virtio: Towards a De-Facto Standard For Virtual I/O Devices

Rusty Russell
IBM OzLabs
8 Brisbane Ave
Canberra, Australia
rusty@au.ibm.com

ABSTRACT

The Linux Kernel currently supports at least 8 distinct virtualization systems: Xen, KVM, VMware's VMI, IBM's System p, IBM's System z, User Mode Linux, lguest and IBM's legacy iSeries. It seems likely that more such systems will appear, and until recently each of these had its own block, network, console and other drivers with varying features and optimizations.

The attempt to address this is virtio: a series of efficient, well-maintained Linux drivers which can be adapted for various different hypervisor implementations using a shim layer. This includes a simple extensible feature mechanism for each driver. We also provide an obvious ring buffer transport implementation called vring, which is currently used by KVM and lguest. This has the subtle effect of providing a path of least resistance for any new hypervisors: supporting this efficient transport mechanism will immediately reduce the amount of work which needs to be done. Finally, we provide an implementation which presents the vring transport and device configuration as a PCI device: this means guest operating systems merely need a new PCI driver, and hypervisors need only add vring support to the virtual devices they implement (currently only KVM does this).

This paper will describe the virtio API layer as implemented in Linux, then the vring implementation, and finally its embodiment in a PCI device for simple adoption on otherwise fully-virtualized guests. We'll wrap up with some of the preliminary work to integrate this I/O mechanism deeper into the Linux host kernel.

General Terms

Virtio, vring, virtio_pci

Keywords

Virtualization, I/O, ring buffer, Linux, KVM, lguest

1. INTRODUCTION

The Linux kernel has been ported to a huge number of platforms; the official kernel tree contains 24 separate architecture directories and almost 2 million lines of architecture-specific code out of 8.4 million. Most of these architectures contain support for multiple platform variants. Unfortunately we are aware of only one platform which has been deleted from the tree (as the last machine of its kind was destroyed) while new hardware variants sprout like weeds after a rain. With around 10,000 lines changing every day, the kernel has at least one of everything you can imagine.

When we look at Linux as a guest under virtualization, we are particularly blessed: IBM's System p, System z and legacy iSeries are all supported. User Mode Linux[4] has long been included, for running Linux as a userspace process on Power, IA64 and 32 and 64 bit x86 machines. In the last two years the x86 architecture has proven particularly fecund, with support for Xen[2] from XenSource, VMI[1] from VMware and KVM[5] from Qumranet. Last and least, we should mention my own contribution to this mess, lguest[7]: a toy hypervisor which is useful for development and teaching and snuck quietly into the tree last year.

Each of these eight platforms want their own block, network and console drivers, and sometimes a boutique framebuffer, USB controller, host filesystem and virtual kitchen sink controller. Few of them have optimized their drivers in any significant way, and offer overlapping but often slightly different sets of features. Importantly, no-one seems particularly delighted with their drivers, or having to maintain them.

This question became particularly pertinent as the KVM project, which garnered much attention when it burst onto the Linux scene in late 2006, did not yet have a paravirtual device model. The performance limitations of emulating devices were becoming clear[6], and yet the prospect of either adopting the very-Xen-centric driver model was almost as unappealing as developing Yet Another driver model. Having worked on the Xen device model, we believe it possible to create a general virtual I/O mechanism which is efficient[14], works on multiple hypervisors and platforms, and atones for Rusty's involvement with the Xen device configuration system.

2. VIRTIO: THE THREE GOALS

Our initial goal of driver unification is fairly straight-forward: all the work is inside the Linux kernel so there's no need for any buy-in by other parties. If developers of boutique virtual I/O mechanisms are familiar with Linux, it might guide them to map the Linux API neatly onto their own ABI. But "if" and "might" are insufficient: we can be more ambitious than this.

Experience has shown that boutique transport mechanisms tend to be particular not only to a given hypervisor and ar-

chitecture, but often to each particular kind of device. So the next obvious step in our attempt to guide towards uniformity is to provide a common ABI for general publication and use of buffers. Deliberately, our *virtio_ring* implementation is not at all revolutionary: developers should look at this code and see nothing to dislike.

Finally, we provide two complete ABI implementations, using the virtio_ring infrastructure and the Linux API for virtual I/O devices. These implement the final part of virtual I/O: device probing and configuration. Importantly, they demonstrate how simple it is to use the Linux virtual I/O API to provide feature negotiation in a forward and backward compatible manner so that future Linux driver features can be detected and used by any host implementation.

The explicit separation of drivers, transport and configuration represents a change in thinking from current implementations. For example, you can't really use Xen's Linux network driver in a new hypervisor unless you support XenBus probing and configuration system.

