Virtualising PCI

National ICT Australia

and

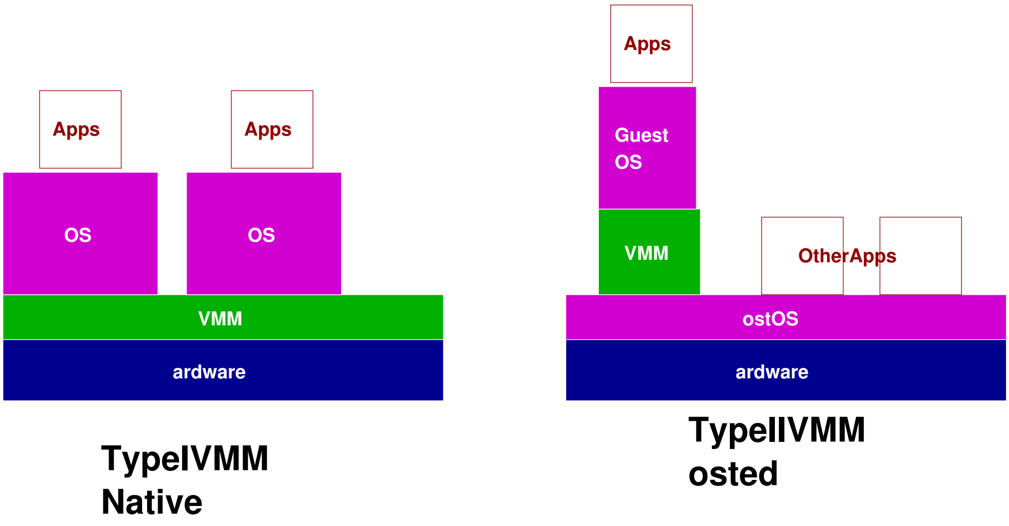
The University of New South Wales

Myrto Zehnder and Peter Chubb

Gelato Project

October 2006

Virtual Machine Monitors

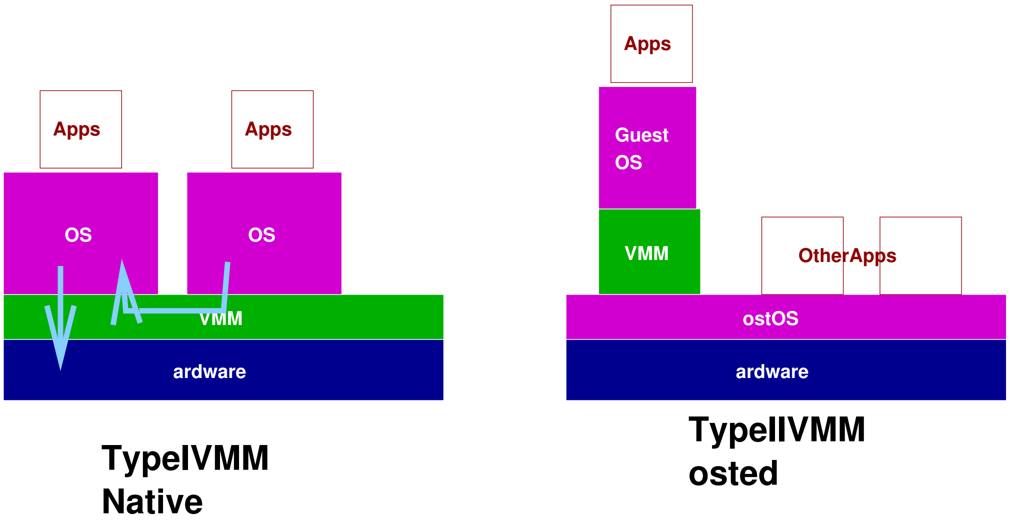


There are two kinds of virtual machine monitors (VMM). A *Type I* VMM controls the hardware directly (The nomenclature was originally introduced by (Goldberg, 1973)). Guest operating systems see a virtualised version of the hardware. Xen (Barham et al., 2003) and vNUMA (Chapma are examples of Type I VMMs.

