still available for use on the 3leaf front-end clients. But the 3leaf virtual storage software also maintains its own stable naming mechanism. This can be administered centrally from the back-end storage server.

- Boot support for diskless front-end clients. The front-end client is able to load its operating system from the back-end server and then use a virtual disk as its root device. Access to this root device will also be highly available using the redundant paths provided by the storage virtualization.
- COW virtual devices. As with UML, the 3leaf COW devices are especially useful for root disks. It allows several client front ends to share a common root file system, allocating additional storage only for each client's modifications to that shared image.
- NPIV. N-port interface virtualization is another mechanism which allows the owner of the SAN to regulate access to devices on that SAN by individual front-end clients. This is based upon a virtual host bus adapter model, where the SAN administrator can associate sets of storage devices with virtual HBAs and then associate individual VHBAs with different client machines.
- Centralized provisioning. The back-end servers interact with a distributed set of tools to specify the virtual storage environment for the storage client. Storage devices can be added or removed from the client's environment, generating hotplug events to update the client's storage name space.

The 3leaf virtual storage implementation is conceptually similar to the Xen storage device implementation. Both are based on an efficient and reliable means for passing messages and DMA data transfers between the "guest" or front-end clients, and the "host" or back-end servers.

The messaging mechanism is used to implement an rpc-like communication between the back-end servers and the front-end clients. This rpc mechanism is used to simulate SCSI/Fibre channel behavior when needed, and to support the creation/deletion of virtual disks, the construction of COW devices, and VHBAs.

When the underlying transport supports it (e.g., infiniband RDMA), the transfer of disk data is transferred using RDMA read and RDMA write operations. RDMA read and write operations are initiated on the client. The RDMA operations use "opaque" handles to identify memory on the back-end servers that are the source or target of RDMA reads and writes, respectively. The client memory to be used is identified by a scatter/gather list.

These opqaue rdma handles are generated on the back-end servers as part of registering portions of the server's physical memory to be used for RDMA operations. In the case of infiniband, these handles are encoded in so that they cannot be forged. This provides some isolation between client front ends, making it difficult for clients to maliciously generate RDMA handles to memory they should not have access to. The opaque memory handles are transmitted from the back end server nodes to the front end's servers using the RPC mechanism. Each client is given a set of RDMA handles for segments of server memory. These memory segments sets are not shared between clients. Thus there is no chance of misdirected RDMA operations. Each client has ownership and control of its set of RDMA handles until it releases them. To perform a disk IO transfer, a client allocates an RDMA handle from the set given to it by the destination server. In the case of a write, it first transfers data using RDMA into the server's client memory. Then issues an rpc call to the server instructing it to write that memory to a disk device. In the case of a disk read, the client first sends an rpc to the server instructing it to read data from disk into the server's memory. It then uses its RDMA handle for that memory to transfer that data from the server's memory to the client's.

The rest of the virtual disk implementation is built upon these messaging and RDMA primitives. On the client, the virtual disk implementation appears to the Linux operating system's block layer and SCSI mid-layer as just another disk driver. The driver accepts either SCSI requests or block request structures, and translates them into rpc messages and RDMA operations targeted towards one of the servers providing access to disk storage. If the targeted server fails, these operations will be re-directed to one of the redundant servers for that storage.

On the server side, the disk virtualization is logically at the same level in the Linux disk software stack as the Linux page cache. The client and server's virtualization software manages its own pool of memory for RDMA operations. RPC requests from the client cause block requests to be submitted to the server's block layer to cause disk read and write operations. In this way, disk data transfers are performed without any copying of disk data by the CPUs on either the clients or the servers.

2.2.3 Non-local memory transport of data via IB

OpenFabrics.org is another player in the fabric of virtual IO solutions. Their solutions use infiniband as the transport to provide network and storage solutions to low-cost, front-end machines that run with their drivers and commodity HCA cards.

In this paradigm, virtual network interfaces are provided over the HCA ports with a protocol built on the verbs/ access layer called the IPoIB module. This uses the two physical interfaces for each HCA; however, the MAC used is the GUID, which means it becomes difficult to put these packets on ipv4 networks. The ethernet header then needs to be massaged before transmitted packets can go out. When the virtual device's MAC is tied to the hardware, it becomes difficult to migrate virtual devices to other ports. The packets have to be bridged from the IB network to the ethernet network, much as packets from other solutions.

The storage solution provided is a module based on SCSI Request Protocol (SRP) initiator. It provides access to IB-based SRP storage targets.

3 Virtualized I/O

3.1 Virtualized Networking

The need for unified virtual ethernet drivers are many. First, the multitude of pre-existing code, and the potential for more in the future. The partial addition of features in an uncoordinated effort when many of the rest could also benefit from these features. UML provides switching from a user-level daemon which will have less performance than a kernel module. Not all implementations support hot-plug additions of virtual devices to the VM. Self-virtualizing devices are soon to be on the market and a common architecture should be adapted to take advantage of that functionality in the NICs.

3.2 Virtualized Storage

The need for unification of virtualized storage drivers is several fold. First, file-based partitions are slow. While convenient for desktop users, they do not meet the needs of an enterprise-scaled organization. It is well documented that read/write performance in the interfaces UML uses are slow. Additionally, file-based partitions also suffer from unchecked buffer cache growth on the host system. That tends to help performance, keeping much of the disk/partition resident in memory; however, as the number of disks group, it can cause inequities as to which VM has the lion's share of memory tied up. Attempted solutions to resolve that have been controversial and seem too specialized for the particular hypervisor running. `mmap` performance is not much better as an alternative, as some experiences indicate.

Inconsistent CoW implemenatations. The market has shown the need to have a single master root drive that is read-only, backed by a writable device to manage VM-based configuration differences which CoW provides a per-block mechanism for satisfying. Using the Linux Volume Manager as a solution goes beyond its original design; likewise there are limitations in using a bitmap implementation exclusively.

As far as disk-based partitions vs. file-based partitions, a disk gives a better unit of granularity that SAN storage providers have deployed with. It makes concepts like zoning and n-port virtualization much more achieveable.

SCSI-based devices seem a familiar and dependable mechanism to build a framework under. This mechanism is used by many other drivers and is itself a dependable framework supporting multiple device types. SCSI request blocks are already a well defined way to format requests and completions. A flexible back end should support all device types; however, for the scope of this document the discussion will be focused on SCSI disks. Such disks could be SAS (serial attached scsi) or storage that is allocted from a SAN fabric through an HBA (host bus adapter).

Lastly, a single flexible implementation will be better supported by the Linux community.

4 Design requirements and open issues

Abstracting the existing virtual drivers into a generic implementation raises interesting design requirements and issues. Regardless of how the underlying virtualization actually works, there are a few basic interfaces for the network transport layer to interlock with the virtual ethernet driver, and the storage transport layer to interlock with the virtual SCSI driver.

4.1 Net

All the virtual ethernet device drivers described above contain certain basic features which are common, though the underlying methods of transfer the data from one point to another differs. All virtual network device drivers have functions which create and delete the device, start and stop the network stack, and transmit and receive data. These basic functions can be created in a generic virtual ethernet driver, which would then be supplemented by a specific module that handles any transport-specific calculations and transports the data from one point to another.

By providing a `veth_ops` data structure, with the pointers to the transport layer "helper" functions, this division of labor can greatly increase the ability to separate the existing virtual ethernet driver into a generic and transport layer.

For example in Xen, the packet to be transmitted has a few transport-specific calculations that need to be done (like insertion into the grant table and calculation for the number of pages the skb fits into). Those can be pushed down into a transmit transport layer, to which the skb is handed for transmission. In contrast, the tun/tap driver simply queues the packet onto a skb queue.

```
veth_ops->tx(struct sk_buff *skb,
             struct net_device *netdev);
```

This enables generic error checking and setup that all data transmission routes need in the generic tx stub, and the transport-specific work can be done in the function pointed to by the `veth_ops`.

For receiving, things can be significantly simplier using NAPI. Registering a generic poll routine on device open, and providing a skb queue to pull from, makes this very generic. The transport layer populates the queue as packets are pulled off its transport layer (via interrupt, etc.).

Open and close will need generic and transport-layer constructs to set up the virtual device to transmit and receive data, as well as destroy any resources allocated. While some of these functions are specific to the transport layer (like creating buffer pools), most of the drivers require the very basic setup of zeroing statistics and starting the transmit queue.

```
veth_ops->open(struct net_device
               *netdev);
veth_ops->close(struct net_device
                *netdev);
```

Unfortunately, driver probe and create are very dependent on how the attributes are passed to the driver. Since some are passed via `hcall` and others are provided by the user-space tools, there really is no abstract way to do this.

There are other generic functions that can easily be made generic. Specificly, functions to change the MTU, transmit timeout, and get statistics are all generic and really do not need any hooks in them to work for all virtualization implementations.

There are currently some offloading technologies which have been implemented in some of the drivers via software, and which have not been proliferated into all of the existing drivers. For example, GSO and checksum offloads. Integrating these functions into existing implementations might be easy; they are very implementation-dependent and should only be abstracted after further investigation.

Our current implementation uses the API defined above to communicate between the generic virtual ethernet driver and a few generic transport layers. Specificly, we generalized the Xen network to behave in the above way, as well as UML and 3leaf.

4.2 Disk

We propose virtualization-aware devices using a unified generic stub. This allows for a transport/virtualization-specific layer. If a transport were to be connected through the generic stubs, the front end and back end would not have to be co-located on the same piece of hardware.

By providing a `vstore_ops` structure, with the structure elements being pointers to transport layer "helper" functions, one can separate existing virtual storage drivers into a generic and transport layer.

In Xen the IO request results in pages inserted into the grant table for the blkback driver for reads and writes. They can be pushed down to the transport layer in a way that shields the upper layers from the transport. In contrast, UML with its UBD driver does reads and writes to the host operating system, largely driven through the `os_*` interfaces. These are block device interfaces, but the pertinent information is available from the `ioreq` struct. The 3leaf front-end driver can accomodate both. Writes and reads have the same prototypes:

```
vstore_ops->dma_write(struct scatterlist
                      local_buf_list[],
                      int nbuf, int size);
vstore_ops->dma_read(struct scatterlist
                     local_buf_list[],
                     int nbuf, int size);
```

Open and release of block devices are generic enough that the transport-specific portion is easily isolated. The open causes caches of buffers to be allocated and data structure initialization. The release causes those resources to be freed.

```
vstore_ops->open(struct inode *inode,
                 struct file *filp);

vstore_ops->release(struct inode *inode,
                    struct file *filp);
```

The auto-provisioning in the 3leaf model is preferable for many reasons. There needs to be a way for a VM to probe for devices; this callback fills that requirement, not unlike Xen—when it reads its configuration, it's technically accessing the Xen bus.

5 Roadmap/Future work

We plan on extending this work into all virtualization implementations mentioned in this paper. However, due to certain logistical limitations, we have not been able to do this. For example, there was no access to IBM POWER-enabled hardware.

Self-virtualizing devices which can be offloaded with certain virtualization functionality are becoming more prevalent. These devices require that one physical device appear as multiple devices, each VM having a semi-programmable device exclusively for its own. Exporting the generic driver to these devices could be quite beneficial.

A performance analysis of the merged drivers, with a contrast to the existing drivers, should be done prior to any merging into mainline.

The driver should be expanded to accomodate block devices as well as scsi, using the 3leaf model for supporting both types.

Features in the 3leaf solution should be addressed, including multiple redundant paths to storage, name persistence, and centralized provisioning.

6 Conclusion

We implemented a generic virtual device driver with implementation-specific transportation layer for UML, Xen, and 3leaf Systems. We have shown a breadth of virtualization technologies that would benefit from using this, and the generic API which can be used to do this.

We will continue to clean and extend the usage of this implementation in Linux. Hopefully, by the time this is read, it will be submitted for inclusion into the Linux kernel.

7 Terms

front end, back end
One class of distributed computer system in which the computers are divided into two types: back-end computers and front-end computers. Front-end computers typically have minimal peripheral hardware (e.g., storage and ethernet) and interact with users and their applications. Back-end computers provide the front ends with access to expensive peripheral devices or services (e.g., a database), so as to share the cost of those peripherals across the front ends.

Also: client, server.

bridge
A mechanism to forward network packets between ports or interfaces.

VM (virtual machine)
Software that provides a virtual environment on top of

the hardware platform. This virtual environment provides services for creating and managing virtual IO devices such as disks and network interfaces.

CoW (Copy on Write)
A technique for efficiently sharing the un-modified contents of a disk or file system whose contents are "read mostly"—for example, a root file system. Writes to this shared content are re-directed to a write-able device that is unique to each user.

DMA (Direct Memory Access)
A hardware mechanism used by peripheral devices to transfer data between the pre-determined locations in computer memory and the peripheral device, without using the computer's cpu to copy that data.

RDMA (Remote Direct Memory Access)
An extension of DMA where data is transferred between pre-determined memory locations on two computer systems over a network connection, without utilizing cpu cycles to copy data.

MAC (Media Acess Control) address
A unique identifier associated with a network adapter.

GUID (Globally Unique Identifier)
A 128-bit number, unique identifier that is associated with an infiniband HCA.

IPoIB (Internet Protocol over InfiniBand)
An implementation of the internet protocol on the Infiniband network fabric.

SRP (SCSI RDMA Protocol)
A combination of the SCSI protocol with Infiniband RDMA, providing SAN storage.

UML (User Mode Linux)
A virtual machine implementation where one or more guest Linux operating systems run in user-mode Linux processes.

hypvervisor (also, virtual machine monitor)
The software that provides the virtual machine mechanisms to support guest operating systems.

infiniband
A point-to-point communications link used to provide high performance data and message transfer between computer nodes.

NIC (Network Interface Controller)
A hardware device that allows computers to communicate over a network.

page cache, buffer cache
A cache of disk-backed memory pages. The Linux operating system uses a page cache for holding process memory pages as well as file data.

scatterlist
A list (typically an array) of physical memory addresses and lengths used to specify the source or destination for a DMA transfer.

RPC (Remote Procedure Call)
A protocol where software on one computer can invoke a function on a remote computer.

SCSI (Small Computer System Interconnect)
A standard for physically connecting and transferring data between computer systems and peripheral storage devices.

API (Application Programming Interface)
A software interface definition for providing services.

HCA (Host Channel Adapter)
A hardware device for connecting a computer system to an infiniband communications link.

HBA (Host Bus Adapter)
A hardware device that connecting a computer system to a SCSI or Fibre Channel link.

hotplug (hot plugging)
A method for adding or removing devices from a computer system while that computer system is operating.

N-Port (Node Port)
a Fibre Channel node connection.

Fibre Channel
A network implementation that is used mostly for accessing storage.

SAN (Storage Area Network)
A network architecture for attaching remote storage to a server computer.

Volume Manager
Software for managing and allocating storage space.

8 Legal Statement

References

[1] M.D. Day, R. Harper, M. Hohnbaum, A. Liguori, & A. Theurer. "Using the Xen Hypervisor to Supercharge OS Deployment," Proceedings of the 2005 Linux Symposium, Ottawa, Vol. 1, pp. 97–108.

[2] K. Fraser, S. Hand, C. Limpach, and I. Pratt. "Xen and the Art of Open Source Virtualization," Proceedings of the 2004 Linux Symposium, Ottawa, Vol. 2, p. 329.

[3] Xen 3.0 Virtualization Interface Guide `http://www.linuxtopia.org/online_books/linux_virtualization/xen_3.0_interface_guide/linux_virualization_xen_interface_25.html`

[4] Dave Boutcher, Dave Engebretsen. "Linux Virtualization on IBM POWER5 Systems," Proceedings of the 2004 Linux Symposium, Ottawa, Vol. 1, p. 113.

[5] `http://www.kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/Documentation/networking/tuntap.txt`

[6] Raj, Ganev, Schwan, and Xenidis. Scalable IO Virtualization via Self-Virtualizing Devices, `http://www-static.cc.gatech.edu/~rhim/Self-VirtTR-v1.pdf`

[7] Jeff Dike, *User Mode Linux*, Prentice Hall, 2006.

[8] `http://wiki.openvz.org`