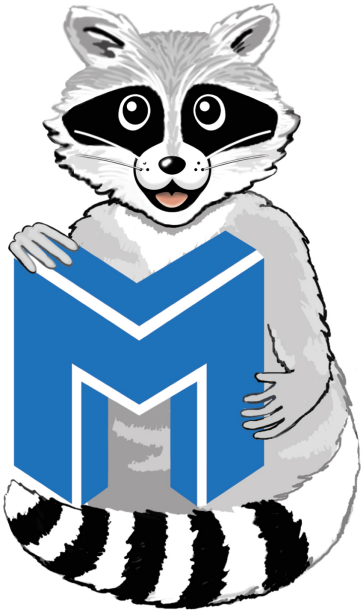
IOMMU driver for MINIX 3



Author:

Adriana Szekeres

Supervisors:

Erik van der Kouwe

dr. Andrew S. Tanenbaum

May, 2011

Table of Contents

Introduction

The Input/Output Memory Management Unit (IOMMU) is a hardware device (chipset feature) designed to translate the I/O virtual addresses, i.e. the addresses used by devices to access the main memory, to physical addresses. Its main function is to protect the main memory against faulty devices/drivers, by controlling their DMA operations.

While a traditional MMU is used to translate to physical addresses the virtual addresses issued by the CPU, the IOMMU sits between devices and the main memory, intercepting and translating to physical addresses the I/O virtual addresses issued by the devices, (see Fig. 1.1).

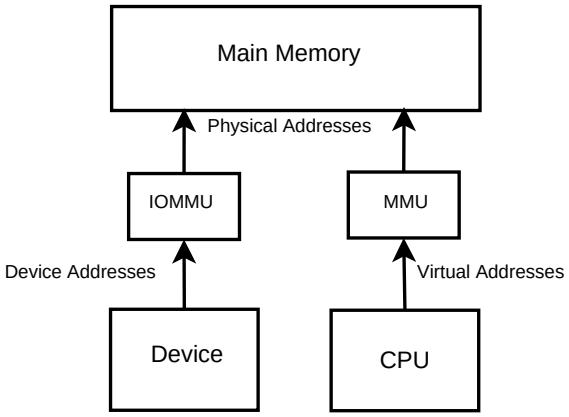


Fig. 1.1: Comparison between the IOMMU and the traditional MMU

The IOMMU has been recently added to AMD and Intel chipsets. The AMD-Vi is the AMD implementation of the IOMMU. Its specification is published in [amd09]. Intel has also published a specification for their IOMMU technology, called Virtualization Technology for Directed I/O, abbreviated VT-d [int08].

This project aims to build the IOMMU driver for MINIX, for an AMD-Vi device.

Architecture

The general architecture of the MINIX 3 IOMMU driver has been designed to be compatible with the two IOMMU specifications (AMD and Intel). The main components of the IOMMU driver and their interaction with the other MINIX 3 servers/drivers are shown in Fig. 2.1. The direction of the arrow specifies which driver/server is the initiator of the message (using *sendrec* or *send* syscalls). For example, the IOMMU is the one initiating the requests to both PCI driver and ACPI driver, using the *sendrec* syscall.

