Using IOMMUs for Virtualization in Linux and Xen

Muli Ben-Yehuda, Jon Mason, Orran Krieger, Jimi Xenidis, Leendert Van Doorn, Asit Mallick, Jun Nakajima, Elsie Wahlig

muli@il.ibm.com

Table of Contents

Introducing IOMMUs

An I/O Memory Management Unit (IOMMU) is a hardware component that provides two main functions: **IO Translation** and **Device Isolation.**

- The IOMMU translates memory addresses presented by devices from “IO space” to “machine space” to allow a particular device to access physical memory potentially out of its range. It does this by providing an “in range” address to the device and translating the “in range” address to the physical memory address on the fly.
- The same translation function, when coupled with access permissions (“who can access this memory?”) can limit the ability of devices to access specific regions of memory.

Why do we need an IOMMU?

- Pros
  - 32-bit DMA capable, non-DAC, devices can access physical memory addresses higher than 4-GB.
  - IOMMUs can be programmed so that the memory region appears to be contiguous to the device on the bus (scatter/gather coalescing).
  - Device Isolation and other RAS features.
- Cons
  - TANSTAAFL, “there ain’t no such thing as a free lunch.” Remapping adds a performance hit to the transfer, albeit one which can be mitigated by an IOTLB.

The Main Advantage — Isolation

- For device isolation, an IO translation needs to be available to a given device, but not to some other device.
- We also need to restrict which domain can program which device — this is done by the hypervisor by controlling which domain can program which adapter.

Unfortunately, only one IOMMU available today on Intel and AMD based servers can do isolation.

AMD GART

- AMD Graphical Aperture Remapping Table (GART) provides a basic, translation only, IOMMU.
- Implemented in the on-chip memory controller
- Physical memory window and list of pages to be translated.
- Addresses outside the window are not translated.
- Fully supported in Linux; Xen support posted by not in the main tree.

AMD IOV Technology and Intel VT-d

- Provides translation and isolation.
- 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 its cache (IOTLB) and (if necessary) the I/O page tables for that device, based on the device's Bus/Dev/Func.
- Can be arranged in a topology of IOMMUs.
- IO page tables maintained in system memory by host software; with AMD's implementation, the page table format is compatible with the MMUs page table format.

swiotlb

- Linux includes **swiotlb** which is a software implementation of the translation function of an IOMMU. Also known as “bounce buffers”.
- Linux always uses swiotlb on IA64 machines, which have no hardware IOMMU, and can use it on x86-64 when told to do so or when the machine has too much memory and not enough IOMMU.
- As of 3.0.0, Xen always uses swiotlb in dom0, since swiotlb provides machine contiguous chunks of memory (required for DMA) unlike the rest of the kernel memory allocation APIs when running under Xen.
- Using swiotlb (or any other IOMMU) is completely transparent to the drivers - everything is implemented in the architecture’s DMA mapping API implementation.

Calgary TCEs

- Calgary's Translation Control Entries (TCEs) provide functionality to translate and isolate.
- Calgary provides a unique I/O address space up to 4-GB in size to all devices behind each PCI Host Bridge (PHB)
- Calgary uses the DMA address as an index into its IOTLB. If a translation is not found in the IOTLB, the address is used as an index a system controlled translation table in memory.
- Calgary is found in some of IBM's System P and System X servers.

Calgary TCE format

```
#define TCE_ENTRY_SIZE   8   /* in bytes */

#define TCE_READ_SHIFT   0
#define TCE_WRITE_SHIFT  1
#define TCE_HUBID_SHIFT  2   /* unused */
#define TCE_RSVD_SHIFT   8   /* unused */
#define TCE_RPN_SHIFT    12
#define TCE_UNUSED_SHIFT 48  /* unused */
```

Linux

- Linux has a standard API for dealing with DMA memory which all well written drivers are already using, the DMA-API.
- First we cleaned up the x86-64 DMA-API implementation to support more than nommu, swiotlb and gart cleanly — the dma-ops patch.
- Then we did Calgary bringup on bare metal Linux.
- And implemented the DMA-API for Calgary on the server formerly known as “xSeries x366.”
- Despite the hardware having never been validated, it actually works.
- We had to work around a few oddities creatively - cue funny story about TCE shoot-downs.

Linux continued

- Calgary merged in 2.6.18-rc1.
- This provides an isolation capable IOMMU on Intel/AMD based servers - get your DMA handling wrong and the DMA will be stopped by the IOMMU with an informative message rather than corrupting memory!
- Some open issues: direct userspace access (i.e., X), graceful handling and recovery of driver errors, better integration with swiotlb, NUMA support, etc, etc...

dmesg in action

```
PCI-DMA: Using Calgary IOMMU
Calgary: enabling translation on PHB 0
Calgary: errant DMAs will now be prevented on this bus.
Calgary: enabling translation on PHB 1
Calgary: errant DMAs will now be prevented on this bus.
Calgary: enabling translation on PHB 2
Calgary: errant DMAs will now be prevented on this bus.
```

Direct Hardware Access

One of the main selling points of virtualization is machine consolidation. So let’s assume for a second that you put your database virtual machine and your web server virtual machine on the same physical machine. Your database needs fast disk access; your web server, fast network access.

Xen supports the ability to allocate different **physical** devices to different domains (multiple “driver domains”). However, due to architectural limitations of most PC hardware, this cannot be done securely. In effect, any domain that has direct hardware access has to be considered “trusted”.

The Problem with Direct Access

The reason why is that all IO is done in physical addresses. Consider the following case:

- domain A is mapped in 0-2 GB of physical memory
- domain B is mapped in 2-4 GB of physical memory
- domain A has direct access to a PCI NIC
- domain A programs the NIC to DMA in the **2-4GB** physical memory range, overwriting domain B's memory. Ooops!

The solution - an IOMMU.

Xen

- Main goal: using Calgary to provide direct access to devices from multiple driver domains.
- Almost there — dom0 running with Calgary enabled.
- Working on getting another driver domain running with Calgary enabled as well.
- dom0 detects Calgary in the machine — notifies hypervisor which initializes Calgary support.
- New privileged hypercalls: iommu detected, create and destroy IO space. IO spaces are identified by BDF or parts of BDF.
- New hypercalls: map and unmap translation entry in IO space.

Roadmap

- Integration with Xen’s grant tables and PCI frontend and backend drivers.
- Support for Intel and AMD’s upcoming IOMMUs.
- Migration?
- Fully virtualized OS’s with direct device access?
- Performance...
- How do we build better IOMMUs?

Status and Availability

- Linux code merged into mainline: 2.6.18-rc1 should have it all.
- Xen trees available on xenbits:
  - `http://xenbits.xensource.com/ext/xen-iommu.hg`
  - `http://xenbits.xensource.com/ext/linux-iommu.hg`

Questions?