A *Type II*, or *hosted* VMM runs as an application on a normal operating system. Examples are UML (Dike, 2000), VMware (Sugerman et al., 2001) and Gelato@UNSW's own LinuxOnLinux (Chubb, 2005).

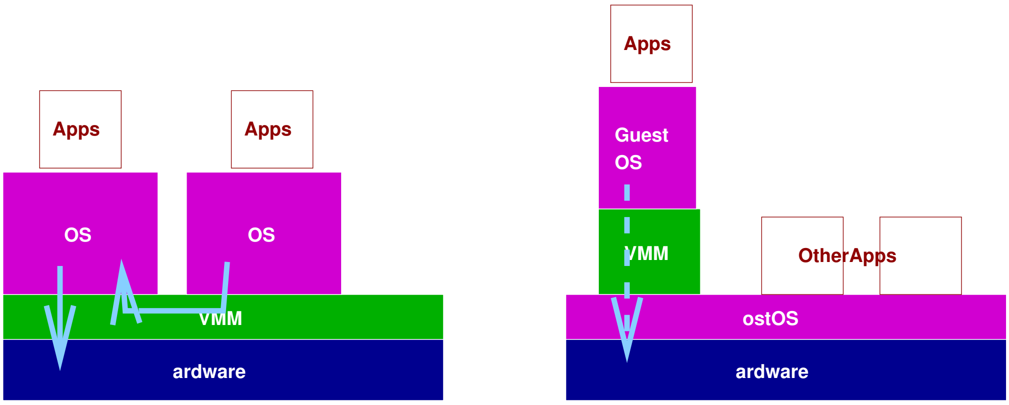
Both a type I and a type II VMM have to virtualise not only the processor(s), but also some kind of I/O system. Most commonly, devices are emulated at 'standard' I/O addresses — for example, the free VMware emulates a standard IDE controller based on the Intel PIIX4 chipset, a VGA controller, and a LANCE ethernet. The alternative is to paravirtualise disk and console acess, as vNUMA does using the SKI simulator Supervisor System Calls to emulate a SCSI disk and ethernet controller.

Virtual Machine I/O



In addition, a Type I VMM can allow transparent access to the real hardware from guest operating systems. Until recently, Xen allowed this only for one specially privileged guest, 'Dom0'. Other guests see simple emulated devices, that talk to the 'real' device drivers in Dom0.

Virtual Machine I/O



**TypeIVMM
Native**

**TypeIIVMM
osted**

Our question was, what does it take to allow the native drivers in a guest in a type II VMM access to real devices?

LinuxOnLinux

- Type II VMM
- Uses (enhanced) SKI SSCs
- Uses SKI simscsi, simserial and simeth
- Can emulate many processors
- Reasonably fast
- Itanium only!
- Minimal (200 line) change to native Linux

LinuxOnLinux was explained in my ‘Paravirualisation without Pain’ (Chubb, 2005) talk last year. It uses automatic pre-virtualisation (‘afterburning’) to allow rapid tracking of upstream kernels and reasonable performance. It allows no real access to hardware — the simulated network device maps onto `/dev/tun`, the serial device to standard input/output of the emulator, and the scsi device to a regular file in the host’s filesystem.

User Level Driver Framework

- Exclusive claiming of a device.
- Interrupt management
  - but not IRQ sharing
- DMA setup/teardown

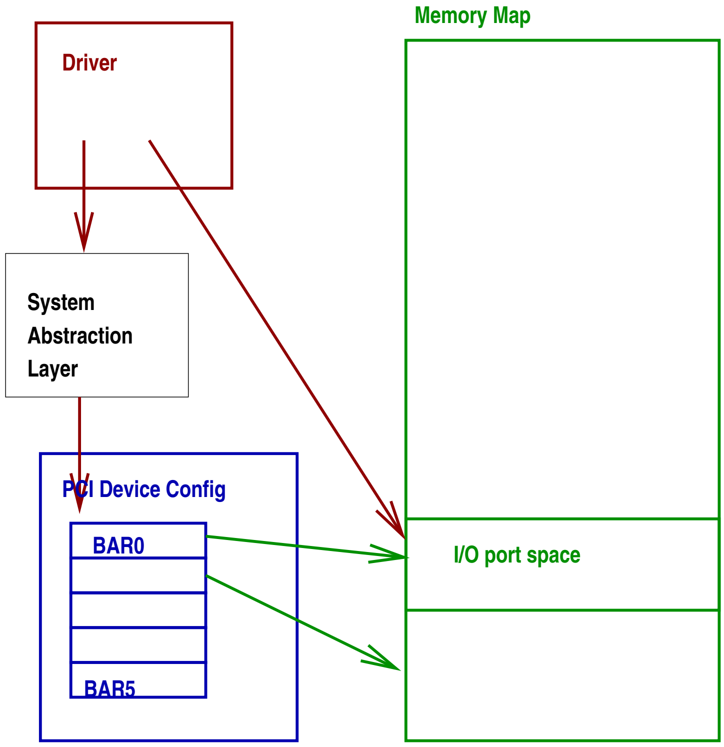
Likewise, the user-level driver framework has been explained elsewhere — e.g., in (Chubb, 2004b) and (Chubb, 2004a). There are three parts:

1. The new system call `usr_pci_open()`, that allows a suitably privileged user-space program to claim a PCI device and obtain a handle on it (the handle happens to be implemented as a file descriptor).

2. Extension of `/proc/irq/` to include a named file for every possible interrupt on the system, that can be opened and closed, and that when `read()` pauses until that interrupt occurs. `poll` also works; and for this project we implemented *asynch I/O*, to provide a signal when an interrupt occurs.

3. The new system call `usr_pci_map()` that pins virtual memory pages, and provides a scatterlist suitable for a device to use for DMA.

Driver Architecture: Unmodified Linux



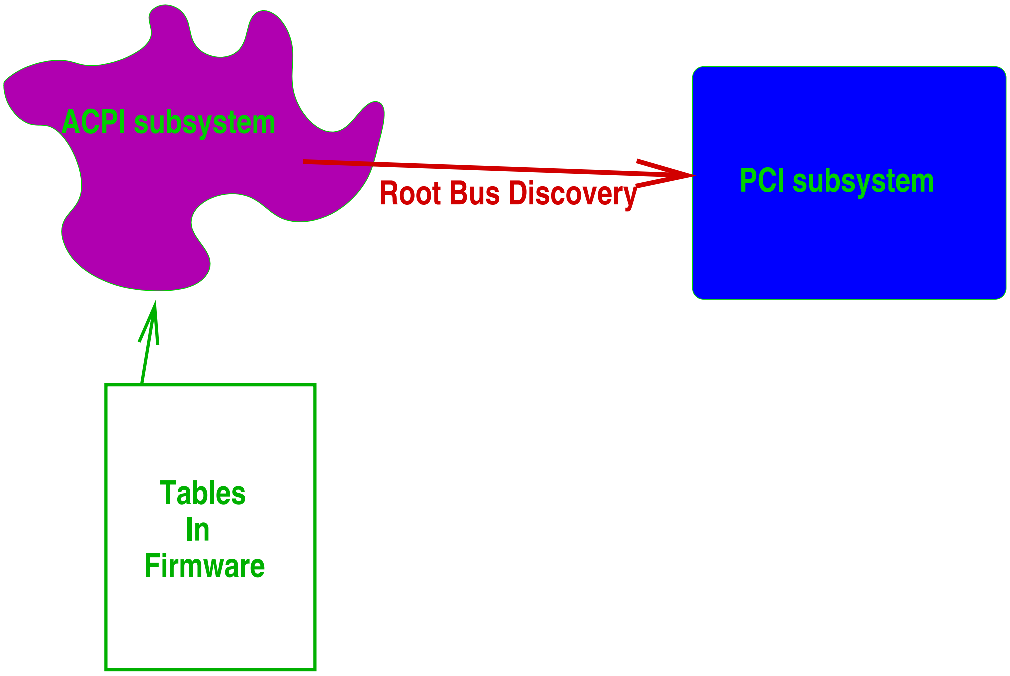
Each PCI device has an address: a triple (bus, slot, function). Each platform has a way to get at a device's *configuration space*; for Itanium, PCI configuration space accesses are mediated by the *System Abstraction Layer* (SAL). Within the configuration space for each device are *Base Address Registers* (BARs) that give the address in either port space or MMIO space of the control registers for the device.