3. VIRTIO: A LINUX-INTERNAL ABSTRACTION API

If we want to reduce duplication in virtual device drivers, we need a decent abstraction so drivers can share code. One method is to provide a set of common helpers which virtual drivers can use, but more ambitious is to use common drivers and an operations structure: a series of function pointers which are handed to the generic driver to interface with any of several transport implementations. The task is to create a transport abstraction for all virtual devices which is simple, close to optimal for an efficient transport, and yet allows a shim to existing transports without undue pain.

The current result (integrated in 2.6.24) is that virtio drivers register themselves to handle a particular 32-bit device type, optionally restricting to a specific 32-bit vendor field. The driver's `probe` function is called when a suitable virtio device is found: the `struct virtio_device` passed in has a `virtio_config_ops` pointer which the driver uses to unpack the device configuration.

The configuration operations can be divided into four parts: reading and writing feature bits, reading and writing the configuration space, reading and writing the status bits and device reset. The device looks for device-type-specific feature bits corresponding to features it wants to use, such as the `VIRTIO_NET_F_CSUM` feature bit indicating whether a network device supports checksum offload. Features bits are explicitly acknowledged: the host knows what feature bits are acked by the guest, and hence what features that driver understands.

The second part is the configuration space: this is effectively a structure associated with the virtual device containing device-specific information. This can be both read and written by the guest. For example, network devices have a `VIRTIO_NET_F_MAC` feature bit, which indicates that the host wants the device to have a particular MAC address, and the configuration space contains the value.

These mechanisms give us room to grow in future, and for hosts to add features to devices with the only requirement being that the feature bit numbers and configuration space layout be agreed upon.

There are also operations to set and get an 8 bit device status word which the guest uses to indicate the status of device probe; when the `VIRTIO_CONFIG_S_DRIVER_OK` is set, it shows that the guest driver has completed feature probing. At this point the host knows what features it understands and wants to use.

Finally, the reset operation is expected to reset the device, its configuration and status bits. This is necessary for modular drivers which may be removed and then re-added, thus encountering a previously initialized device. It also avoids the problem of removing buffers from a device on driver shutdown: after reset the buffers can be freed in the sure knowledge that the device won't overwrite them. It could also be used to attempt driver recovery in the guest.

3.1 Virtqueues: A Transport Abstraction

Our configuration API is important, but the performance-critical part of the API is the actual I/O mechanism. Our abstraction for this is a *virtqueue*: the configuration operations have a `find_vq` which returns a populated structure for the queue, given the virtio device and an index number. Some devices have only one queue, such as the virtio block device, but others such as networking and console devices have a queue for input and one for output.

A virtqueue is simply a queue into which buffers are posted by the guest for consumption by the host. Each buffer is a scatter-gather array consisting of readable and writable parts: the structure of the data is dependent on the device type. The virtqueue operations structure looks like so:

```
struct virtqueue_ops {
    int (*add_buf)(struct virtqueue *vq,
                   struct scatterlist sg[],
                   unsigned int out_num,
                   unsigned int in_num,
                   void *data);
    void (*kick)(struct virtqueue *vq);
    void *(*get_buf)(struct virtqueue *vq,
                     unsigned int *len);
    void (*disable_cb)(struct virtqueue *vq);
    bool (*enable_cb)(struct virtqueue *vq);
};
```

The `add_buf` call is used to add a new buffer to the queue; the data argument is a driver-supplied non-`NULL` token which is returned when the buffer has been consumed. The `kick` call notifies the other side (i.e., the host) when buffers have been added; multiple buffers can be added before a kick, for batching. This is important as notification usually involves an expensive exit of the guest.

The `get_buf` call gets a used buffer: the length which was written to the buffer by the other side is returned (we'll see why in the discussion of inter-guest communication). It returns the cookie handed to `add_buf` or `NULL`: buffers are not necessarily used in order.

`disable_cb` is a hint that the guest doesn't want to know when a buffer is used: this is the equivalent of disabling a device's interrupt. The driver registers a callback for the virtqueue when it is initialized, and the virtqueue callback might disable further callbacks before waking a service thread. There's no guarantee that the callback will not still be called after this, however: that would require expensive synchronization especially on an SMP system. In effect, this is merely an optimization to reduce unnecessary interaction with the host or VMM.

`enable_cb` is the counterpart to `disable_cb`. Often a driver will re-enable callbacks once it has processed all the pending buffers in the queue. On some virtio transports there is a race: buffers might be used between `get_buf` returning `NULL` and the `enable_cb` call, and yet no callback will be called. Level-triggered interrupt implementations would not have this problems, but for those that do, `enable_cb` will return false to indicate more work has appeared in that window where the callback was disabled.