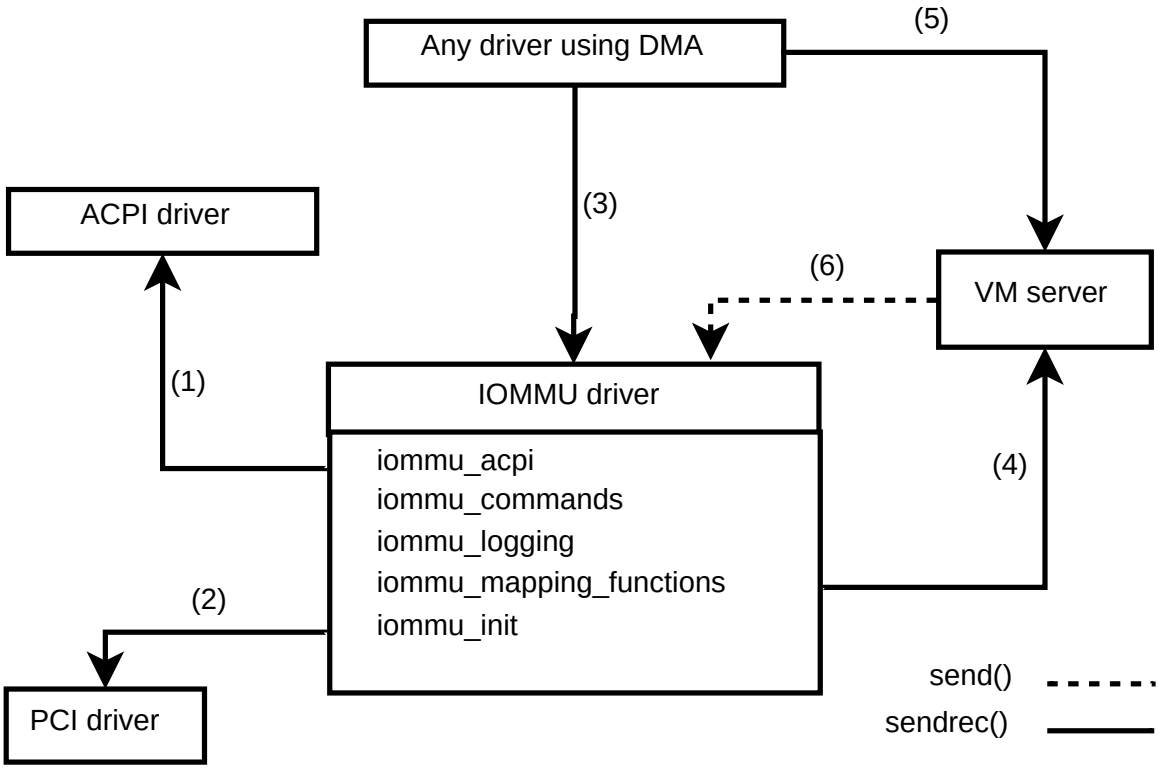


Fig. 2.1: The architecture for the testing system

The numbers assigned to the arrows in Fig. 2.1 refer to the following messages (defined in **common/include/minix/acpi.h** and **common/include/minix/com.h**) that are sent between the drivers/servers:

- (1)
  ACPI_REQ_GET_TABLE_SIZE
  ACPI_REQ_GET_TABLE

- (2)
  BUSC_PCI_BDF
- (3)
  IOMMU_MMAP
  IOMMU_MUNMAP
- (4)
  VM_IOMMU_REGISTER
- (5)
  VM_IOMMU_MMAP
  VM_IOMMU_MUNMAP
- (6)
  IOMMU_CLEAR_PT

The roles of the above mentioned messages are explained in detail in the next chapters of this document.

Configuration and status information for the IOMMU are mapped into PCI configuration space using a PCI capability block. Also, the same and additional information can be found in the ACPI tables. The IOMMU's registries can be accessed through memory mapped I/O (MMIO).

As it is shown in Fig. 2.1, the IOMMU driver consists of five components:

- iommu_acpi - extracts the necessary information from the ACPI tables
- iommu_commands - implements the commands supported by the IOMMU
- iommu_logging - implements the logging mechanism
- iommu_mapping_functions - implements and fills-in the structures used by the IOMMU to map the physical addresses that drivers are allowed to access.
- iommu_init - initializes the IOMMU

AMD's IOMMU driver Implementation

In this chapter we will present the implementation of each of the five components for the AMD-Vi IOMMU driver.

3.1 IOMMU_acpi

This component parses the ACPI IVRS table. The IVRS table contains information about all IOMMUs present in the system. The IVRS table is composed of a 48-byte long header, followed by IVHD and IVMD blocks. The header contains information such as: the length of the entire table, the Vendor ID of the utility that created the table, and the virtual and physical address sizes (which are common to all IOMMUs).

The IVHD blocks are used to describe the I/O topology of I/O devices and slots served by the IOMMUs. Basically, for each IOMMU in the system, there is an IVHD block that specifies which devices that IOMMU serves. Therefore, an IVHD must exist for each IOMMU in the system. Additionaly, the IVHD block contains important information about the IOMMU it describes, such as: DeviceID (BDF) of IOMMU (this will be used to access the PCI configuration space, as I will describe bellow), the offset in Capability space for control fields of IOMMU (the Capability space is also in the PCI configuration space, after the standardized header) and Base address of IOMMU control registers in MMIO space (this will be basically used to send commands to the IOMMU).

The IVRS table is described in more detail in IOMMU Architectural Specification [amd09].

