The Virtual Contiguous Memory Manager

Zach Pfeffer
Qualcomm Innovation Center (QuIC)
`zpfeffer@quicinc.com`

Abstract

An input/output memory management unit (IOMMU) maps device addresses to physical addresses. It also insulates the system from spurious or malicious device addresses and allows fine-grained mapping attribute control. The Linux kernel core does not contain a generic API to handle IOMMU mapped memory; device driver writers must implement device specific code to interoperate with the Linux kernel core. As the number of IOMMUs increases, coordinating the many address spaces mapped by all discrete IOMMUs becomes difficult without in-kernel support.

To address this complexity the Qualcomm Innovation Center (QuIC) created the Virtual Contiguous Memory Manager (VCMM) API. The VCMM API enables device independent IOMMU control, VMM interoperation and non-IOMMU enabled device interoperation by treating devices with or without IOMMUs and all CPUs with or without MMUs, their mapping contexts and their mappings using common abstractions. Physical hardware is given a generic device type and mapping contexts are abstracted into Virtual Contiguous Memory (VCM) regions. Users "reserve" memory from VCMs and "back" their reservations with physical memory. We have implemented the VCMM to manage the IOMMUs of an upcoming ARM based SoC. The implementation will be posted to the Code Aurora Foundation's site.

1 Motivation and Opportunities

Driver writers who control devices with IOMMUs must contend with device control and memory management. Driver writers have a large device driver API that they can leverage to control their devices, but they are lacking a unified API to help them program mappings into IOMMUs and share those mappings with other devices and CPUs in the system.

Sharing is complicated by Linux?s CPU centric VMM. The CPU centric model generally makes sense because average hardware only contains a MMU for the CPU and possibly a graphics MMU. If every device in the system has one or more MMUs, a CPU centric memory management (MM) programming model breaks down.

The VCMM was built to allow abstract device programming and mapping interoperation.

2 VCMM Abstractions

Abstracting IOMMU programming into a common API has already begun in the Linux kernel. It was built to abstract the difference between AMD's and Intel's IOMMUs to support x86 virtualization on both platforms. The interface is listed in kernel/include/linux/iommu.h. It contains interfaces for mapping and unmapping as well as 'domain management.' This interface has not gained widespread use outside the x86; PA-RISC, Alpha and SPARC architectures and ARM and PowerPC platforms all use their own mapping modules to control their IOMMUs. The VCMM contains an IOMMU programming layer, but since its abstraction supports map management independent of device control, the layer is not used directly. This higher-level view enables a new kernel service, not just an IOMMU interoperation layer.

Looking at mapping from a system-wide perspective reveals a general graph problem. The VCMM's API is built to manage the general mapping graph. Each node that talks to memory, either through an MMU or directly (physically mapped) can be thought of as the device end of a mapping edge. The other edge is the physical memory (or intermediate virtual space) that is mapped.

In the direct mapped case the device is assigned a 'one-to-one' MMU. This scheme allows direct mapped devices to participate in general graph management.

The CPU nodes can also be brought under the same mapping abstraction with the use of a light overlay on the existing VMM. This light overlay allows VMM managed mappings to interoperate with the common API. The light overlay enables this without substantial modifications to the existing VMM.

In addition to CPU nodes that are running Linux (and the VMM), remote CPU nodes that may be running other operating systems can be brought into the general abstraction. Routing all memory management requests from a remote node through the central memory management framework enables new features like system-wide memory migration. This feature may only be feasible for large buffers that are managed outside of the fast-path, but having remote allocation in a system enables features that are impossible to build without it.

The fundamental objects that support these abstractions are:

- Virtual Contiguous Memory Regions
- Reservations
- Associated Virtual Contiguous Memory Regions
- Memory Targets
- Physical Memory Allocations

In a nut-shell, users allocate Virtual Contiguous Memory Regions and associate those regions with one or more devices by creating an Associated Virtual Contiguous Memory Region. Users then create Reservations from the Virtual Contiguous Memory Region. At this point no physical memory has been committed to the reservation. To associate physical memory with a reservation a Physical Memory Allocation is created and the Reservation is backed with this allocation.