All of these calls are usable from any context in Linux, and it is up to the caller to ensure that they are not called simultaneously. The only exception is `disable_cb`—it is often called from the callback itself, and also used to disable the callback, but it is unreliable so it could occur at any time.

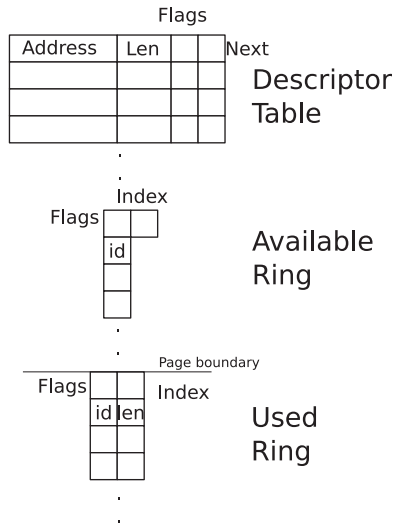

Figure 1: virtio_ring layout in guest memory

4. VIRTIO_RING: A TRANSPORT IMPLEMENTATION FOR VIRTIO

Although we believe any optimal transport will share similar characteristics, the Linux virtio/virtqueue API is biased towards our particular transport implementation, called virtio_ring.

Prior to the creation of virtio, lguest already had a virtual I/O system: a generic, n-way I/O mechanism which we used for inter-guest networking, based on an earlier one for Xen. But much of the complexity of the system came from its broadcast N-way nature, which it seemed was only desired for a special case of a guest LAN. As interesting as that feature is, sacrificing it leads us to a simpler scheme involving ringbuffers which is a standard method of high-speed I/O[2][3]. After a few implementation iterations we had the virtio_ring scheme in use by both lguest and KVM today [10].

The virtio_ring consists of three parts: the descriptor array where the guest chains together length/address pairs, the available ring where the guest indicates what descriptors chains are ready for use, and the used ring where the host indicates which descriptors chains it has used. The size of the ring is variable, but must be a power of two.

```
struct vring_desc
{
    __u64 addr;
    __u32 len;
    __u16 flags;
    __u16 next;
};
```

Each descriptor contains the guest-physical address of the buffer, its length, an optional 'next' buffer for chaining, and two flags: one to indicate whether the next field is valid and one controlling whether the buffer is read-only or write-only. This allows a chained buffer to contain both readable and writable sections, which proves useful for implementing a block device. By convention, readable buffers precede writable buffers.

The use of 64-bit addresses even in 32-bit systems is a trade-off: it allows one universal format at the cost of 32 bits on older platforms. All structures are chosen to avoid padding on any but the most perverse architectures, but we stop short of defining a specific endian format: the guest assumes its natural endianness.

```
struct vring_avail
{
    __u16 flags;
    __u16 idx;
    __u16 ring[NUM];
};
```

The available ring consists of a free-running index, an interrupt suppression flag, and an array of indices into the descriptor table (representing the heads of buffers). The separation of the descriptors from the available ring is due to the asynchronous nature of the virtqueue: the available ring may circle many times with fast-serviced descriptors while slow descriptors might still await completion. This is obviously useful for implementing block devices, but also turns out to be useful for zero-copy networking.

```
struct vring_used_elem
{
    __u32 id;
    __u32 len;
};
```

```
struct vring_used
{
    __u16 flags;
    __u16 idx;
    struct vring_used_elem ring[];
};
```

The used ring is similar to the available ring, but is written by the host as descriptor chains are consumed. [1] Note that there is padding such as to place this structure on a page separate from the available ring and descriptor array: this gives nice cache behavior and acknowledges that each side need only ever write to one part of the virtqueue structure. [2]

Note the `vring_used` flags and the `vring_avail` flags: these are currently used to suppress notifications. For example, the used flags field is used by the host to tell the guest that no kick is necessary when it adds buffers: as the kick requires a vmexit, this can be an important optimization, and the KVM implementation uses this with a timer for network transmission exit mitigation. Similarly, the avail flags field is used by the guest network driver to advise that further interrupts are not required (i.e., `disable_cb` and `enable_cb` set and unset this bit).

Finally, it's worth noting that we have no infrastructure for guest-aware suspend or resume; we don't need them, as we are merely publishing our own buffers. Indeed, the host implementation of suspend and resume for KVM has proven fairly trivial as well.

4.1 A Note on Zero-Copy And Religion of Page Flipping

When designing efficient I/O we have to keep in mind two things: the number of notifications required per operation, and the amount of cache-cold data which is accessed. The former is fairly well handled by the virtio_ring interrupt suppression flags (and a Linux virtio interface which encourages batching). The handling of cache-cold data is worth some further discussion.