Impelementation details:

First, the IOMMU_acpi will interrogate the ACPI driver for the IVRS table and then it will parse it and store it into some structures (some of the structures have been copied from actbl2.h). Because in MINIX each driver is a separate process, with its own address space, we cannot access directly functions from the ACPI driver. Therefore, the ACPI driver has been extended, as we will describe in Chapter 5.

The implementation can be found in the following files:

- **iommu_acpi_amd.h** This file contains the structures that describe the ACPI IVRS table: the ACPI_TABLE_HEADER, which is common to all ACPI tables

(one of its most important fields is the length field, which specifies the length, in bytes, of the whole table); the ACPI_TABLE_IVRS, which is the header of the IVRS table; the ACPI_IVRS_HEADER, which is the header of the IVRS subtables, i. e. IVHD and IVMD; the ACPI_IVRS_HARDWARE, which is the structure describing an IVHD block; the ACPI_IVRS_DE_HEADER, which is the header of the device entries; the ACPI_IVRS_MEMORY, which describes an IVMD block; the iommu_container_t, which describes one IOMMU; etc. This file also contains three important global variables: iommu_s, which contains a list of all IOMMUs in the system; iommu_lookup, which maps the device to the IOMMU that is responsible for it and the domain_lookup which is a hash that maps a domain id to its corresponding structure.

This file also contains some masks which are used to extract specific parts from the table's fields. For example, the 32-bit long Info field from the structure ACPI_TABLE_IVRS is divided into 5 parts: bits [7:0] are reserved and must be 0, bits [14:8] represent the physical address size, bits [21:15] represent the virtual address size, bit 22 is the ATS address translation range reserved and bits [31:23] are also reserved and must be zero. So, for example, to extract the virtual address size, we use the mask (21 << 8 | 15) in which we encoded two numbers: the last bit of the mask, 21, and the first bit of the mask, 15, that specifies with how many bits the field (in our case, Info) must be shifted to the right. The following functions are used for field extraction:

```
#define LAST(m)                                          \
    (0xFFFFFFFF >> (31 - (m >> 8)))

#define FIRST(m)                                         \
    (0xFFFFFFFF << (m & 0xFF))

#define MASK(m)                                          \
    (FIRST(m) & LAST(m))

#define GET_FIELD(p, m)                                  \
    (((p) & MASK(m)) >> (m & 0xFF))
```

- **iommu_acpi_amd.c** This file contains the implementation of the ACPI IVRS table parsing. The system_call ACPI_REQ_GET_TABLE is used to copy the IVRS table to IOMMU driver's address space (see Modifications to MINIX, at the end of this document). After receiving the table, the IOMMU_acpi parses it and stores the references to all the IVHD subtables it finds (in function *iommu_acpi_parse_ivrs()*). During the IOMMU initialization, specifically after the device table structure is allocated, function *process_iommu_entries_amd()* is called to further process the IVHD subtable, with all its entries, and to fill-in the device table structure accordingly.

3.2 IOMMU_init

This component initializes the IOMMU structures and enables the IOMMUs. It allocates all the necessary memory for the IOMMU structures and sets them up at the device. Communication with the IOMMU is done through PCI configuration space and Memory Mapped I/O (MMIO). The PCI driver offers system calls to access the PCI configuration space (we need to know only the DeviceID, BDF - we find this in the IVHD block from the IVRS table). MMIO access is a normal memory access (we just need to know the address where the device is mapped into memory - we find this in the capability block in the PCI configuration space).

Impelementation details:

The implementation of this component can be found in the following files:

- **iommu_init_amd.h** This file only defines some sizes for the structures used by the IOMMU, such as: the maximum size of the device table, the size of the command and event logging buffers, etc.
- **iommu_init_amd.c** This file implements the functionality of this component. Basically, the main function is *iommu_init_all_amd()*, which detects all the IOMMUs from the PCI devices array and sets them up by allocating the necassary memory space for their structures and by writing the addresses of the allocated buffers into their MMIO space (in the device's registers).

3.3 IOMMU_commands

The IOMMU is controled by using a shared circular buffer in system memory. The IOMMU driver writes commands into the buffer and then notifies the IOMMU. The IOMMU then reads the commands and executes them asynchronously. To implement this mechanism, the IOMMU has the control registers mapped into the memory (MMIO space), e.g. the Command Buffer Base Address Register, Command Buffer Head Pointer Register, etc. The supported commands are:

COMPLETION_WAIT: it synchronizes the IOMMU driver with the IOMMU device. The COMPLETION_WAIT command does not finish until all older commands issued since a prior COMPLETION_WAIT have completely executed.