3 Virtual Contiguous Memory Regions

A Virtual Contiguous Memory Region (VCM) abstracts the memory space a device 'sees.' The addresses of the region are only used by the devices which are associated with the region. This address space would normally be implemented as a device page-table.

A VCM is created and destroyed with three functions:

```
vcm_id = vcm_create(start_addr, len);
```

```
vcm_id = vcm_create_from_prebuilt(ext_
vcm_id);
```

```
vcm_free(vcm_id);
```

start_addr is an offset into the address space where allocations will start from. len is the length from start_addr of the VCM. Both functions generate a vcm_id which is an opaque instance of a VCM.

ext_vcm_id is used to pass a request to the VMM to generate a vcm_id. In the current implementation the call simply makes a note that the vcm_id is a VMM vcm_id for other interfaces usage. This 'muxing' is seen throughout the implementation.

vcm_create() and vcm_create_from_prebuilt() produce vcm_ids for virtually mapped devices (IOMMUs and CPUs). To create a one-to-one mapped VCM users pass the start_addr and len of the physical region. The VCMM matches this and records that the vcm_id is a one-to-one VCM.

The newly created vcm_id can be passed to any function that needs to operate on or with a virtual contiguous memory region. Its main attributes are a start_addr and a len as well as an internal setting that allows the implementation to mux between true virtual spaces, one-to-one mapped spaces and VMM managed spaces.

The current implementation uses the genalloc library to manage the VCM for IOMMU devices.

4 Reservations

A Reservation is a contiguous region allocated from a VCM. There is no physical memory associated with it.

A Reservation is created and destroyed with:

```
res_id = vcm_reserve(vcm_id, len, attr);
```

```
vcm_unreserve(res_id);
```

A vcm_id is a VCM created above. len is the length of the request. It can be up-to the length of the VCM region the reservation is being created from. attr are mapping attributes: read, write, execute, user, supervisor, secure, not-cached, write-back/write-allocate, write-back/no write-allocate, write-through. These attrs can be changed to match to any architecture.

The implementation calls gen_pool_alloc() for IOMMU devices, alloc_vm_area() for VMM areas and is a pass through for one-to-one mapped areas.

5 Associated Virtual Contiguous Memory Regions and Activation

An Associated Virtual Contiguous Memory Region (AVCM) is a mapping of a VCM to a device. The mapping can be active or inactive.

An AVCM is managed with:

```
avcm_id = vcm_assoc(vcm_id, dev_
id, attr);
```

```
vcm_deassoc(avcm_id);
```

```
vcm_activate(avcm_id);
```

```
vcm_deactivate(avcm_id);
```

A vcm_id is a VCM created above. dev_id is an opaque device handle that's passed down to the device driver the VCMM muxes in to handle a request. attr are association attributes: split, use-high or use-low. split controls which address hit a 'high-address' page-table and which addresses hit a "low-address" page-table. For instance, all addresses whose most-significant-bit is one would use the "high-address" page-table, any other register would use the 'low address' page-table. One vcm_id can be associated with many devices and many vcm_ids can be associated with one device.

An AVCM is only a link. To program and deprogram a device with a VCM the user calls vcm_activate() and vcm_deactivate().For IOMMU devices, activating a mapping programs the base address of a page-table into an IOMMU. For VMM and one-to-one based devices, mappings are active immediately; the API does require an activation call for them for internal reference counting.

6 Memory Targets

A Memory Target is a platform independent way of specifying a physical pool; it abstracts a pool of physical memory. The physical memory pool may be physically discontinuous, need to be allocated from in a unique way or have other user-defined attributes.

7 Physical Memory Allocation and Reservation Backing

Physical memory is allocated as a separate step from reserving memory. This allows multiple reservations to back the same physical memory. A Physical Memory Allocation is managed using the following functions:

```
physmem_id = vcm_phys_
alloc(memtype, len, attr);
```

```
vcm_phys_free(physmem_id);
```

```
vcm_back(res_id, physmem_id);
```

```
vcm_unback(res_id);
```

attr can include an alignment request, a specification to map memory using various block sizes and/or to use physically contiguous memory. memtype is one of the memory types listed in Memory Targets.