In the KVM and lguest models, the guest memory appears as a normal part of the virtual address space of a process in the host: to the host OS, that process *is* the guest. Hence I/O from the guest to the host should be as fast as I/O from any normal host process, with the possible additional cost of world switch between guest and host (which chip manufacturers are steadily reducing). This is why virtio concentrates on publishing buffers, on the assumption that the target of the I/O can access that memory.

Xen does not have such a natural access model: there is no "host" which has access to other guests' memory, but all domains are peers. This is equivalent to the inter-guest communication in KVM or lguest, where mapping buffers from one guest into another is necessary to allow zero-copy between guests.

Copying time is dominated by the amount of cache-cold data being copied: if either the guest or the host touch significant amounts of the data, the cost of copying is highly amortized. The cost of page mapping is independent of data size, but it only works on page-aligned page-sized data. For this reason, it's only interesting for large data.

In a general "page flipping" scheme each inter-guest I/O involves two separate page table changes: one to map and one to unmap. We must ensure the buffer is unmapped before being released, otherwise the page might be recycled for something else while another guest still has access to it. The cost can be amortized somewhat by batching and deferring notification of completion, but on SMP systems such manipulations are still expensive.

Permanently sharing a fixed area of memory avoids the need to page flip, but does not fit with a general purpose guest OS such as Linux: if we're going to copy to and from a fixed area of memory so the other side can access the data, we might as well simply copy between the guests.

Large inter-guest copies are currently rare: virtio_net is limited to 64k packets due to the TSO implementation, and inter-guest block devices seem an obscure use case. Nonetheless, proving the worth of page-flipping is a simple matter of code and while we suspect the results would be marginal, we hope some enthusiast will take this as a challenge to prove us wrong.

One reason such work may never be done is the upcoming use of DMA engines for copying large amounts of data. They are optimal in similar cases to those where page flipping would expect to provide benefits: large cache-cold transfers.

5. CURRENT VIRTIO DRIVERS

Now we're familiar with the Linux virtio and virtqueue concepts and API and have seen a transport implementation, it's useful to look at some of the existing virtio drivers. We have a simple and very dumb console driver for lguest, while KVM uses emulation for the console; console performance is unlikely to receive attention until someone releases something like a virtcon benchmark. [3]

We also have a simple balloon driver which allows the host to specify the number of pages it wants to extract from the guest. The guest passes arrays of (guest-physical) page numbers to the host; the host is allowed to unmap these pages and replace them with zeroed pages when they are next accessed.

We'll dive into more detail for the two more common and important drivers, the block and network drivers.

5.1 Virtio Block Driver

For the block device[8], we have a single queue for requests. The first 16 bytes of each buffer in the queue is always a

[1] the id field is u32 only for padding reasons; it's tempting to steal 16 of those bits to enhance the length field in future.

[2] You may note that we don't publish the other side's producer/consumer index; it's not strictly necessary as neither side can overfill the ring. This is a mistake, as the ring state is not fully encapsulated. Fortunately, we can use feature bits to fix this in a backward-compatible fashion.

[3] Not to be confused with the current virtcon benchmark which measures how much venture capital one can extract for a virtualization project.

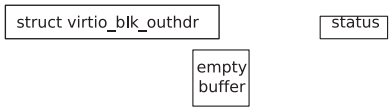

Figure 2: Ingredients for a virtio block read

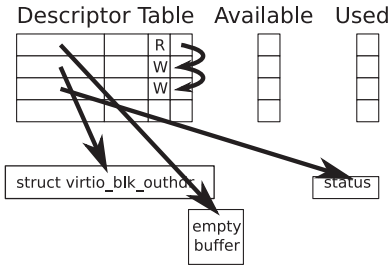

Figure 3: Virtio request placed into descriptor table

read-only descriptor:

```
struct virtio_blk_outhdr
{
    __u32 type;
    __u32 ioprio;
    __u64 sector;
};
```

The type indicates whether it is a read, write or generic SCSI command, and whether a write barrier should precede this command. The I/O priority (higher values are higher priority) allow the guest to hint about relative priorities of requests, which are duly ignored by all current implementations, and the sector is the 512-byte offset of the read or write.

All but one byte of the remainder of the descriptor is either read-only or write-only, depending on the type of request, and the total length determines the request size. The final byte is write-only, and indicates whether the request succeeded (0) or not (1), or is unsupported (2).

The block device can support barriers, and simple SCSI commands (mainly useful for ejecting virtual CDROMs). For more sophisticated uses, a SCSI HBA over virtio should be implemented.

5.1.1 Virtio Block Mechanics

To snap the mechanism into focus, let's walk through the conceptual path that the virtio block driver traverses to do a single block read, using virtio_ring as the example transport. To begin with, the guest has an empty buffer which the data will be read into. We allocate a `struct virtio_blk_outhdr` with the request metadata, and a single byte to receive the status (success or fail) as shown in Figure 2.

We put these three parts of our request into three free entries of the descriptor table and chain them together. In this example the buffer we're reading into is physically contiguous: if it wasn't, we'd use multiple descriptor table entries. The header is read-only, and the empty buffer and status byte are write-only, as shown in Figure 3.

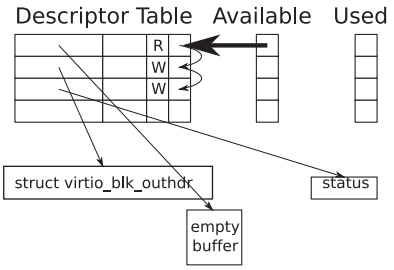

Figure 4: Virtio block read ready to be serviced

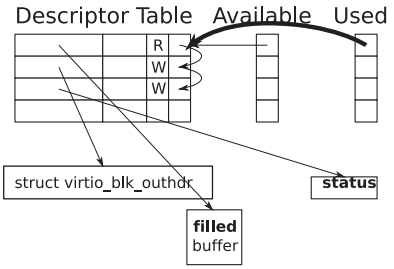

Figure 5: Virtio block request completed

Once this is done, the descriptor is ready to be marked available as Figure 4 shows. This is done by placing the index of the descriptor head into the "available" ring, issuing a memory barrier, then incrementing the available index. A "kick" is issued to notify the host that a request is pending (in practice our driver sets up all the pending requests which fit into the ring, then issues a single kick).

At some point in the future, the request will be completed as in Figure 5: the buffer is filled and the status byte updated to indicate success. At this point the descriptor head is returned in the "used" ring and the guest is notified (ie. interrupted). The block driver callback which runs does `get_buf` repeatedly to see which requests have been finished, until `get_buf` returns `NULL`.

5.2 Virtio Network Driver

The network device[9] uses two queues: one for transmission and one for receiving. Like the block driver, each network buffer is preceded by a header, allowing for checksum offload and TCP/UDP segmentation offload. Segmentation offload was developed for network hardware to give the effect of large MTUs without actually requiring networks to move from 1500 byte packets; fewer packets means fewer PCI transfers to the card. In a virtual environment it means a reduced number of calls out from the virtual machine, and performance follows.

```
struct virtio_net_hdr
{
// Use csum_start, csum_offset
#define VIRTIO_NET_HDR_F_NEEDS_CSUM     1
    __u8 flags;
#define VIRTIO_NET_HDR_GSO_NONE         0
#define VIRTIO_NET_HDR_GSO_TCPV4        1
#define VIRTIO_NET_HDR_GSO_UDP          3
#define VIRTIO_NET_HDR_GSO_TCPV6        4
#define VIRTIO_NET_HDR_GSO_ECN          0x80
    __u8 gso_type;
    __u16 hdr_len;
    __u16 gso_size;
    __u16 csum_start;
    __u16 csum_offset;
};
```

The virtio network driver in 2.6.24 has some infrastructure to TSO on incoming packets, but as it does not allocate large receive buffers, it cannot be used. We'll see how to make this change when we address forward compatibility.

An interesting note is that the network driver usually suppresses callbacks on the transmission virtqueue: unlike the block driver it doesn't care when packets are finished. The exception is when the queue is full: the driver re-enables callbacks in this case so it can resume transmission as soon as buffers are consumed.

6. VIRTIO_PCI: A PCI IMPLEMENTATION OF VRING AND VIRTIO

So far we have addressed two ways in which we can unify virtual I/O. Firstly, by using virtio drivers within the Linux kernel and providing appropriate ops structures to have them drive particular transports. Secondly, by using the virtio_ring layout and implementation as their transport. We now address the issues of device probing and configuration which make for a complete virtual I/O ABI.

As most full-virtualization hosts already have some form of PCI emulation and most guests have some method for adding new third-party PCI drivers, it was obvious that we should provide a standard virtio-over-PCI definition which gives maximum portability for such guests and hosts. This is a fairly straight-forward vring implementation, plus configuration using an I/O region. For example, the virtio_pci network devices use the `struct virtio_net_hdr` from Linux's virtio_net API as its ABI, and simply passes that header through to the host. Such structures were deliberately designed to be used in this way, and it makes a pass-through transport much simpler.

Qumranet, who started the KVM project, has donated their device IDs (vendor ID 0x1AF4) from 0x1000 through 0x10FF. The subsystem vendor and device ids of the PCI device become the virtio type and vendor fields, so the PCI driver does not need to know what virtio types mean; in Linux this means it creates a `struct virtio_device` and registers it on the virtio bus for virtio drivers to pick up.