INVALIDATE_DEVTAB_ENTRY: When the IOMMU driver changes a device table entry, it must instruct the IOMMU to invalidate that DeviceID from its internal caches. The IOMMU is then forced to reload the device table entry before DMA from the device is allowed. The IOMMU may reload the device table entry any time after the invalidation has completed.

INVALIDATE_IOMMU_PAGES: this command instructs the IOMMU to invalidate a range of entries in its translation cache for the specified DomainID.

INVALIDATE_IOTLB_PAGES: this command is only present in IOMMU implementations that support remote IOTLB caching of translations. This command instructs the specified device to invalidate the given range of addresses in its IOTLB.

INVALIDATE_INTERRUPT_TABLE: this command instructs the IOMMU to invalidate all cached interrupt remapping table entries for the device.

However, not all of the above commands are used in the driver yet. For example, the driver does not yet support interrupt remapping. Consequently, the command INVALIDATE_INTERRUPT_TABLE is not yet used, although it is implemented by the IOMMU driver.

Impelementation details:

The implementation of this component can be found in the following files:

- **iommu_cmd_amd.h** This file defines the constants used to implement the commands for the IOMMU. Mainly, this file contains the constants used to work with the IOMMU Control Registers (specified in section 3.6.2 from [amd09]).
- **iommu_cmd_amd.c** The main function in this file is *iommu_cmd_amd()*, which accepts a command request, fills-in a command buffer, with the specified command format, and puts it into the circular Command Buffer.

3.4 IOMMU_logging

The IOMMU reports events to the IOMMU driver by means of another shared circular buffer in system memory. The IOMMU device writes event records into the buffer. The IOMMU driver increments the IOMMU's head pointer to indicate to the IOMMU device that it has consumed event log entries. If the IOMMU needs to report an error but finds that the event log is already full, it sets MMIO Offset 2020h[EventOverflow]. The IOMMU can be configured to signal an interrupt whenever the event log is written. However, for this it needs the OS to support MSI/MSI-X interrupts. MINIX currently doesn't support MSI/MSI-X. Therefore, for now, the IOMMU driver just prints the log events after each command request to the IOMMU device, even if no event happened. The following events can appear:

ILLEGAL_DEV_TABLE_ENTRY: when the IOMMU performs a lookup in the device table and encounters a device table entry that it does not support or that is formatted incorrectly.

IO_PAGE_FAULT: when the IOMMU performs a lookup in the page tables for a device and encounters an error condition.

DEV_TAB_HARDWARE_ERROR: if the IOMMU detects a hardware error (master abort, target abort, poisoned data, etc.) while accessing the device table.

PAGE_TAB_HARDWARE_ERROR: if the IOMMU detects a hardware error (master abort, target abort, poisoned data, etc.) while accessing the I/O page tables.

ILLEGAL_COMMAND_ERROR: if the IOMMU reads an illegal command (including an unsupported command code, or a command that incorrectly has reserved bits set).

COMMAND_HARDWARE_ERROR: if the IOMMU detects a hardware error (master abort, target abort, poisoned data, etc.) while accessing the command buffer.

IOTLB_INV_TIMEOUT: if the IOMMU sends an invalidation request to a device and does not receive a response before the invalidation timeout timer expires.

INVALID_DEVICE_REQUEST: if the IOMMU receives a request from a device that is not allowed to perform.

Impelementation details:

The implementation of this component can be found in the following files:

- **iommu_log_amd.h** This file defines the format for each log event (specified in section 3.4 from [amd09]).
- **iommu_log_amd.c** The main function in this file is *display_log_amd()*, which reads all the events from the Event Log Buffer, parses and then *pretty prints* them to the console.

3.5 IOMMU_mapping_functions

This component is responsible for generating the I/O virtual addresses that will be used by the devices to perform DMA transfers. The I/O virtual address is basically an index into a tree structure of page tables. The page tables are used by the IOMMU to check if the device has the permission to access the requested memory location.

The AMD's IOMMU page tables are designed to support a full 64-bit device virtual address space. They are a multi-level tree of 4K tables indexed by groups of 9 virtual address bits (determined by the level within the tree) to obtain 8-byte entries. Each

page table entry is either a *page directory entry* (PDE) pointing to a lower-level 4K page table, or a *page translation entry* (PTE) specifying a physical page address (frame). A page translation entry is a page table entry with the Next Level field set to 0h or 7h. A page directory entry is a page table entry with the Next Level field not equal to 0h or 7h. More information on AMD's IOMMU page tables can be found in section 3.2.3 from [amd09].

By default, the IOMMU driver maps the physical address of the DMA buffer to the page table index (IO virtual address) with the same value. This behaviour is preferred as it will allow the DMA transfers to be successful even if the IOMMU device (hardware) fails. However, the IOMMU can be instructed first to generate an IO virtual address and then map the physical address to that page table index. We believe that this second behavior will be useful to the Virtual Machine Manager (VMM).

In the following two subsections we will first describe the details of the IO virtual address generator and then the how the IOMMU driver manipulates the page tables.

3.5.1 Simple virtual address allocator

The virtual address allocator allocates a free range of virtual addresses from a given virtual address pool, which is basically a linked list of free address ranges.

Impelementation details:

The implementation of the allocator can be found in the following two files:

- **iommu_va_allocator.h** This file contains the structure defining the element of the linked list that contains the currently unallocated virtual address ranges:

```
typedef struct range {
 u32_t start;
 u32_t end;
 struct range *next;
} range_t;
```

  Each IOMMU initializes this structure with only one element, i.e. the range from 0 to the maximum number of pages supported by MINIX. As MINIX operates on 32 bits and the last 12 bits are for page offset, the maximum number of pages is 2^{20}. When the IOMMU needs to allocate a given number of pages, it passes this structure to the allocation functions, described next.

- **iommu_va_allocator.c** This file contains the (de)allocation functions. These functions basically work on the given *range* structure, passed as a parameter. The allocator supports the following functions:

```
int va_alloc(range_t **range, u32_t size, u32_t *va)
```

This function finds the first free range of pages that fit in the requested size. It accepts the following parameters:

range - the range on which the allocation will be performed (this parameter will be modified accordingly after the allocation)

size - size in number of pages of the address space that is needed

va - (output parameter) it contains the first page that has been allocated

```
int va_relloc(range_t **range, u32_t size, u32_t va)
```

This function also allocates a range of pages, but this time it knows the virtual address that it has to allocate (it doesn't find the first fit). This function is used when the IOMMU has to map the mappings received from the VM, as well as in the normal operation of the IOMMU (when the flag IOMMU_GEN_VA is not used, see 4). It uses the same parameters as the previous function, except for va, which is now an input parameter and represents the page at which the allocation must start.

```
int va_free(range_t **range, u32_t va, u32_t size)
```

This function frees a previous allocation. It accepts the same parameters as the previous two functions.

3.5.2 Page tables manipulation

This is the part that deals with the mapping of physical addresses into the IOMMU page tables. The (un)mapping algorithm is simple. We start with a virtual page address, that was received or not from the simple allocator. This virtual address represents a page index into the IOMMU page tables, which we will call *first index*. Let the *end index* be the sum of *start index* and the number of pages to be mapped. The first thing to do is see if *end index* exceeds the current level of the page tables, i.e. it is too big for the currently supported number of pages. If it does, a new page directory/page transiton table is allocated, and so on until the page table level has a mximum page index bigger or equal with the *end index*. Then we start to (un)map pages from the IOMMU page tables from *start index* to the *end index*. To do this we use a recursive function, that basically performs something like a BDF (breadth-first search), starting from the page directory/page translation table corresponding to the first *start index* and *end index*, changing the start and end indexes accordingly, at each recursion.

Impelementation details:

The implementation of the allocator can be found in the following two files:

- **iommu_mappings_amd.h** This file contains the definitions that help to easily modify the page directory entries and the page translation entries fields. It also

defines some constants that are used for page tables manipulation, such as the size of the page (12 bits), the size of the page table level (9 bits), etc.

- **iommu_mappings_amd.c** The functions worth mentioned are:

```
int map_pa(u16_t devid, u32_t pa, u32_t size,
                                u32_t *va, int flags)
```

This function maps the requested number of pages into IOMMU page tables. It accepts the following parameters:

devid - the device id (BDF) for which the mapping is performed

pa - the physical address of the first memory location to be mapped (the whole memory zone must be contignous in memory)

size - size in number of bytes

va - it can be an input, as well as an output parameter. As an output parameter, it will contain the virtual address of the first page, as returned by the simple allocator. As an input parameter, it will contain the virtual address of the first page at which the mapping should start.

```
int unmap_pa(u16_t devid, u32_t va,
                                  u32_t size)
```

This function is similar with the previous one, but modifies in a different way the page tables entry, i.e. it unsets the page present bit.

IOMMU's driver interface

This chapter describes the interface provided by the IOMMU driver. It is bassically a guide to driver developers who need to use the IOMMU functions.

The following two functions, implemented in *libsys*, should be called when a driver instructs a device to perform DMA:

```
void *iommu_mmap(u16_t devid, size_t size,
                 phys_bytes *addr, int flags)
```

This function maps a requested number of pages into the IOMMU page tables. By default, it first will try to allocate a DMA buffer and then map this buffer into the IOMMU page tables. However, it can be instructed not to allocate a buffer, but to use a given one. Also, by default, the IOMMU driver uses the physical address of the DMA buffer as the virtual address into the page tables (it doesn't generate a new virtual address, but maps the physical address of the buffer to a virtual address, i.e. the index in the page tables, equal to the buffer's physical address). It accepts the following parameters:

devid - the BDF of the device

size - size in bytes of the requested memory zone

addr - it contains one of the following:

- (output parameter) the physical address of the allocated DMA buffer, if no IOMMU is present
- (output parameter) the virtual address as mapped by the IOMMU driver (this must be used by the device when performing DMA)
- (input parameter) the physical address of the already supplied DMA buffer, when used in conjunction with IOMMU_USE_BUF flag

flags - can be one or more of the following (combined with the "OR" bit operator):

- IOMMU_USE_BUF - instructs the mapping function to use the provided buffer (in **addr** parameter)
- IOMMU_R - read access for the device
- IOMMU_W - write access for the device

- IOMMU_GEN_VA - instructs the IOMMU driver to generate a virtual address where to map the physical address of the buffer

```
int iommu_munmap(u16_t devid, size_t size,
                 vir_bytes vaddr, phys_bytes ioaddr, int flags)
```

This function unmaps the requested pages from the IOMMU's page tables. It accepts the following parameters:

devid - the BDF of the device

size - size in bytes of the memory zone that must be unmapped

vaddr - the virtual address of the DMA buffer, as returned by alloc_contig()

ioaddr - the address used by the device to perfom DMA, i.e. the virtual address at which the memory pages have been mapped by the IOMMU driver

flags - can be one or more of the following (combined with the "OR" bit operator):

- IOMMU_KEEP_BUF - the dafault operation of the unmap function is to first free the DMA buffer and then unmap the pages from IOMMU's page tables. This flags tells the unmap function not to free the DMA buffer.

Modifications/Additions to other MINIX components

While implementing the IOMMU driver, we had to do several modifications/additions to the existing MINIX components. In this chapter we briefly describe these modifications/additions.

VM server: To make the IOMMU driver restartable, we modified the VM server to record each DMA memory request. When the IOMMU driver is started/restarted, it will first ask for these requests from the VM server and then it will map them in new page tables and instruct the IOMMU to use these page tables. Also, when a driver that uses DMA dies, the VM will have to clear this driver's mappings and then inform the IOMMU about the dead process so that the IOMMU driver could also cleanup its page tables accordingly.

The modifications can be seen in file *servers/vm/alloc.c* and they mainly consist in the following:

- the structure that contains a DMA memory request/mapping:

```
#define NR_IOMMU_MMAPPINGS 1024

PRIVATE struct iommu_table
{
        int flags;
        u16_t dev_bdf;
        phys_bytes dev_base;
        phys_bytes pa_base;
        phys_bytes size;
        endpoint_t source;
} iommutab[NR_IOMMU_MMAPPINGS];
```

  To record the mappings, we used a static buffer of 1024 entries. Considering that most of the drivers will require only one DMA buffer, the number of entries should suffice. Only the disk driver required more that one buffer. However, while testing, the maximum number of entries required in total was 50.

- function *do_iommu_register()* which *sys_safecopyto()* the requestor the mappings vector. This function is called when the IOMMU driver registers at

the VM. To register to the VM, the driver should use the following function, implemented in *lib/libc/other/_vm_iommu_calls.c*:

```
int vm_iommu_register(cp_grant_id_t gid)
```

- VM mapping functions *do_iommu_mmap()* and *do_iommu_munmap()* used to manipulate the mappings vector. These functions are used by those implemented in *lib/libc/other/_vm_iommu_calls.c*, which, in turn, are used by those described in 4.
- function *clear_iommutab()*. This function is called when VM learns that a process died. It searches into the iommutab structure for possible mappings pertaing to the dead process. If these mappings exist, they are removed from the mappings array and then the IOMMU gets notified about this.

PCI driver: We modified the PCI driver to support the following function (implemented in *drivers/pci/pci.c*):

```
int pci_bdf_s(int devind, u16_t *bdf)
```

This function returns the BDF (bus, device, function) for the given deviceid, i.e. the index in the PCI devices vector. The correspondig library function is implemented in *libsys* in file *lib/libsys/pci_bdf.c* and has the following signature:

```
void pci_bdf(int devind, u16_t *bdf)
```

ACPI driver: As the IOMMU driver has to parse the IVRS ACPI tables, we needed to implement two more function in the ACPI driver (in file **drivers/acpi/acpi_utils.c**):

```
void do_get_acpi_table_size(message *m)
```

This function returns the size of the requested table.

```
void do_get_acpi_table_size(message *m)
```

This function copies the requested table to the requesting process memory space. The requesting process must first request the table size of the table, allocate the necessary memory space and create a grant to let the acpi driver copy the table into its memory space (using *sys_safecopyto*).

To see an example that uses the above functions, see the function *iommu_acpi_parse_ivrs()*, implemented in file **drivers/iommu/iommu_acpi_amd.c**.

Future Work

There is more functionality that can be added to the IOMMU driver. There are some things that need to be implemented and there are also some things that are implemented, but need to be tested on a machine that presents the needed environment.

- **support for INTEL's IOMMU**: As the AMD driver is already written, and the architecture was designed such as to support both the AMD and INTEL IOMMU devices, this task should be easier. The two specifications are much alike, for example, they both use the ACPI tables to store almost the same information, they both use similar structures for the mappings, i.e. the page tables, etc.
- **event logging triggered by interupts**: Currently, MINIX doesn't support MSI/MSI-X, so if an event happens, it is just put into the Event Log Buffer, but no interrupt is generated to announce the event.
- **IOMMUs in the same function**: A device may implement more than one IOMMU within a single function. Configuration and status information for the IOMMU are mapped into PCI configuration space using a PCI capability block. One or more IOMMU capability blocks may be implemented in a function. This part has not been tested as the testing machine has only one IOMMU.
- **IVMD structure**: Platform firmware (BIOS) may have memory usage requirements to communicate to system software based on its needs or on hardware characteristics. Platform firmware can inform system software of memory usage restrictions or requirements by using I/O Virtualization Memory Definition (IVMD) blocks. Each IVMD entry may be per-device, specifying the DeviceID to which the entry applies, or the IVMD entry may apply to all devices and the DeviceID is ignored. The code parsing the IVMD structure needs to be implemented and tested on an architecture that allows it. On the current architecture, the IVMD structure does not exist.
- **improve page tables manipulation**: The mapping algorithm could be improved by filling-in the page tables such as to skip levels where it is possible (a mechanism supported by the AMD IOMMU). For example, if all the 9 bits form the virtual address designating a level are 0, then the PDE could be filled-in such as to point to a (level - 2) PDE/PTE, instead of pointing to a consecutive lower level.

- **USB support**: Currently, MINIX does not have support for USB so the USB is emulated by the BIOS. The BIOS should have built an IVMD structure in which it should have specified the memory zone reserved for the USB, so that the IOMMU driver could instruct the IOMMU device to allow the USB access to these memory zones. However, the BIOS doesn't build the IVMD structure so the IOMMU blocks all DMA accesses performed by the USB. To workaround this problem, we use an environment variable that can be used (in the boot monitor) to specify which devices are allowed to perform DMA. For example: **iommu_exclude=165:144:45:146** will instruct the IOMMU to allow DMA performed by devices 165, 144, 145 and 146. This problem should dissapear once the USB driver is be implemented.

Bibliography

[amd09] AMD I/O Virtualization Technology (IOMMU) Specification. http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf, February 2009. 1, 4, 7, 8, 9

[int08] Intel® Virtualization Technology for Directed I/O. ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf, September 2008. 1