The current implementation manages two pools of memory. One pool is a contiguous block of memory and the other is a set of contiguous block pools. In the current implementation the blocks pools contain 4K, 64K and 1M blocks. The physical allocator does not try to split blocks from the contiguous block pools to satisfy requests.

The use of 4K, 64K and 1M blocks solves a problem with some IOMMU hardware. IOMMUs are placed in front of multimedia engines to provide a contiguous address space to the device. Multimedia devices need large buffers and large buffers may map to a large number of physical blocks. IOMMUs tend to have small translation lookaside buffers (TLBs). The number of physical blocks that map a given range needs to be small or else the IOMMU will continually fetch new translations during a typical streamed multimedia flow since the TLB is small. By using a 1 MB mapping (or 64K mapping) instead of a 4K mapping the number of misses can be minimized, allowing the multimedia block to meet its performance goals.

8 Low Level Control

It is necessary to access attributes of the abstractions. The API contains many functions but the two that are typically used are:

```
devaddr = vcm_get_dev_addr(res_id);
```

```
vcm_hook(dev_id, user_
handler, void *data);
```

The first function, vcm_get_dev_addr() returns a device address given a reservation. This device address is a virtual IOMMU address for reservations on IOMMU VCMs, a virtual VMM address for reservations on VMM VCMs and a 'virtual' (physical) address for one-to-one devices.

The second function, vcm_hook allows a caller in the kernel to register a user_handler. The handler is passed the data during a fault. The user can return 1 to indicate that the underlying driver should handle the fault and retry the transaction or can return 0 to halt the transaction. If the user doesn't register a handler the low-level driver will print a warning and terminate the transaction.

9 A Detailed Walk Through

The following call sequence walks through a typical allocation sequence. In the first stage the memory for a device is reserved and backed. This occurs without mapping the memory into a VMM VCM region. The second stage maps the first VCM region into a VMM VCM region so the kernel can read or write it. The second stage is not necessary if the VMM does not need to read or modify the contents of the original mapping. Figure 1 shows the mappings schematically.

Stage 1: Map and Allocate Memory for a Device

The call sequence starts by creating a VCM region:

```
vcm_id = vcm_create(start_addr, len);
```

The next call associates a VCM region with a device:

```
avcm_id = vcm_assoc(vcm_id, dev_
id, attr);
```

To activate the association users call vcm_activate() on the avcm_id from the associate call. This programs the underlining device with the mappings.

```
vcm_activate(avcm_id);
```

Once a VCM region is created and associated it can be reserved from.

```
res_id = vcm_reserve(vcm_id, res_
len, res_attr);
```

A user allocates physical memory:

```
physmem_id = vcm_phys_
alloc(memtype, len, phys_attr);
```

To back the reservation with the physical memory allocation the user calls:

```
vcm_back(res_id, physmem_id);
```

Stage 2: Map the Device's Memory into the VMM's VCM region

If the VMM needs to read and/or write the region that was just created the following calls are made.

The first call creates a prebuilt VCM:

```
vcm_vmm_id = vcm_from_prebuit(ext_vcm_
id);
```

The prebuilt VCM is associated with the CPU device and activated:

```
avcm_vmm_id = vcm_assoc(vcm_vmm_id, dev_
cpu_id, attr);
```

```
vcm_activate(avcm_vmm_id);
```

A reservation is made on the VMM VCM:

```
res_vmm_id = vcm_reserve(vcm_vmm_
id, res_len, attr);
```

Once the topology has been set up a vcm_back() allows the VMM to read the memory using the physmem_id generated in stage 1:

```
vcm_back(res_vmm_id, physmem_id);
```

10 Mapping IOMMU, one-to-one and VMM Reservations

Figure 3 demonstrates mapping IOMMU, one-to-one and VMM reservations to the same physical memory. It shows the use of phys_addr and phys_size to create a contiguous VCM for one-to-one mapped devices. Figure 2 shows the mappings schematically.