The Itanium has no separate port space; all I/O registers are mapped into the physical memory space. However, accessing mapped port registers obeys port semantics (i.e., synchronous operation) and is therefore slow. Memory-mapped I/O regions are *posted* — i.e., write operations are queued, so that the CPU can continue to process instructions in parallel with the device's write operation. MMIO operations also have 'interesting' synchronisation requirements, in that it's the programmers' responsibility to ensure outstanding writes have completed before issuing reads in some circumstances.

- Linux has to *walk the busses* querying each slot.
- IA64 may have more than 255 PCI busses – divides into *segments* or *domains*
- Populated range of busses available from ACPI
- ACPI is **huge**

  — so we don't want to emulate it if we don't *have* to

The Itanium architecture extends the PCI address space with a *segment ID*, to allow more than 255 busses. Typically, for good performance, each bus will have only one or two slots used. One way to discover what devices are present would be to iterate over all possible PCI addresses. This would be *far* too slow, and in most cases is unnecessary, so ACPI provides a table, the *_SB* section of the *DSDT* (differentiated services descriptor table), that can be used to determine which buses are available.



We decided to emulate only a single root bus, and map all accessible devices onto that bus, regardless of their true address. This means we can avoid most of ACPI, and instead hack the guest kernel to call a single discovery routine. Obviously this isn't a long-term soultion, but it gets us going for now.

PCI Config space access

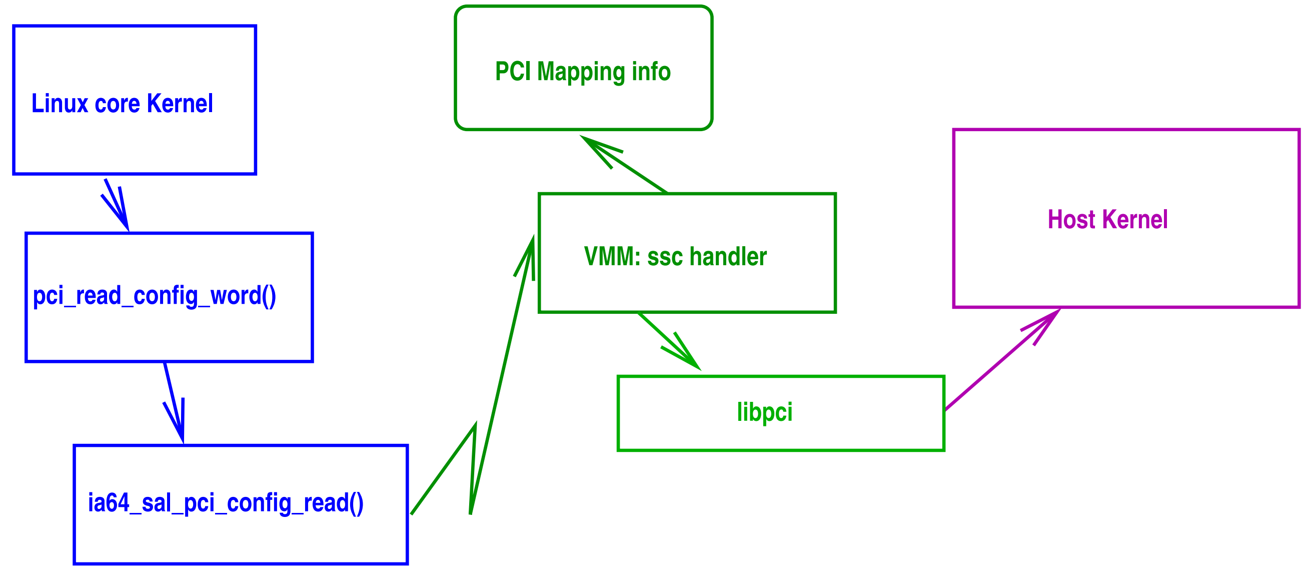
- On Host via libpci (`/proc/bus/pci`, `/sys/bus/pci/`)
- On Guest via SAL

→ Virtualise SAL

Most architectures provide access to the PCI configuration space through the system firmware. On IA64 such access is mediated by the System Abstraction Layer. A virtual machine implementation already has to emulate SAL, but LinuxOnLinux follows the Ski Simulator: the bootloader provided in `arch/ia64/hp/sim/boot` emulates a very small subset of the available SAL calls.

Usually, PCI is disabled when configuring Linux for the simulator. However, the bootloader contains code to access the configuration space by doing port I/O to the legacy space IO ports 0xCF8 and 0xCFC (which is where, on PC99 architecture, the PCI controller is traditionally mapped).

I'm not sure that this code ever worked. Consequently, we disabled it, and instead implemented a couple of new Supervisor System Calls to access the PCI configuration space.



The result is that the VMM can virtualise the PCI configuration space BARs.

MMIO and Port I/O access

- Keep virtual I/O layer out of the way
- Remap `/dev/mem` into Region 0
- Virtualise BAR registers

And everything's hunky-dory.

Interrupts

Three issues here:

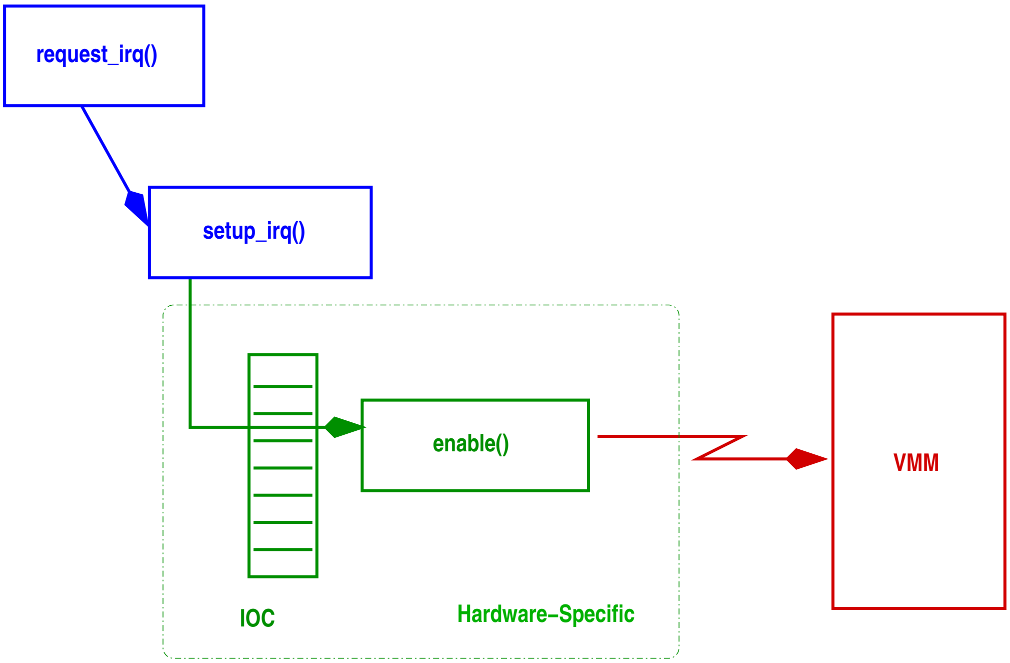
1. Claiming an interrupt
2. Masking/Unmasking an interrupt
3. Delivering an interrupt

There are three issues in handling interrupts.

1. How is an interrupt claimed?
2. How to enable or disable a particular interrupt?
3. How an interrupt can be hooked up to cause an action in the guest when it occurs in the underlying hardware.

The first two of these can be handled in the same way.

Linux Generic Interrupt handling



In the guest, a driver will discover which interrupt to claim by reading the PCI configuration space, and then invoke `request_irq()` to associate the interrupt with its handler.

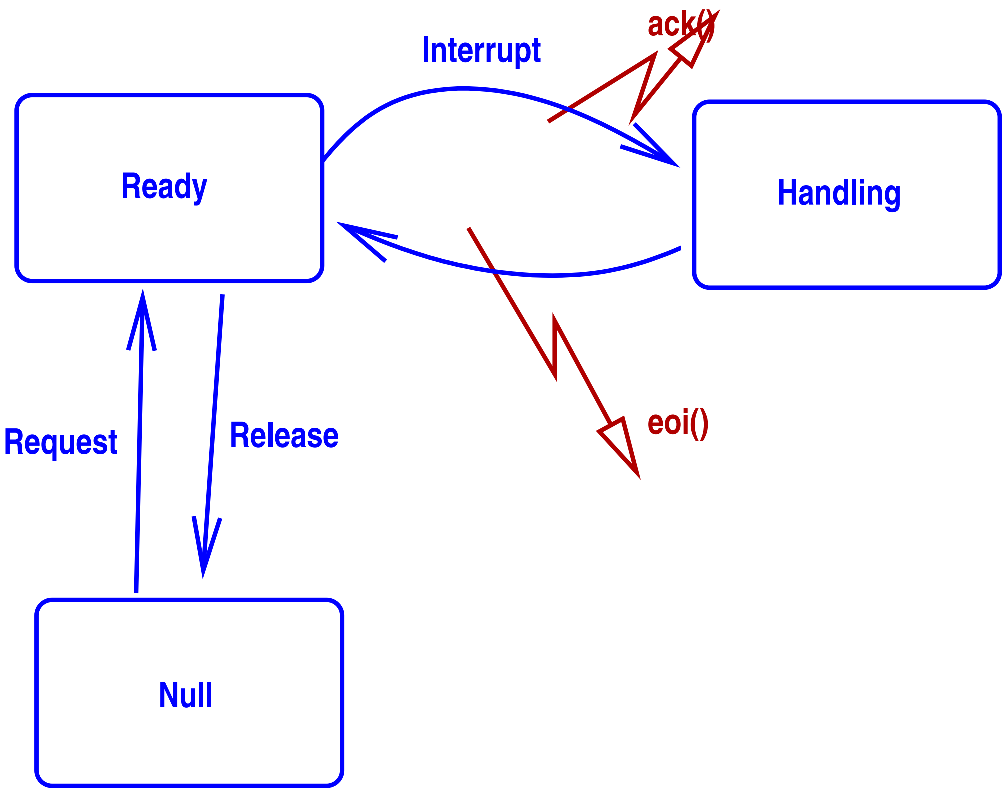
`request_irq()` in its turn calls `setup_irq()` in the generic code, then eventually calls the IOC-specific `enable()` routine.

To paravirtualise interrupt requesting, we provide a new emulated interrupt controller, that uses Supervisor System Calls to call into the VMM.

Linux uses a `struct irq_chip` (formerly `struct hw_interrupt_type`) that contains a vector of routines for what we want.

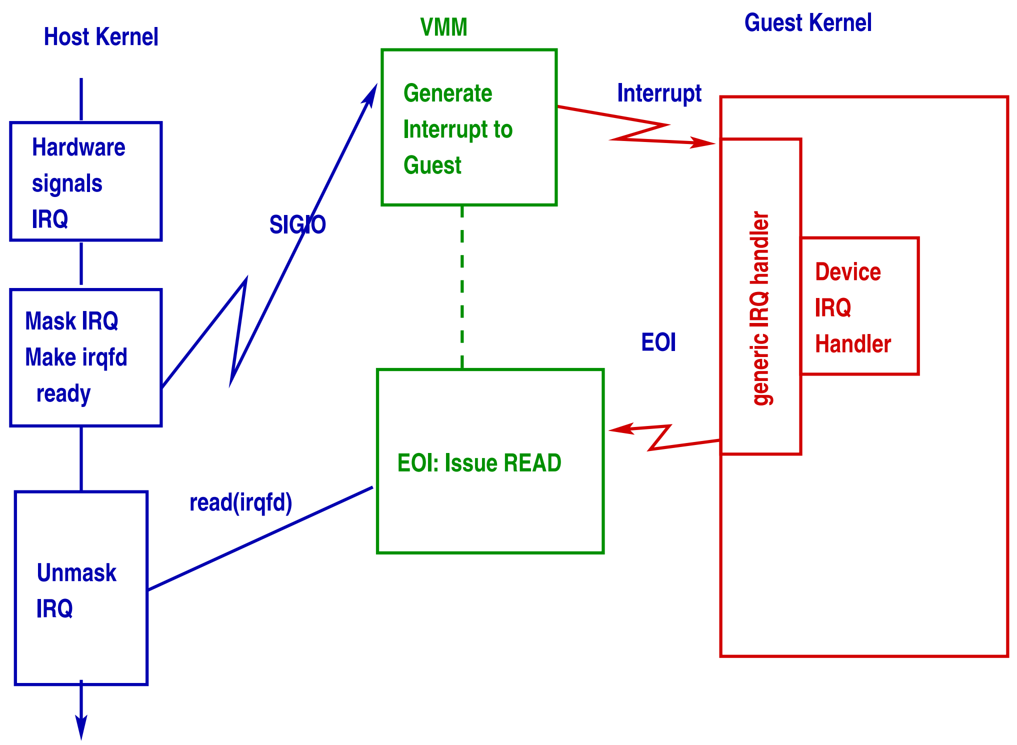
Thus paravirtualisation merely means creating and populating the `struct irq_chip` and making the methods call into the VMM with an SSC.

IRQ handling state diagram



When a device asserts an interrupt, the Advanced Peripheral Interrupt Controller it is attached to signals the processor that an interrupt is pending, and gives information as to *which* interrupt is pending. Linux provides a generic set of methods for controlling this chip; but as far as we're concerned the only important one is the `eoi()`.

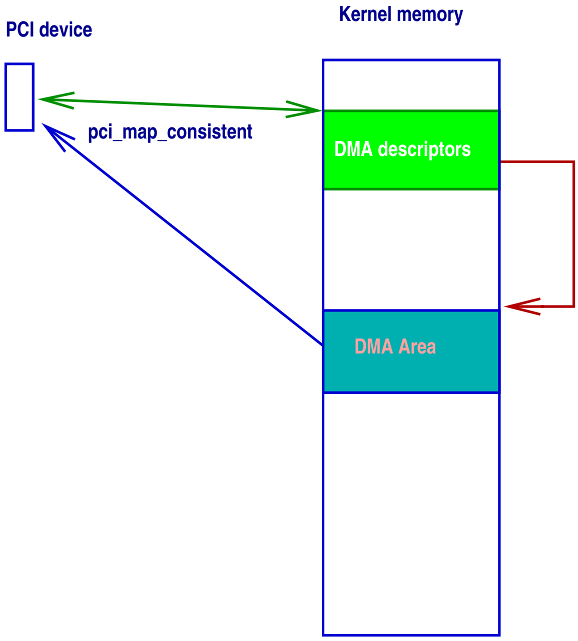
User-level Driver IRQ handling: with Signals



On end-of-interrupt, the VMM needs to be informed. The chip-specific EOI routine invokes an SSC to tell the VMM to complete the work; which it does by issuing a non-blocking read on the IRQ file descriptor. In the kernel, the user driver framework uses this as a request to unmask the interrupt. If the interrupt is still pending at this time (e.g., because it has been reasserted by the device) the read will return a value greater than one and mask the interrupt again. At this time, the VMM has to re-send the interrupt to the guest, and the whole cycle repeats.

An alternative implementation that was considered was to use a separate thread that did a blocking read on the IRQFD; because it would have to raise a signal *anyway* (because of the architecture of VMM) the asynchronous I/O approach was considered better in this instance.

DMA



The addresses a device works with are as seen from the PCI bus. On simple systems they are physical memory addresses; but most modern IA64 machines incorporate an IO MMU that virtualises these addresses.

To perform DMA, a typical device starts by requesting that some kernel memory be allocated and mapped such that both the kernel and the device can access it. Such memory is called, in Linux-speak, 'PCI-consistent memory'. Because many PCI devices cannot see the full 64-bit address space, PCI-consistent memory may have to be allocated at a low physical address, or an IO TLB slot reserved for it. This memory is used for DMA descriptors, that are set up by the driver to perform individual I/O operations, and updated by the device as DMA operations finish. Typically a single page is reserved for each device's DMA shared area, and remains allocated for the entire time the device driver is loaded into the kernel.

When an individual I/O operation is to occur, the device driver uses generic kernel to pin the I/O buffer into memory (so it isn't paged out), and to map it into PCI virtual memory space. The routine used for this is `pci_map_sg()`, which eventually calls into the device-dependent IOMMU code.

The scatterlist returned from `pci_map_sg()` is usualy fairly easy to convert into a set of DMA descriptors suitable for the device. When the driver has updated the DMA descriptors, it can set the I/O going. When the I/O completes, the device will update the DMA descriptors with the status of the result and (eventually) raise an interrupt.

When the interrupt happens, the driver can free any IOMMU resurces by calling `pci_unmap_sg()`.

The user drive framework exposes `pci_map_sg()` and `pci_unmap_sg()` to user space. The obvious way to hook up dma is to add the pci_map and pci_unmap (etc) operations to the machine vector for the simulator. However, time did not permit us to do this.

Results

- One device hacked in for device discovery
- Interrupts enabled and hooked up
- Config space virtualised
- IO port and MMIO spaces remapped apropriately

So far so good... we have a proof of concept (an IDE card hooked up enough to be able to get the disk status and read the partition table). This is enough to validate the overall scheme; but not enough to be able to say anything about performance.

Boot log

Future Work

- Interrupts
- DMA
- Device Discovery and ACPI
  - Genericise to remove hard-wired bus discovery

References

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. pages 164–177, Bolton Landing, NY, USA.

Chapman, M. and Heiser, G. (2005). Implementing transparent shared memory on clusters using virtual machines. pages 383–386, Anaheim, CA, USA.

Chubb, P. (2004a). Get more device drivers out of the kernel! In *Ottawa Linux Symposium*, Ottawa, Canada.

Chubb, P. (2004b). Linux kernel infrastructure for user-level device drivers. Adelaide, Australia.

Chubb, P. (2005). [Para]virtualisation without pain. In *Gelato ICE*, Brazil.

Dike, J. (2000). A user-mode port of the linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, Georgia, USW.

Goldberg, R. P. (1973). Architecture of virtual machines. pages 74–112, New York.

Sugerman, J., Venkitachalam, G., and Lim, B.-H. (2001). Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. pages 1–14, Boston, MA, USA.