The I/O space may require special accessors, depending on the platform, but conceptually it looks like the following structure:

```
struct virtio_pci_io
{
    __u32 host_features;
    __u32 guest_features;
    __u32 vring_page_num;
    __u16 vring_ring_size;
    __u16 vring_queue_selector;
    __u16 vring_queue_notifier;
    __u8 status;
    __u8 pci_isr;
    __u8 config[];
}
```

The features publishing and accepting bits are the first two 32-bit fields in the I/O space: the final bit can be used to extend that when it becomes necessary. `vring_queue_selector` is used to access the device's virtqueues: the queue doesn't exist if the `vring_ring_size` is zero. Otherwise, the guest is expected to write the location it has allocated for that queue into the `vring_page_size`: this is unlike the lguest implementation where the host allocates the space for the ring. [4]

The `vring_queue_notifier` is used to kick the host when a queue has new buffers, and the `status` byte is used to write the standard virtio status bits, and 0 to reset the device. The `pci_isr` field has a side-effect of clearing the interrupt on read; non-zero means one of the device's virtqueues is pending, and the second bit means that the device's configuration has changed.

The ring itself contains "guest-physical" addresses: for both KVM and lguest there is a simple offset from these addresses to the the host process's virtual memory. [5] Hence the host need only check that addresses are not beyond the guest memory size, then simply apply the offset and hand them to readv or writev: if they hit a memory hole for some reason, this would give an -EFAULT and the host would return an error to the guest.

All in all, implementing the PCI virtio driver is a fairly simple exercise; it's around 450 lines, or 170 semicolons in 2.6.25.

7. PERFORMANCE

Regrettably, little usable data is available on performance at this early stage, except scratch tests to ensure that our performance is not terrible. There's no obvious roadblock which will prevent us from meeting our goal of setting the record for virtual I/O when modern Linux guests run on a modern Linux host: like others[13] we expect to approach bare-metal speeds as hardware support improves. [6]

Networking performance is currently receiving the most at-

[4]Changing this might be useful, because it can be difficult for the guest to allocate large contiguous buffers. This would be done via a feature bit or some equivalent backwards-compatible option.

[5]KVM may use non-linear mappings in the future by mapping memory regions separately, but the logic is very similar.

[6]At the first Linux Virtualization Minisummit, I (Rusty) gamely prediced that we should reach 95% of bare-metal network performance by the middle of this year. Caveats have been added ever since.

tention: enabling various TSO options is a priority, as is removing the copies which occur within the QEMU framework for KVM. Once this is done, we expect heuristics to suppress notifications to receive more attention: our Linux driver will enter polling mode under receive load as is standard for high-performance NICs, but the method for reducing notifications on packet transmission is still primitive.

8. ADOPTION

Currently KVM and lguest both use virtio as their native transport; for KVM that means supporting Linux virtio drivers on 32 and 64-bit x86, System z (i.e., S/390), IA64 and an experimental PowerPC port so far. lguest is 32-bit x86 only, but there are patches to extend it to 64-bit which may be included in the next year or so.

Qumranet has produced beta virtio_pci Windows drivers for Windows guests. KVM uses the QEMU emulator to support emulated devices, and the KVM version of QEMU has host support for virtio, but it is not optimized; this is an area where there is a great deal of work going on. lguest's launcher also supports a very minimal virtio implementation. We are not aware of other host implementations, and there are currently no in-kernel host implementations, which might be used to gain the last few percentage points of performance.

We expect to see more virtio guest drivers for other operating systems, for the simple reason that virtio drivers are simple to write.

8.1 Adapting Existing Transports to Use Virtio Drivers

You will have noticed that both example drivers write a defined header to the front of buffers: this looks a great deal like an ABI, and indeed for KVM and lguest this is the case, as they are passed straight through to the host.

However, a key aim of virtio drivers is to allow them to work over different transports, and this is possible by manipulating the configuration ops and the feature bits a device sees. For example, if the network driver is told that the host does not support TSO or checksum offload, the entire network header can be ignored on transmission, and zeroed out on receive. This would happen in the `add_buf` callback. If the format of the header is different, or the equivalent information is sent some other way, it can similarly be interpreted at this point.

One task which lies ahead of us is to create these shims for other hypervisors, perform testing and benchmarks, and hopefully convince the maintainers to switch over to the virtio drivers. The Xen drivers are the most challenging: not only have they been optimized to a fair degree already, they are quite feature complete and support a disconnect and reconnect model which the virtio drivers currently do not.

Replacing currently-working drivers is a marginal benefit, but there are two cases where we believe that using the virtio infrastructure is compelling. The first is when a virtualization technology adds a new type of virtual device which is already supported by a virtio driver, where adapting that is a lesser burden than writing a driver from scratch. For example, there is already a virtio entropy driver to provide randomness to the guest [12] which will be merged into Linux in 2.6.27.

The second case is when new virtualization transports want support in Linux; while we would hope that they simply use vring, if they do not they can at least get the existing drivers for free, rather than having to implement and support Linux drivers which is often outside their field of expertise.

9. FUTURE WORK

virtio and the drivers are under ongoing development; while the ABI is officially stable as of 2.6.25, feature additions and optimizations are expected to continue for some time yet. It's worth covering in some depth how we plan to add new features in a compatible fashion, then some of the experiments going on at the moment to provide insight into what might be coming in future revisions.

9.1 Feature Bits and Forward Compatibility

Of course not all hosts will support all features, either because they are old, or because they simply don't support accelerations like checksum offload or TSO.

We've seen the feature bit mechanism, but the implementations are worth mentioning: both lguest and virtio_pci use two bitmaps, one for features presented by the host and another for features accepted by the driver. When the `VIRTIO_CONFIG_S_DRIVER_OK` status bit is set, the host can examine this accepted feature set and see what the guest driver is capable of.

So far all the defined features are specific to a particular device type, such as indicating that the host supports barriers on a block device. We have reserved a few bits for device-independent features, however (currently bits 24 through 32). While we don't want to add feature bits randomly, as it makes for more complicated guest and host interaction, we do want to allow experimentation.

9.2 Inter-guest Communication

It's fairly easy for the host to support inter-guest communication; indeed, there's an experimental lguest patch which does just this. The launcher process for each guest maps the other guest's memory as well as its own, and uses a pipe to notify the other launcher of inter-guest I/O. The guests negotiate which virtqueues to join together, and then the process is simple: get a buffer from this guest's virtqueue, the other guest's virtqueue, and memcpy data between them based on the read/write flags of the buffers. This mechanism is completely independent of what the virtqueues are being used for; and because the virtio_net protocol is symmetrical it works for point-to-point inter-guest networking without any guest changes. The exact same code would allow one guest to serve a block or console device to another guest, if the serving guest had a driver to do so.

There is a subtle but important protocol consideration for efficient communication with untrusted parties, such as this. Consider the case of an inter-guest network protocol in which a guest receives a packet which claims to be 1514 bytes long.

If all 1514 bytes weren't copied in (because of a malicious or buggy transmitter), the receiving guest would treat old data which was in the buffer as the remainder of the packet. This data could leak to userspace, or be forwarded out from the guest. To prevent this, a guest would have to sanitize all its receive buffers, which is complex, inefficient or both.

This is why the used ring in vring contains a length field. As long as the amount copied in is written by a trusted source, we can avoid this. In the lguest prototype, the launcher was doing the copy, so it can be trusted. A virtio transport implementation which only ever connects to the host or another trusted source can also simply provide this length field. If a transport can connect to an untrusted source, and has no assured way of knowing the length copied in, it must zero out writable buffers before exposing them to the other end. This is far safer than requiring it of drivers, especially since thoughtful transports will not have this issue.

While a host can conveniently join two vrings together for inter-guest I/O, negotiating features is harder. We want to offer all features to each guest so they can take advantage of them, such as TSO and checksum offload, but if one guest turns down a feature we already offered to the other, we either need to do some translation between every I/O or hot-unplug the device and re-add it with fewer features offered.

The solution we are considering to this is to add a 'multi-round' negotiation feature: if the guest acknowledged that feature, then after the driver has completed feature acknowledgement by setting the status field, it will expect the features to be re-presented, until finally the 'multi-round' feature is missing. We would use this by presenting a minimal feature set at first, with the 'multi-round' bit: if both guests acknowledged it, we would present all possible features, and iteratively remove those not accepted by the other side until both were happy. This would be our first non-device-specific feature bit.

9.3 Tun Device Vring Support

The userspace virtio host device in QEMU/KVM and lguest uses the Linux tun device; this is a userspace-terminated network interface into which the driver `read()`s incoming ethernet packets and `writes()`s outgoing ones. We posted a simple patch to this device to support the virtio_ring header[11] to allow us to test the virtio network driver's GSO support, and Anthony Liguori and Herbert Xu adapted it for use with KVM. According to Herbert, while the performance while streaming TCP from guest to host was comparable with Xen under full virtualization, the speed was only half that of the guest loopback device.

This version copied the packets twice on transmission: once inside QEMU to linearize the buffer, and once inside the kernel. The former is a QEMU internal architectural limitation to overcome, and the latter is more complicated to resolve. If we want to avoid copying the packet buffers, we must pin the userspace pages and only complete the write operation when they are no longer referenced by any packet. Given the packet manipulation which occurs especially to packets being received by local sockets or large packets being split for non-GSO-capable devices, creating such a destructor callback is not entirely trivial. In addition, it could take an arbitrarily long time to complete; the packet could sit in a local socket forever if a process is not reading from it.

Fortunately, we believe we have a solution for this last problem: vring! It handles out-of-order buffer completion, is efficient, and already has a well-defined ABI. Hence we have a '/dev/vring' patch which creates a file descriptor associated with a vring ringbuffer in user memory. This file descriptor can be polled (have any buffers been used?), read (to update the last-seen index and hence clear the polled flag) and written (to notify the other end of the vringfd of new available buffers). Finally a small patch adds methods to attach such vringfds to the receive and transmission of the tun device.

We have also implemented an ioctl to set an offset and bounds to where the vring can access; with this the guest's network vrings can be directly exposed to the host kernel's tap device. The ultimate experiment in efficiency would be to avoid userspace altogether, and this is actually quite simple to do once all the other pieces are in place.

10. CONCLUSIONS

As we see more virtual I/O solutions, the common shape of them is starting to appear: a ring buffer, some notification, some feature bits. In fact, they look a lot like high speed physical devices: you assume they can DMA from and to your memory, and you talk to them as infrequently as possible. Quality of implementation becomes more interesting, which implies that the time is right to create a quality implementation and see if it will become a standard.

You will surely have noticed how mundane virtio is: it has interrupts, device features, configuration spaces and DMA rings. These are all familiar concepts to driver authors, and an operating system's device infrastructure is built to accommodate them. If virtio drivers become the norm for Linux under virtualized environments, it tethers one end of the virtual I/O design process. Without this guidance arbitrary differences run wild and free, integration into the guest OS becomes an afterthought, and the resulting drivers look like they've dropped in from an alien universe and are unreadable to those not trained in that particular baroque interface [15].

There is still a great deal to be done in virtual I/O, but if we establish a solid foundation this innovation will accelerate, rather then being bogged down reinventing the uninteresting parts of configuration and bit-pushing. It is our hope that virtio will raise the bar for what people believe they can accomplish with a virtual I/O system, and encourage all kinds of virtual devices in the future.

11. ACKNOWLEDGMENTS

The author would like to thank Muli Ben-Yehuda and Eric Van Hensbergen for insisting, repeatedly, that virtio deserved a paper, Anthony Liguori for proofreading, and the Operating Systems Review reviewers for proofreading and cluebatting. A special thanks to all the KVM developers for getting behind virtio and running with it.

12. REFERENCES

[1] Z. Amsden, D. Arai, D. Hecht, A. Holler, and P. Subrahmanyam. VMI: An interface for

paravirtualization. In *OLS '06: The 2006 Ottawa Linux Symposium*, pages 371–386, July 2006.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[3] J. Corbet. Van Jacobson's network channels. Available: http://lwn.net/Articles/169961/ [Viewed May 14, 2008], 2006.

[4] J. Dike. User-mode linux. In *ALS '01: Proceedings of the 5th Annual Linux Showcase & Conference*, pages 3–14, Berkeley, CA, USA, 2001. USENIX Association.

[5] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[6] I. Molnar. KVM/NET, paravirtual network device. Available: http://www.mail-archive.com/kvm-devel@lists.sourceforge.net/msg00824.html [Viewed April 14, 2008], January 2007.

[7] R. Russel. lguest: Implementing the little Linux hypervisor. In *OLS '07: Proceedings of the 2007 Ottawa Linux Symposium*, pages 173–177, 2007.

[8] R. Russell. Linux virtio_blk definitions. Available: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=include/linux/virtio_blk.h [Viewed April 14, 2008], 2008.

[9] R. Russell. Linux virtio_net definitions. Available: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=include/linux/virtio_net.h [Viewed April 14, 2008], 2008.

[10] R. Russell. Linux virtio_ring definitions. Available: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=include/linux/virtio_ring.h [Viewed April 14, 2008], 2008.

[11] R. Russell. [PATCH 2/3] partial checksum and gso support for tun/tap. Available: http://www.mail-archive.com/netdev@vger.kernel.org/msg59903.html [Viewed April 14, 2008], January 2008.

[12] R. Russell. virtio: hardware random device. Available: http://lwn.net/Articles/282721/ [Viewed May 26, 2008], May 2008.

[13] J. Santos, Y. Turner, J. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *Proceedings of the 2008 USENIX Annual Technical Conference*.

[14] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[15] XenSource. Xen virtual network driver. Available: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=drivers/net/xen-netfront.c [Viewed April 14, 2008], 2005.