Utilizing IOMMUs for Virtualization in Linux and Xen

Muli Ben-Yehuda
muli@il.ibm.com

Jon Mason
jdmason@us.ibm.com

Orran Krieger
okrieg@us.ibm.com

Jimi Xenidis
jimix@watson.ibm.com

Leendert Van Doorn
leendert@us.ibm.com

Asit Mallick
asit.k.mallick@intel.com

Jun Nakajima
jun.nakajima@intel.com

Elsie Wahlig
elsie.wahlig@amd.com

Abstract

IOMMUs are hardware devices that translate device DMA addresses to proper machine physical addresses. IOMMUs have long been used for RAS (prohibiting devices from DMA'ing into the wrong memory) and for performance optimization (avoiding bounce buffers and simplifying scatter/gather). With the increasing emphasis on virtualization, IOMMUs from IBM, Intel, and AMD are being used and re-designed in new ways, e.g., to enforce isolation between multiple operating systems with direct device access. These new IOMMUs and their usage scenarios have a profound impact on some of the OS and hypervisor abstractions and implementation.

We describe the issues and design alternatives of kernel and hypervisor support for new IOMMU designs. We present the design and implementation of the changes made to Linux (some of which have already been merged into the mainline kernel) and Xen, as well as our proposed roadmap. We discuss how the interfaces and implementation can adapt to upcoming IOMMU designs and to tune performance for different workload/reliability/security scenarios. We conclude with a description of some of the key research and development challenges new IOMMUs present.

1 Introduction to IOMMUs

An I/O Memory Management Unit (IOMMU) creates one or more unique address spaces which can be used to control how a DMA operation from a device accesses memory. This functionality is not limited to translation, but can also provide a mechanism by which device accesses are isolated.

IOMMUs were first created to solve the problem where the addressing capability of the device was smaller than the addressing capability of the host processor, which means the device could not access all of physical memory. The introduction of 64bit processors and the Physical Address Extension (PAE) for x86, which allowed processors to address well beyond the 32bit limits, merely exacerbated the problem.

Legacy PCI32 bridges only had a 32bit interface which limited the DMA address range to less than 4GB. The PCI SIG [11] came up with a non-IOMMU fix for the 4GB limitation, Dual Address Cycle (DAC). DAC-enabled systems/adapters bypass this limitation by having two 32bit address phases on the PCI bus (thus allowing 64bits of total addressable memory). This modification is backward compatible to allow 32bit, Single Address Cycle (SAC) adapters to function in DAC-enabled slots. However, this did not solve the case where the addressable range of a specific adapter was limited.

In the absence of an IOMMU, a region of system memory that each adapter can address would have to be reserved, and the device would then be programmed to DMA to this reserved area. The processor would then copy the result to the target memory that was beyond the "reach" of the device. An IOMMU can create a unique translated address space, that is independent of any address space instantiated by the MMU of the processor, that can map the addressable range of a device to all of system memory.

IOMMU isolation solves a very different problem than IOMMU translation. Isolation restricts the access of an adapter to the specific area of memory that the IOMMU allows. Without isolation, an adapter controlled by an untrusted entity (such as a virtual machine when running with a hypervisor, or a non-root user-level driver) could corrupt memory in a buggy or malicious manner, compromising the security or availability of the system.

The IOMMU mechanism can be located on the device, the bus, the processor module or even in the processor core. Typically it is located on the bus that bridges the processor/memory areas and the PCI bus. In this case, the IOMMU is intercepting all PCI bus traffic over the bridge and translates the in and out-bound addresses. Depending on implementation, this translation window can be as small as a few megabytes to as large as the entire addressable memory space by the adapter (4GB for 32bit PCI adapters). When isolation is not an issue, it may be possible to have addresses beyond this window pass through unmodified.

AMD IOMMUs: GART, Device Exclusion Vector, and I/O Virtualization Technology

AMD's Graphical Aperture Remapping Table (GART) is a simple translation-only hardware IOMMU [4]. GART is the integrated translation tables designed for use by AGP GART which are located in the processor's memory controller as an IOMMU for PCI. GART works by specifying a physical memory window and list of pages to be translated inside that window. Addresses outside the window are not translated. GART exists in AMD's Opteron and Athlon64 processors.

Newer AMD processors have a Device Exclusion Vector (DEV) table define the bounds of a set of protection domains providing isolation. DEV is a bit vectored protection table that assigns per-page access rights to devices in that domain. DEV forces a permission check of all device DMAs indicating whether devices in that domain are allowed to access the corresponding physical page. DEV uses 1 bit per physical 4K page to represent each page in the machine's physical memory. A table of size 128K represents up to 4GB.

AMD's I/O Virtualization Technology defines an IOMMU which will translate and protect memory from any DMA transfers by peripheral devices [1]. Devices are assigned into a protection domain with a set of I/O page tables defining the allowed memory addresses. Before a DMA transfer begins, the IOMMU intercepts the access and checks both the I/O page

tables for that device and its cache (IOTLB). The translation and isolation functions of the IOMMU may be used independently of hardware or software virtualization; however, these facilities are a natural extension to virtualization.

The AMD IOMMU is configured as a capability of a bridge or device which may be HyperTransport or PCI based. A device downstream of the AMD IOMMU in the machine topology may optionally maintain a cache (IOTLB) of its own address translations. An IOMMU may also be incorporated into a bridge downstream of another IOMMU capable bridge. Both topologies form scalable networks of distributed translations. Hypervisors or privileged OSes maintain the page structures used by the IOMMU.

The AMD IOMMU can be used instead of the GART or DEV. While GART can translate up to a 2GB window, the AMD IOMMU is not limited to a window but all physical memory.

Intel's VT-d

Intel® Virtualization Technology for Directed I/O Architecture provides DMA remapping hardware that adds support for isolation of device accesses to memory as well as translation functionality [2]. The DMA remapping hardware intercepts device attempts to access system memory. Then, it uses I/O page tables to determine whether the access is allowed and its actual location. The translation structure is unique to an I/O device function (PCI bus, device, and function) and is based on a multi-level page table. Each I/O device is given the DMA virtual address space same as the physical address space or a purely virtual address space defined by software. The DMA remapping hardware uses a context-entry table that is indexed by PCI bus, device and function to find the root of the address translation table. The hardware may cache context-entries as well as the effective translations (IOTLB) to minimize the overhead incurred for fetching them from memory. DMA remapping faults detected by the hardware are processed by logging the fault information and reporting the faults to software through a fault event (interrupt).

IBM IOMMUs: Calgary, DART and Cell

IBM's Calgary PCI-X bridge chips provide hardware IOMMU functionality to both translate and isolate. Translations are defined by a set of Translation Control Entries (TCEs) in a table in system memory. The table can be considered an array where the index is the page number in the bus address space and the TCE at that index describes the physical page number in system memory. The TCE may also contain additional information such as per-direction access rights and specific devices (or device groupings) that each translation can be considered valid. Calgary provides a unique bus address space to all devices behind each PCI Host Bridge (PHB). The table can be large enough to cover 4GB. Calgary will fetch translations as appropriate and cache them locally in a manner similar to a TLB, or IOTLB. The IOTLB, much like the TLB on an MMU, provides a software accessible mechanism that can invalidate cache entries as the entries in system memory are modified. Addresses above the 4GB boundary are presented using DAC commands. If these commands originate from the device and are permitted, they will bypass the TCE translation. Calgary ships in IBM pSeries and xSeries systems.

IBM's CPC925 (U3) northbridge, which can be found on JS20/21 Blades and Apple G5 machines, provides IOMMU mechanisms using a DMA Address Relocation Table (DART). It is

similar to the Calgary IOMMU table, but differs in that the entries only track validity rather than access rights. As with the Calgary, the U3 maintains an IOTLB and provides a software accessible mechanism for invalidating entries.

The Cell Processor has an IOMMU implemented on chip. Its bus address space uses a segmented translation model that is compatible with the MMU in the PowerPC core (PPE). This two-level approach not only allows for efficient user level device drivers, but also allows applications running on the Synergistic Processing Engine (SPE) to interact with devices directly. The Cell IOMMU maintains two local caches – one for caching segment entries and another for caching page table entries, the IOSLB and IOTLB, respectively. Each have a separate software accessible mechanism to invalidate entries. However, all entries in both caches are software accessible, so it is possible to program all translations directly in the caches, increasing the determinism of the translation stage.

2 Linux IOMMU support and the DMA mapping API

Linux runs on many different platforms. Those platforms may have a hardware IOMMU and may have software emulation such as SWIOTLB. In order to write generic, platform independent drivers, Linux abstracts the IOMMU details inside a common API, known as the "DMA" or "DMA mapping" API [7] [8]. The software that enables these IOMMUs must abstract their internal, device specific DMA mapping functions behind the generic DMA API. As long as the implementation conforms to the semantics of the DMA API, a well written driver that is using the DMA API properly should "just work" with any IOMMU.

Prior to this work, Linux's x86-64 architecture included three DMA API implementations: NOMMU, SWIOTLB, and GART. NOMMU is a simple, architecture-independent implementation of the DMA API. It is used when the system has neither a hardware IOMMU nor software emulation. All it does is return the physical memory address for the memory region it is handed as the DMA address for the adapter to use.

Linux includes a software implementation of an IOMMU's translation function, called SWIOTLB. SWIOTLB was first introduced in arch/ia64 [3] and is used today by both IA64 and x86-64. It provides translation through a technique called "bounce buffering." At boot time, SWIOTLB sets aside a large physically contiguous memory region (the "aperture"), which is off limits to the OS. The size of the aperture is configurable and ranges from several to hundreds of megabytes. SWIOTLB uses this aperture as a location for DMAs that need to be remapped to system memory higher than the 4GB boundary. When a driver wishes to DMA to a memory region, the SWIOTLB code checks the system memory address of that region. If it is directly addressable by the adapter, the DMA address of the region is returned to the driver and the adapter DMAs there directly. If it is not, SWIOTLB allocates a "bounce buffer" inside the aperture, and returns the bounce buffer's DMA address to the driver. If the requested DMA operation is a DMA read (read from memory), the data is copied from the original buffer to the bounce buffer, and the adapter reads it from the bounce buffer's memory location. If the requested DMA operation is a write, the data is written by the adapter to the bounce buffer, and then copied to the original buffer.

SWIOTLB treats the aperture as an array, and during a DMA allocation it traverses the array searching for enough contiguous empty slots in

the array to satisfy the request using a next-fit allocation strategy. If it finds enough space to satisfy the request, it passes the location within the aperture for the driver to perform DMA operations. On a DMA write, SWIOTLB does not perform the copy ("bounce") until it is unmapped. On a DMA read or bidirectional DMA, the copy occurs during the mapping of the memory region. Synchronization of the bounce buffer and the memory region can be forced at any time through the various `dma_sync_xxx` function calls.

SWIOTLB is wasteful in CPU operations and memory, but is the only way some adapters can access all memory on systems without an IOMMU. Linux always uses SWIOTLB on IA64 machines, which have no hardware IOMMU. On x86-64, Linux will only use SWIOTLB when the machine has greater than 4GB memory and no hardware IOMMU (or when forced through the `iommu=force` boot command line argument).

The only IOMMU that is specific to x86-64 hardware is AMD's GART. GART's implementation works in the following way: the BIOS (or kernel) sets aside a chunk of contiguous memory (the "aperture"), which is off limits to the OS. GART uses addresses in this aperture as the IO addresses of DMAs that need to be remapped to system memory higher than the 4GB boundary. An unfortunate side effect of this remapping is that the physical pages covered by the aperture are consumed since they can no longer be addressed by devices. The GART Linux code keeps a list of the used buffers in the aperture via a bitmap. When a driver wishes to DMA to a buffer, the code verifies that the system memory address of the buffer's memory falls within the device's DMA mask. If it does not, then the GART code will search the aperture bitmap for an opening large enough to satisfy the number of pages spanned by the DMA mapping request. If it finds the required number of contiguous pages, it programs the appropriate remapping (from the aperture to the original buffer) in the IOMMU and returns the DMA address within the aperture to the driver.

3 Xen IOMMU support

Xen [6] [5] is a virtual machine monitor for x86, x86-64, IA64 and PowerPC that supports execution of multiple guest operating systems on the same physical machine with high performance and resource isolation. Operating systems running under Xen are either para-virtualized (their source code is modified in order to run under a hypervisor) or fully-virtualized (the same kernel binary that runs on bare metal also runs under the hypervisor). Xen makes a distinction between "physical" (interchangeably referred to as "pseudo-physical") frames and machine frames. An operating system running under Xen runs in a contiguous "physical" address space, spanning from physical address zero to end of guest "physical" memory. Each guest "physical" frame is mapped to a host "machine" frame. Naturally, the physical frame number and the machine frame number will be different most of the time.

Xen has different uses for IOMMU than traditional Linux. Xen virtual machines may straddle or completely reside in system memory over the 4GB boundary. Additionally, Xen virtual machines run with a physical address space that is not identity mapped to the machine address space. Therefore, Xen would like to utilize the IOMMU so that a virtual machine with direct device access need not be aware of the physical to machine translation, by presenting an IO address space that is equivalent to the physical address space. Additionally, Xen

would like virtual machines with hardware access to be isolated from other virtual machines.

In theory, any IOMMU driver used by Linux on bare metal could also be used by Linux under Xen after being suitably adapted. The changes required depend on the specific IOMMU, but in general the modified IOMMU driver would need to map from PFNs to MFNs and allocate a machine contiguous aperture rather than a pseudo-physically contiguous aperture. In practice, as of Xen's 3.0.0 release, only a modified version of SWIOTLB is supported.

Xen's controlling domain (dom0) always uses a modified version of SWIOTLB. Xen's SWIOTLB serves two purposes. First, since Xen domains may reside in system memory completely above the 4GB mark, SWIOTLB provides a machine-contiguous aperture below 4GB. Second, since a domain's pseudo-physical memory may not be machine contiguous, the aperture provides a large machine contiguous area for bounce buffers. When a stock Linux driver running under Xen makes a DMA API call, the call always goes through dom0's SWIOTLB, which makes sure that the returned DMA address is below 4GB if necessary and is machine contiguous. Naturally, going through SWIOTLB on every DMA API call is wasteful in CPU cycles and memory and has a non-negligible performance cost. GART or Calgary (or any other suitably capable hardware IOMMU) could be used to do in hardware what SWIOTLB does in software, once the necessary support is put in place.

One of the main selling points of virtualization is machine consolidation. However, some systems would like to access hardware directly in order to achieve maximal performance. For example, one might want to put a database virtual machine and a web server virtual machine on the same physical machine. The database needs fast disk access and the web server needs fast network access. If a device error or system security compromise occurs in one of the virtual machines, the other is immediately vulnerable. Because of this need for security, there is a need for software or hardware device isolation.

Xen supports the ability to allocate different physical devices to different virtual machines (multiple "driver domains" [10]). However, due to the architectural limitations of most PC hardware, notably the lack of an IOMMU, this cannot be done securely. In effect, any domain that has direct hardware access is considered "trusted". For some scenarios, this can be tolerated. For others (e.g., a hosting service that wishes to run multiple customers virtual machines on the same physical machine), this is completely unacceptable.

Xen's grant tables are a software solution to the lack of suitable hardware for isolation. Grant tables provide a method to share and transfer pages of data between domains. They give (or "grant") other domains access to pages in the system memory allocated to the local domain. These pages can be read, written, or exchanged (with the proper permission) for the purpose of providing a fast and secure method for domains to receive indirect access to hardware.

How does data get from the hardware to the local domain that wishes to make use it, when only the driver domain can access the hardware directly? One alternative would be for the driver domain to always DMA into its own memory, and then pass the data to the local domain. Grant tables provide a more efficient alternative by letting driver domains DMA directly into pages in the local domain's memory. However, it is only possible to DMA into pages specified within the grant table. Of course, this is only significant for non-privileged domains (as privileged domains could always access the memory of non-privileged domains). Grant tables have two methods for allowing access to

remote pages in system memory: shared pages and page flipping.

For shared pages, a driver in the local domain's kernel will advertise a page to be shared via a hypervisor function call ("hypercall" or "hcall"). The hcall notifies the hypervisor that other domains are allowed to access this page. The local domain then passes a grant table reference ID to the remote domain it is "granting" access to. Once the remote domain is finished, the local domain removes the grant. Shared pages are used by block devices and any other device that receives data synchronously.

Network devices, as well as any other device that receives data asynchronously, use a method known as "page flipping". When page flipping, a driver in the local domain's kernel will advertise a page to be transferred. This call notifies the hypervisor that other domains can receive this page. The local domain then transfers the page to the remote domain and takes a free page (via producer/consumer ring).

Incoming network packets need to be inspected before they can be transferred, so that the intended destination can be deduced. Since block devices already know which domain requested data to be read, there is no need to inspect the data prior to sending it to its intended domain. Newer networking technologies (such as RDMA NICs and Infiniband) know when a packet is received from the wire for which domain is it destined and will be able to DMA it there directly.

Grant tables, like SWIOTLB, are a software implementation of certain IOMMU functionality. Much like how SWIOTLB provides the translation functionality of an IOMMU, grant tables provide the isolation and protection functionality. Together they provide (in software) a fully functional IOMMU (i.e., one that provides both translation and isolation). Hardware acceleration of grant tables and SWIOTLB is possible, provided a suitable hardware IOMMU exists on the platform, and is likely to be implemented in the future.

4 Virtualization: IOMMU design requirements and open issues

Adding IOMMU support for virtualization raises interesting design requirements and issues. Regardless of the actual functionality of an IOMMU, there are a few basic design requirements that it must support to be useful in a virtualized environment. Those basic design requirements are: memory isolation, fault isolation, and virtualized operating system support.

To achieve memory isolation, an operating system or hypervisor should not allow one virtual machine with direct hardware access to cause a device to DMA into an area of physical memory that the virtual machine does not own. Without this capability, it would be possible for any virtual machine to have access to the memory of another virtual machine, thus precluding running an untrusted OS on any virtual machine and thwarting basic virtualization security requirements.

To achieve fault isolation, an operating system or hypervisor should not allow a virtual machine that causes a bad DMA (which leads to a translation error in the IOMMU) to affect other virtual machines. It is acceptable to kill the errant virtual machine or take its devices off-line, but it is not acceptable to kill other virtual machines (or the entire physical machine) or take devices that the errant virtual machines does not own offline.

To achieve virtualized operating system support, an operating system or hypervisor needs to support para-virtualized operating systems, fully-virtualized operating systems that are not

IOMMU aware, and fully-virtualized IOMMU aware operating systems. For para-virtualized OS's, the IOMMU support should mesh in seamlessly and take advantage of the existing OS IOMMU support (e.g., Linux's DMA API). For fully-virtualized but not IOMMU aware OS's, it should be possible for control tools to construct IOMMU translation tables that mirror the OS's pseudo-physical to machine mappings. For fully-virtualized IOMMU aware operating systems, it should be possible to trap, validate and establish IOMMU mappings such that the semantics the operating system expects with regards to the IOMMU are maintained.

There are several outstanding issues and open questions that need to be answered for IOMMU support. The first and most critical question is: "who owns the IOMMU". Satisfying the isolation requirement requires that the IOMMU be owned by a trusted entity that will validate every map and unmap operation. In Xen, the only trusted entities are the hypervisor and privileged domains (i.e., the hypervisor and dom0 in standard configurations), so the IOMMU must be owned by either the hypervisor or a trusted domain. Mapping and unmapping entries into the IOMMU is a frequent, fast-path operation. In order to impose as little overhead as possible, it will need to be done in the hypervisor. At the same time, there are compelling reasons to move all hardware related operations outside of the hypervisor. The main reason is to keep the hypervisor itself small and ignorant of any hardware details except those absolutely essential, to keep it maintainable and verifiable. Since dom0 already has all of the required IOMMU code for running on bare metal, there is little point in duplicating that code in the hypervisor.

Even if mapping and unmapping of IOMMU entries is done in the hypervisor, should dom0 or the hypervisor initialize the IOMMU and perform other control operations? There are arguments both ways. The argument in favor of the hypervisor is that the hypervisor already does some IOMMU operations, and it might as well do the rest of them, especially if no clear-cut separation is possible. The arguments in favor of dom0 are that it can utilize all of the bare metal code that it already contains.

Let us examine the simple case where a physical machine has two devices and two domains with direct hardware access. Each device will be dedicated to a separate domain. From the point of view of the IOMMU, each device has a different IO address space, referred to simply as an "IO space". An IO space is a virtual address space that has a distinct translation table. When dedicating a device to a domain, we either establish the IO space a-priori or let the domain establish mappings in the IO space that will point to its machine pages as it needs them. IO spaces are created when a device is granted to a domain, and are destroyed when the device is brought offline (or when the domain is destroyed). A trusted entity grants access to devices, and therefore necessarily creates and grants access to their IO spaces. The same trusted entity can revoke access to devices, and therefore revoke access and destroy their IO spaces.

There are multiple considerations that need to be taken into account when designing an IOMMU interface. First, we should differentiate between the administrative interfaces that will be used by control and management tools, and "data path" interfaces which will be used by unprivileged domains. Creating and destroying an IO space is an administrative interface; mapping a machine page is a data path operation.

Different hardware IOMMUs have different characteristics, such as different degrees of device isolation. They might support no isolation (single global IO address space for all devices in the system), isolation between different

busses (IO address space per PCI bus), or isolation on the PCI Bus/Device/Function (BDF) level (i.e., a separate IO address space for each logical PCI device function). The IO space creation interface should expose the level of isolation that the underlying hardware is capable of, and should support any of the above isolation schemes. Exposing a finer grained isolation than the hardware is capable of could lead software to a false sense of security, and exposing a coarser grained isolation would not be able to fully utilize the capabilities of the hardware.

Another related question is whether several devices should be able to share the same IO address space, even if the hardware is capable of isolating between them. Let us consider a fully virtualized operating system that is not IOMMU aware and has several devices dedicated to it. Since the OS is not capable of utilizing isolation between these devices and each IO space consumes a small, yet non-negligible amount of memory for its translation tables, there is no point in giving each device a separate IO address space. For cases like this, it would be beneficial to share the same IO address space among all devices dedicated to a given operating system.

We have established that it may be beneficial for multiple devices to share the same IO address space. Is it likewise beneficial for multiple consumers (domains) to share the same IO address space? To answer this question, let us consider a smart IO adapter such as an Infiniband NIC. An IB NIC handles its own translation needs and supports many more concurrent consumers than PCI allows. PCI dedicates 3 bits for different "functions" on the same device (8 functions in total) whereas IB supports 24 bits of different consumers (millions of consumers). To support such "virtualization friendly" adapters, one could run with translation disabled in the IOMMU, or create a single IO space and let multiple consumers (domains) access it.

Since some hardware is only capable of having a shared IO space between multiple non-cooperating devices, it is beneficial to be able to create several logical IO spaces, each of which is a window into a single large "physical IO space". Each device gets its own window into the shared address space. This model only provides “statistical isolation”. A driver programming a device may guess another device’s window and where it has entries mapped, and if it guesses correctly, it could DMA there. However, the probability of its guessing correctly can be made fairly small. This mode of operation is not recommended, but if it’s the only mode the hardware supports...

Compared to creation of an IO space, mapping and unmapping entries in it is straightforward. Establishing a mapping requires the following parameters:

- A consumer needs to specify which IO space it wants to establish a mapping in. Alternatives for for identifying IO spaces are either an opaque, per-domain “IO space handle”" or the BDF that this IO space translates for.

- The IO address in the IO address space to establish a mapping at. The main advantage of letting the domain pick the IO address it that it has control over how IOMMU mappings are allocated, enabling it to optimize their allocation based on its specific usage scenarios. However, in the case of shared IO spaces, the IO address the device requests may not be available or may need to be modified. A reasonable compromise is to make the IO address a “hint” which the hypervisor is free to accept or reject.

- The access permissions for the given mapping in the IO address space. At a minimum, any of "none", "read only", "write only" or "read write" should be supported.
- The size of the mapping. It may be specified in bytes for convenience and to easily support different page sizes in the IOMMU, but ultimately the exact size of the mapping will depend on the specific page sizes the IOMMU supports.

To reduce the number of required hypercalls, the interface should support multiple mappings in a single hypervisor call (i.e., a "scatter gather list" of mappings).

Tearing down a mapping requires the following parameters:

- The IO space this mapping is in.
- The mapping, as specified by an IO address in the IO space.
- The size of the mapping.

Naturally, the hypervisor needs to validate that the passed parameters are correct. For example, it needs to validate that the mapping actually belongs to the domain requesting to unmap it, if the IO space is shared.

Last but not least, there are a number of miscellaneous issues that should be taken into account when designing and implementing IOMMU support. Since our implementation is targeting the open source Xen hypervisor, some considerations may be specific to a Xen or Linux implementation.

First and foremost, Linux and Xen already include a number of mechanisms that either emulate or complement hardware IOMMU functionality. These include SWIOTLB, grant tables, and the PCI frontend / backend drivers. Any IOMMU implementation should "play nicely" and integrate with these existing mechanisms, both on the design level (i.e., provide hardware acceleration for grant tables) and on the implementation level (i.e., do not duplicate common code).

One specific issue that must be addressed stems from Xen's use of page flipping. Pages that have been mapped into the IOMMU must be pinned as long as they are resident in the IOMMU's table. Additionally, any pages that are involved in IO may not be relinquished by a domain (e.g., by use of the balloon driver).

Devices and domains may be added or removed at arbitrary points in time. The IOMMU support should handle "garbage collection" of IO spaces and pages mapped in IO when the domain or domains that own them die or the device they map is removed. Likewise, hotplugging of new devices should also be handled.

5 Calgary IOMMU Design and Implementation

We have designed and implemented IOMMU support for the Calgary IOMMU found in high-end IBM xSeries servers. We developed it first on bare metal Linux, and then used the bare metal implementation as a stepping-stone to a "virtualization enabled" proof of concept implementation in Xen. This section describes both implementations. It should be noted that Calgary is an isolation capable IOMMU, and thus provides isolation between devices residing on different PCI Host Bridges. This capability is directly beneficial in Linux even without a hypervisor. It could be used for example to isolate a device in its own IO space while developing a driver for it, thus preventing DMA

related errors from randomly corrupting memory or taking down the machine.

5.1 x86-64 Linux Calgary support

The Linux implementation is included at the time of this writing in 2.6.16-mm1. It is composed of several parts: initialization and detection code, IOMMU specific code to map and unmap entries, and a DMA API implementation.

The bring-up code is done in two stages, detection and initialization. This is due to the somewhat convoluted way the x86-64 arch-specific code detects and initializes IOMMUs. In the first stage, we detect whether the machine has the Calgary chipset. If it does, we mark that we found a Calgary IOMMU. We also allocate large contiguous areas of memory for each PCI Host Bridge's translation table. Each translation table consists of a number of entries that total the addressable range given to the device (in page size increments). This stage uses the bootmem allocator and happens before the PCI subsystem is initialized. In the second stage, we map Calgary's internal control registers and enable translation on each PHB.

The IOMMU requires hardware specific code to map and unmap DMA entries. This part of the code implements a simple TCE allocator to "carve up" each translation table to different callers, and includes code to create TCEs (Translation Control Entries) in the format that the IOMMU understands and writes them into the translation table.

Linux has a DMA API interface to abstract the details of exactly how a driver gets a DMA'able address. We implemented the DMA API for Calgary, which allows generic DMA mapping calls to be translated to Calgary specific DMA calls. This existing infrastructure enabled the Calgary Linux code to be more easily hooked into Linux without many non-Calgary specific changes.

The Calgary code keeps a list of the used pages in the translation table via a bitmap. When a driver make a DMA API call to allocate a DMA address, the code searches the bitmap for an opening large enough to satisfy the DMA allocation request. If it finds enough space to satisfy the request, it updates the TCEs in the translation table in main memory to let the DMA through. The offset of those TCEs within the translation table is then returned to the device driver as the DMA address to use.

5.2 Xen Calgary support

Prior to this work, Xen did not have any support for isolation capable IOMMUs. As explained in previous sections, Xen does have software mechanisms (such as SWIOTLB and grant tables) that emulate IOMMU related functionality, but does not have any hardware IOMMU support, and specifically does not have any isolation capable hardware IOMMU support.

We added proof of concept IOMMU support to Xen. The IOMMU support is composed of a thin "general IOMMU" layer, and hardware IOMMU specific implementations. At the moment, the only implementation is for the Calgary chipset, based on the bare metal Linux Calgary support. As upcoming IOMMUs become available, we expect more hardware IOMMU implementations to show up.

It should be noted that the current implementation is proof of concept and is subject to change as IOMMU support evolves. In theory it targets numerous IOMMUs, each with distinct capabilities, but in practice it has only been implemented for the single isolation capable IOMMU that is currently available. We an-

ticipate that by the time you read this, the interface will have changed to better accommodate other IOMMUs.

The IOMMU layer receives the IOMMU related hypercalls (both the "management" hcalls from dom0 and the IOMMU map/unmap hcalls from other domains) and forwards them to the IOMMU specific layer. The following hcalls are defined:

- `iommu_create_io_space` - this call is used by the management domain (dom0) to create a new IO space that is attached to specific PCI BDF values. If the IOMMU supports only bus level isolation, the device and function values are ignored.
- `iommu_destroy_io_space` - this call is used to destroy an IO space, as identified by a BDF value.

Once an IO space exists, a domain can ask to map and unmap translation entries in its IOMMU using the following calls:

- `u64 do_iommu_map(u64 ioaddr, u64 mfn, u32 access, u32 bdf, u32 size);`
- `int do_iommu_unmap(u64 ioaddr, u32 bdf, u32 size);`

When mapping an entry, the domain passes the following parameters:

- ioaddr - The address in the IO space that the domain would like to establish a mapping at. This is a hint; the hypervisor is free to use it or ignore it and return a different IO address.
- mfn - The machine frame number that this entry should map. In the current Xen code base, a domain running on x86-64 and doing DMA is aware of the physical/machine translation, and thus there is no problem with passing the MFN. In future implementations this API will probably change to pass the domain's PFN instead.
- access - This specifies the Read/Write permission of the entry ("read" here refers to what the device can do - whether it can only read from memory or write to it as well).
- bdf - The PCI Bus/Device/Function of the IO space that we want to map this in. This parameter might be changed in later revisions to an opaque IO space identifier.
- size - How large is this entry? The current implementation only supports a single IOMMU page size of 4KB, but we anticipate that future IOMMUs will support large page sizes.

The return value of this function is the IO address where the entry has been mapped.

When unmapping an entry, the domain passes the BDF, the IO address that was returned and the size of the entry to be unmapped. The hypervisor validates the parameters, and if they validate correctly, unmaps the entry.

An isolation capable IOMMU is likely to either have a separate translation table for different devices, or have a single, shared translation table where each entry in the table is valid for specific BDF values. Our scheme supports both usage models. The generic IOMMU layer finds the right translation table to use based on the BDF, and then calls the hardware IOMMU specific layer to map or unmap an entry in it. In the case of one domain owning an IO space, the domain can use its own allocator and the

hypervisor will always use the IO addresses the domain wishes to use. In the case of a shared IO space, the hypervisor will be the one controlling IO address allocation. In this case IO address allocation could be done in cooperation with the domains, for example by adding a per domain offset to the IO addresses the domains ask for — in effect giving each domain its own window into the IO space.

6 Roadmap and Future Work

Our current implementation utilizes the IOMMU to run dom0 with isolation enabled. Since dom0 is privileged and may access all of memory anyway, this is useful mainly as a proof of concept for running a domain with IOMMU isolation enabled. Our next immediate step is to run a different, non privileged and non trusted "direct hardware access domain" with direct access to a device and with isolation enabled in the IOMMU.

Once we've done that, we plan to continue in several directions simultaneously. We intend to integrate the Calgary IOMMU support with the existing software mechanisms such as SWIOTLB and grant tables, both on the interface level and the implementation (e.g., sharing for code related to pinning of pages involved in ongoing DMAs). For configuration, we are looking to integrate with the PCI frontend and backend drivers, and their configuration mechanisms.

We are planning to add support for more IOMMUs as hardware becomes available. In particular, we look forward to supporting Intel and AMD's upcoming isolation capable IOMMUs.

Longer term, we see many exciting possibilities. For example, we would like to investigate support for other types of translation schemes used by some devices (e.g. those used by Infiniband adapters).

We have started looking at tuning the IOMMU for different performance/reliability/security scenarios, but do not have any results yet. Most current-day machines and operating systems run without any isolation, which in theory should give the best performance (least overhead on the DMA path). However, IOMMUs make it possible to perform scatter-gather coalescing and bounce buffer avoidance, which could lead to increased overall throughput.

When enabling isolation in the IOMMU, one could enable it selectively for "untrusted" devices, or for all devices in the system. There are many trade-offs that can be made when enabling isolation: one example is static versus dynamic mappings, that is, mapping the entire OS's memory into the IOMMU up front when it is created (no need to make map and unmap hypercalls) versus only mapping those pages that are involved in DMA activity. When using dynamic mappings, what is the right mapping allocation strategy? Since every IOMMU implements a cache of IO mappings (an IOTLB), we anticipate that the IO mapping allocation strategy will have a direct impact on overall system performance.

7 Conclusion: Key Research and Development Challenges

We implemented IOMMU support on x86-64 for Linux and have proof of concept IOMMU support running under Xen. We have shown that it is possible to run virtualized and non-virtualized operating systems on x86-64 with IOMMU isolation. Other than the usual woes associated with bringing up a piece of hardware for the first time, there are also interesting research and development challenges for IOMMU support.

One question is simply how can we build better, more efficient, IOMMUs that are easier to use in a virtualized environment? The upcoming IOMMUs from IBM, Intel and AMD have unique capabilities that have not been explored so far. How can we best utilize them and what additional capabilities should future IOMMUs have?

Another open question is whether we can use the indirection IOMMUs provide for DMA activity to migrate devices that are being accessed directly by a domain, without going through an indirect software layer such as the backend driver. Live virtual machine migration ("live" refers to migrating a domain while it continues to run) is one of Xen's strong points [9], but at the moment it is mutually incompatible with direct device access. Can IOMMUs mitigate this limitation?

Another set of open question relate to the on-going convergence between IOMMUs and CPU MMUs. What is the right allocation strategy for IO mappings? How to efficiently support large pages in the IOMMU? Does the fact that some IOMMUs share the CPU's page table format (e.g., AMD's upcoming IOMMU) change any fundamental assumptions?

What is the right way to support fully virtualized operating systems, both those that are IOMMU aware, and those that are not?

We continue to develop Linux and Xen's IOMMU support and investigate these questions. Hopefully, the answers will be forthcoming by the time you read this.

References

[1] *AMD I/O Virtualization Technology (IOMMU)* Specification, 2006, `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf`.

[2] *Intel Virtualization Technology for Directed I/O Architecture Specification*, 2006, `ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf`.

[3] *IA-64 Linux Kernel: Design and Implementation*, by David Mosberger and Stephane Eranian, Prentice Hall PTR, 2002, ISBN 0130610143.

[4] *Software Optimization Guide for the AMD64 Processors*, 2005, `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF`.

[5] *Xen and the Art of Virtualization*, by B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, in Proceedings of the 19th ASM Symposium on Operating Systems Principles (SOSP), 2003.

[6] *Xen 3.0 and the Art of Virtualization*, by I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, in Proceedings of the 2005 Ottawa Linux Symposium (OLS), 2005.

[7] *Documentation/DMA-API.txt*.

[8] *Documentation/DMA-mapping.txt*.

[9] *Live Migration of Virtual Machines*, by C. Clark, K. Fraser, S. Hand, J. G. Hanseny, E. July, C. Limpach, I. Pratt, A. Warfield, in Proceedings of the 2nd Symposium on Networked Systems Design and Implementation, 2005.

[10] *Safe Hardware Access with the Xen Virtual Machine Monitor*, by K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, M. Williamson, in Proceedings of the OASIS ASPLOS 2004 workshop, 2004.

[11] *PCI Special Interest Group* `http://www.pcisig.com/home`.