11 Summary

The VCMM is an attempt to abstract attributes of three distinct classes of mappings into one API. The VCMM allows users to reason about mappings as first class objects. It also allows memory mappings to flow from the traditional 4K mappings prevalent on systems today to more efficient block sizes. Finally, it allows users to manage mapping interoperation without becoming VMM experts. These features will allow future systems with many MMU mapped devices to interoperate simply and therefore correctly.

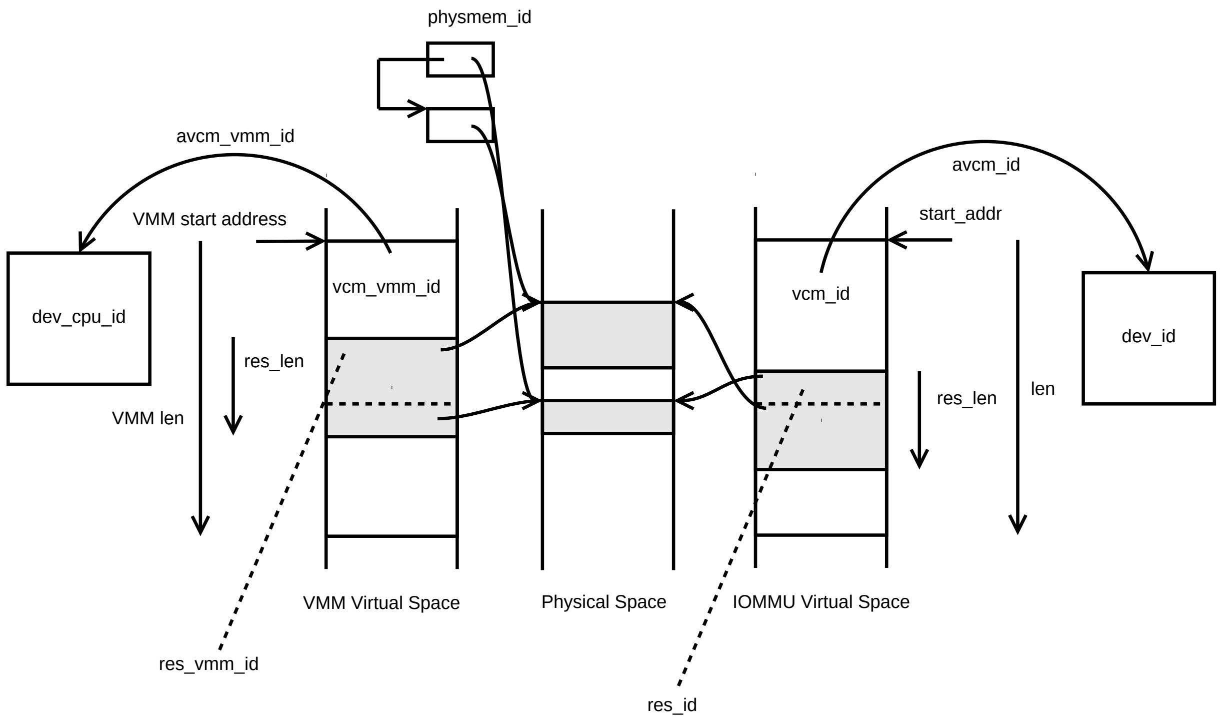

Figure 1: Walk Through

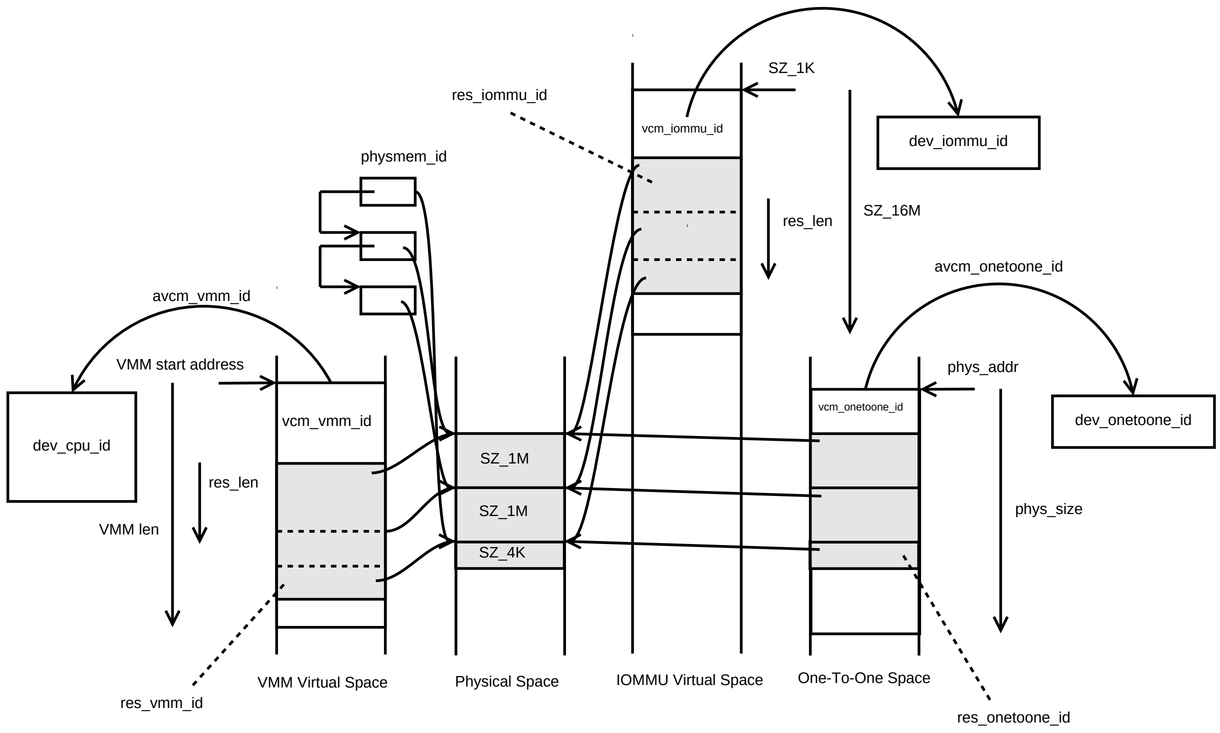

Figure 2: Mapping IOMMU, One-to-One and VMM Reservations

```
physmem_id = vcm_phys_alloc(memtype, SZ_2MB + SZ_4K, CONTIGUOUS);}
vcm_iommu_id = vcm_create(SZ_1K, SZ_16M);}
vcm_onetoone_id = vcm_create(phys_addr, phys_size);}
vcm_vmm_id = vcm_from_prebuit(ext_vcm_id);}

avcm_iommu_id = vcm_assoc(vcm_iommu_id, dev_iommu_id, attr0);}
avcm_onetoone_id = vcm_assoc(vcm_onetoone_id, dev_onetoone_id, attr1);}
avcm_vmm_id = vcm_assoc(vcm_vmm_id, dev_cpu_id, attr2);}

vcm_activate(avcm_iommu_id);}
vcm_activate(avcm_onetoone_id);}
vcm_activate(avcm_vmm_id);}

res_iommu_id = vcm_reserve(vcm_iommu_id, SZ_2MB + SZ_4K, attr);}
res_onetoone_id = vcm_reserve(vcm_onetoone_id, SZ_2MB + SZ_4K, attr);}
res_vmm_id = vcm_reserve(vcm_vmm_id, SZ_2MB + SZ_4K, attr);}

vcm_back(res_iommu_id, physmem_id);}
vcm_back(res_onetoone_id, physmem_id);}
vcm_back(res_vmm_id, physmem_id);}
```

Figure 3: Mapping IOMMU, One-to-One and VMM Reservations Example

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada