UiO : Department of Informatics
University of Oslo

PCIe Device Lending

Using Non-Transparent Bridges to Share Devices

Lars Bjørlykke Kristiansen

Master's Thesis Spring 2015

PCIe Device Lending

Lars Bjørlykke Kristiansen

11th May 2015

Abstract

We have developed a proof of concept for allowing a PCI Express device attached to one computer to be used by another computer without any software intermediate on the data path. The device driver runs on a physically separate machine from the device, but our implementation allows the device driver and device to communicate as if the device and driver were in the same machine, without modifying either the driver or the device. The kernel and higher level software can utilize the device as if it were a local device.

A device will not be used by two separate machines at the same time, but a machine can transfer the control of a local device to a remote machine. We have named this concept "device lending". We envision that machines will have, in addition to local PCIe devices, access to a pool of remote PCIe devices. When a machine needs more device resources, additional devices can be dynamically borrowed from other machines with devices to spare. These devices can be located in a dedicated external cabinet, or be devices inserted into internal slots in a normal computer.

The device lending is implemented using a Non-Transparent Bridge (NTB), a native PCIe interconnect that should offer performance close to that of a locally connected device. Devices that are not currently being lent to another host will not be affected in any way. NTBs are available as add-ons for any PCIe based computer and are included in newer Intel Xeon CPUs.

The proof of concept we created was implemented for Linux, on top of the APIs provided by our NTB vendor, Dolphin. The host borrowing a device has a kernel module to provide the necessary software support and the other host has a user space daemon. No additional software modifications or hardware is required, nor special support from the devices. The current implementation works with some devices, but has some problems with others. We believe however, that we have identified the problems and how to improve the situation. In a later implementation, we believe that all devices we have tested can be made to work correctly and with very high performance.

Contents

List of Figures

List of Tables

List of code snippets

Chapter 1

Introduction

1.1 Background

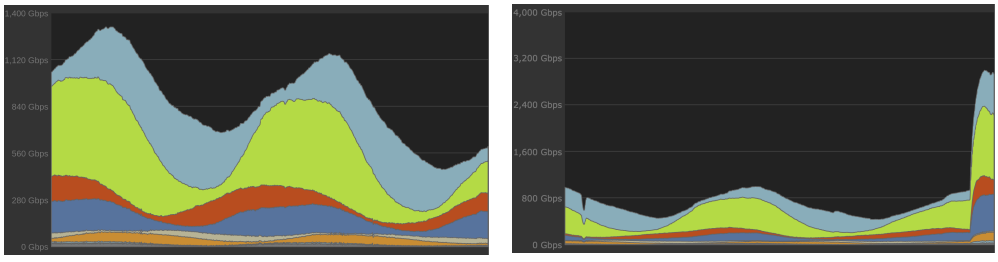
The current trend in Internet services is "cloud computing", where the applications are running in "the cloud" and users have access to their data from their smartphone, tablet or a computer anywhere in the world. Most of the time, the cloud is not just used as a data storage for user applications, but instead, the applications themselves run in the cloud, and only a thin client application runs on the users device.

Cloud computing has gained popularity not only from being convenient for users, but also for its flexibility for the developers and service owners. The burden of maintaining the physical infrastructure is placed on the cloud provider instead of the application developer which allows the developer to focus on developing the application. This is also often cheaper for the developers compared to running their own servers. As the developer creates new applications or the use of their existing application changes, they can rent additional computers and change the specification of their computers to meet the new demands. The machines can also be located around the world to be geographically near their customers.

Cloud providers are able to rent out cheap machines thanks to one important technology: Virtualization. This technology allows multiple virtual machines (VM) to run on a single physical machine. Combined with the ever-increasing advances in computing power, storage and hardware in general, virtualization enables cloud providers to rent out virtual machines at a very low cost. The cloud providers saves physical space, power consumption, hardware cost and maintinance cost by having a smaller number of powerful machines with multiple VMs.

Since multiple virtual machines run on a single physical machine, they share the resources

Figure 1.1: Download rates over 48 hours from the game platform Steam [4] North America is coloured blue, Europe is green.



(a) A normal 48 hour snapshot of Steam downloads.

(b) The sudden spike on the right is GTA V getting released for downloads (preload)

of the host. Some of the resources allocated to each VM are more static than others, for example, when a VM is created, the amount of RAM, number of CPU cores and network interfaces are given. The allocated resources can however in many cases be changed without powering down the VM. Some of the resources are however more naturally shared between the VMs including CPU time (time each virtual core executes on a physical core), disk access (bandwidth) and network bandwidth. These resources are time-shared between the VMs and each VMs time-slice can be dynamically changed and scale to meet the demands of the VM at any given time. If all of the virtual machines use all their resources all the time, this sharing will be close to static. In a lot of situations, however, the demands will change over time and can be "bursty". An example of this can be seen in the day-night cycle in the download rates on the Steam gaming platform in figure 1.1. Other services might have short term spikes in their resource usage such as content generation, simulations, nightly code compilations or a cute cat video gone viral. This can also be seen in the Steam download rates when GTA V is released for download. A VM provider can take advantage of the bursty nature of most computation and overprovision the combined resources of the VMs on a single host. If the cloud provider is able to quickly react to changes in resource usage, the cloud provider can rent out more computing power than they physically have. When the sum of resources used by the VMs on a host gets close to the limit, one or more of the VMs can be migrated, live, to another host without the users noticing it and when the combined VM usage is low, some hosts can be completely freed of VMs. These temporarily idle hosts can now be powered down to conserve power or for upgrades and maintenance.

In addition to the more traditional machine resources, newer VM software has support for allowing VMs to directly access some types of hardware devices. This is useful when the extra performance cost of emulating or otherwise intercepting the hardware affects the performance too much or emulation is infeasible. This direct device assignment is enabled by hardware virtualization support called IO Memory Management Units (IOMMU). In the same way that a normal Memory Management Unit (MMU) allows processes to have their own address space and isolate each process from each other and the kernel, an IOMMU isolates each VM and the VM's assigned devices from the host and other VMs.

With an IOMMU, a VM can be given direct control of a hardware device without the VM being able to break out of its isolation. For example, a dedicated network card can be given exclusively to a VM which gives the VM control of all packets to and from this network card. The host OS or other VMs will not be able to use a device that has been assigned to a VM. While directly assigning hardware devices to a VM improves performance by lowering overhead, it conflicts with the main benefit of virtualization, sharing powerful hardware dynamically.

Some network cards have multiple Ethernet connectors, each one seen as a separate device from the software perspective. In PCI terms this is called a multi-function device. Other devices can also be multi-function, for instance most Graphical Processing Units (GPU) with HDMI connectors have a dedicated audio function in addition to the GPU itself. Each function will have it's own driver instance in an OS and the driver does not need to be aware of the fact that the device is part of a multi-function device. Each of these functions can be separately assigned to a VM, for instance, assigning a single Ethernet port on a multi-port Ethernet card saves physical space in the host machine compared to each VM having it's own dedicated card.

The ideal would be to allow VMs direct control of the hardware to have low overhead, but still share the devices with multiple other VMs. The PCI standard Single Root IO Virtualization (SR-IOV) is a solution to this problem. A device that supports SR-IOV can appear as if it has multiple virtual functions (VF) and from the software perspective, this is similar to multi-function devices. Each VF is isolated from each other and can be safely assigned to a VM, but unlike with ordinary multi-function devices, VFs will share the same physical device resources.

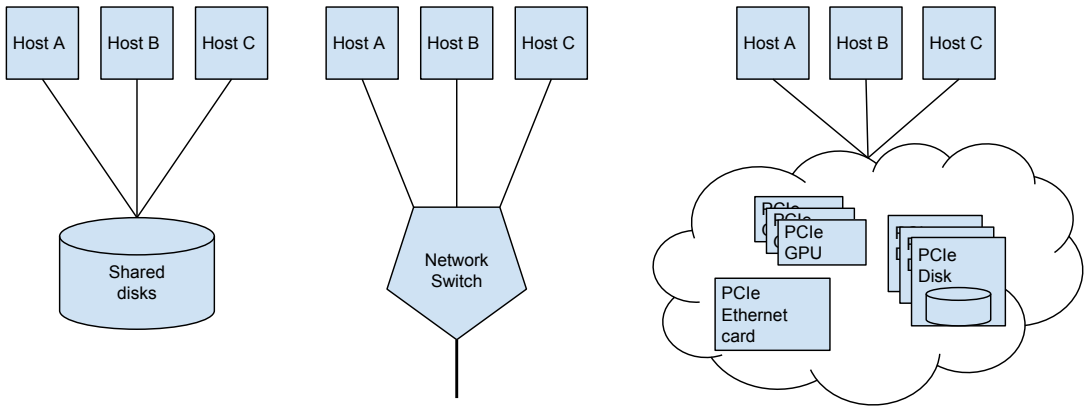


Figure 1.2: Various device pools

For instance, a network card with SR-IOV can allow multiple VMs to share a single Ethernet connector and the device itself will take care of sharing its resources with the VMs accessing it. This is done without affecting the performance benefits of direct hardware control.

Virtualization is beneficial when the application requirements are below the resource limits of the physical hardware, but just as easily, the resources required by an application can be greater than the resources in a single physical machine. To cope with this, the machine resources will need to be increased, but unlike with VMs, a physical machine cannot be dynamically upgraded with the touch of a button. If a host needs more storage space, a new hard drive must be added. The resources provided by hard drives, GPUs, network cards and other hardware devices typically come in chunks unlike like the fluid resources that can be assigned to VMs. If this is more than what is needed, the extra resources will either go unused, or create additional complexity in assigning VMs to physical machines. Additionally, adding more hardware to a host is a lot less dynamic than assigning hardware resources to VMs. There are also practical limits to how far a single host can be upgraded in terms of number of devices. Avoiding this problem can mean distributing the application to run on multiple servers. This is not only substantial work, but depending on the resources that are insufficient, can be impractical or difficult to do efficiently. If only one resource is lacking, for instance hard drive space or GPU power, distributing the work is not ideal.

If the hardware resources can be shared between multiple physical machines in the same way as resources can be shared between VMs, flexibility and resource utilization can be increased. This also no longer applies to only virtualization, but non-virtualized server settings can also benefit from sharing resources with other servers and dynamically assigning the resources to servers that need them. Allowing the resources used by a server to go beyond it's physical confines and be shared by multiple servers will make using optimal price-performance hardware instead of more expensive, powerful hardware possible. Such a setting can be seen as servers having access to a pool of resources as seen in figure 1.2.

A Network Attached Storage (NAS) can be seen as a pool of storage space that can be dynamically assigned and shared by multiple hosts and it is perhaps the most widely used form of inter-host resource sharing. A NAS can give a host access to greater storage space and potentially higher performance, often for a lower price than a locally attached disk in each machine with the same performance and capacity. For a VM provider, having a storage pool makes the assignment of VMs to hosts easier because the storage space required by the VM is no longer a factor in assigning VMs to physical hosts. Also, without the storage pool, if the assignment was not perfect, some of the hosts would have unused storage space, but with a

storage pool, the free space is not fragmented and is available for all hosts to consume. Taking bandwidth into consideration as well, some hosts might have VMs with high disk activity and others where the disk is used for cold storage, but in a NAS the total bandwidth can can also be combined using RAID and shared with all users. The result is that a host potentially has access to more disk space with higher bandwidth while the total number of disks is lower. The added flexibility of a storage pool applies to other resources as well, although not all of them are as easy to share efficiently while retaining the same level of performance.

In some ways, having a "top of the rack" network switch can be seen as a network pool. The servers in the rack are connected to a switch, which in turn is connected by a high speed link to the rest of the network. Compared to having a link from each server to the rest of the network, the cost and complexity should be greatly reduced and depending on the hardware, performance can be as good or better. Indeed, the switch can come at substantially lower cost than a dedicated fiber link in each server.

So far, each of the device pool types have been specific to a single type of device. A more flexible alternative would be a generic device pool that can share any device type and be used to share network, storage and GPUs. Allowing the hosts to share a single high speed network interface eliminates the "top of the rack" switch, the intra rack network cables, and the network cards in every server. This can reduce costs but also increase performance as all servers have the ability to achieve the peak bandwidth of the shared network card. In addition, increasing the network bandwidth of all the servers can be achieved by adding a single new card. This card can be used by all servers in combination with the other or the two card can be split among the servers. The same IO sharing pool can be used to share storage devices and GPUs.

1.2 Problem statement

The IO pool technologies available today are often vendor-dependent and part of a rack architecture. This can make the price too high for some use cases and locks a user to a specific hardware vendor. In this thesis, we investigate the generic IO device pool idea and we examine and evaluate existing solutions, standards and ideas before designing and implementing our own. We aim for our design to be relatively low cost, and require little user effort, so the devices need to be usable without modified drivers. We also want keep the needed additional hardware to a minimum and aim at designing a solution allowing an existing server cluster to be upgraded.

To acheive our goals, we develop a mechanism that allows PCIe devices to be dynamically reassigned from one machine to another. Specifically, the devices inside one server can be used by another machine either locally attached to a host or in an external cabinet. Our implementation is implemented for the Linux kernel and the PCIe based interconnect available from Dolphin Interconnect Solutions (Dolphin).

1.3 Main contributions

PCIe Multi-Root IO Virtualization (MR-IOV) is a standard for sharing multiple devices between multiple hosts. This vendor neutral PCIe standard can be used to create a pool of IO devices including disks, network cards and GPUs. Unfortunately, compliant hardware is virtually non-existent. Various vendors have created alternative PCIe device pool solutions, but most have significant limitations. Perhaps the most common way to get a device pool is as part of a complete rack solution, which creates a vendor lock-in. Possibly, the devices themselves are

part of a fixed package, and thus the flexibility is greatly reduced.

Our proof of concept design is unique because it lacks these limitations. First of all, the hardware required is easily available from multiple vendors and the rest of the solution is software based and can be converted to work with the different hardware available. Unlike most set-ups where all devices are placed in a pool outside the machines, often with a proprietary interconnect, our proposal is based on a standard PCIe interconnect. While the devices can be placed in an external enclosure, locally attached devices can also be used by remote machines. This is possible because our solution works at the software layer and the devices are, as in traditional PCIe, owned by only one computer. Instead of reassigning the device from one computer to another, the device can be controlled by the other computer with no change to the PCIe layer, instead, software arbitrates which computer controls a device. Since each device is controlled by a single computer at a time, we have named this "device lending". Because our implementation only deals with the remote access of a device connected to another computer, access to devices directly attached will not be affected in any way.

1.4 Limitations

In this thesis, we have discussed multiple other solutions for multi-host PCIe sharing, but because of the lack of availability for most of the solutions, we were unable to compare real world benchmarks of our solution to the others.

For our proof of concept, we describe multiple design alternatives , but to limit the scope of this thesis, not all were implemented. Also, some functionalities which we believe to be critical for optimal performance where not implemented since they were not necessary for a simple working prototype. We prioritized the implementation of the functionalities required for a few selected devices to work. In particular, functionalities considered legacy and not used in modern devices, such as port IO, was not implemented.

1.5 Research method

In this thesis, we have followed the design paradigm as defined by Association for Computing Machinery [9]. Following this, our goal was to specify, design and implement a proof of concept. The implementation is tested in order to validate the design. Because of our incomplete understanding of the background material when starting this thesis, the design and requirements changed multiple times before we settled on a final design. The various designs that we discussed, tested and abandoned are documented in the thesis as well as what we learned under way.

1.6 Overview

This thesis will begin with chapter 2, on PCIe, in which we will detail the workings of PCIe, focusing on what we were required to know before creating our proof of concept and what we learned. In the next chapter, chapter 3, we will cover the background material needed for our implementation related to the Linux kernel, focusing on the interface between a device driver and a device. After this, all background information needed to explain the implementation is covered. Chapter 4 explains the implementation and our design choices. While chapter 4 details the final implementation, the next chapter, chapter 5 tells more of how we gradually developed

the solution by testing different devices and and debugging the interaction between device driver and device.

Chapter 2

PCI Express

Peripheral Component Interconnect (PCI) is a standardized bus used in different computer architectures. Any computer architecture that has a PCI bus can use the same type of devices. Introduced in 1992, its an old standard and has been superseded by PCI Express (PCIe). Currently the newest version of the PCIe specification [20] is 3.1. Both PCI and PCIe are standards developed by PCI-SIG.

Software-wise, PCIe is fully backwards-compatible with its predecessor. This allows operating systems and drivers written for PCI to work with PCIe. The identical software architecture made the transition easier for developers and software as drivers and operating systems designed for conventional PCI would support PCIe out of the box. This also means that the legacy PCI specification (PCI Local Bus Specification [19]) is still relevant for PCIe. From the hardware side on the other hand, PCIe and PCI are quite different. One of the more important differences are that in PCI, most devices are on the same bus and thus, they share bandwidth. This meant that two devices could not use the maximum bandwidth provided by PCI at the same time, PCIe, on the other hand, has no shared bus and is, in fact, packet based.

PCIe is layered in a similar way to TCP/IP. The top layer is the transaction layer containing Transaction Layer Packets (TLP) which are mostly memory read or write requests. In addition, there is also port IO requests, legacy interrupts and some internal events. Below is the Data Link Layer which guarantees the delivery of TLPs and is responsible for retransmissions and error corrections. At the bottom is the physical layer, dealing with circuitry and such and is specific to the physical medium.

2.1 PCI device

Each PCI device can be identified by its location in the PCI tree. Its location is given by the bus it's connected to and a unique device number on this bus and each device can have multiple functions. Each function is identified by the combination of <bus number, device number, function number>, the "bus-device-function" (BDF).

When the machine boots up, the platform firmware will scan for PCI devices on all buses, the device <X,0,0> is always present. The firmware discovers a device by reading its *configuration space*. The contents of the configuration space is standardized and is either a type 1 which is used for bridges, and type 0 which is used for all other devices, including endpoint devices and it can seen in figure 2.1. The contents of a device's configuration space allows the firmware to determine the type of device and other vital information about it.

If there is no device for a given BDF, all configuration reads will yield `0xff`. This allows the system to discover the valid BDFs and the types of devices present. The devices' most basic

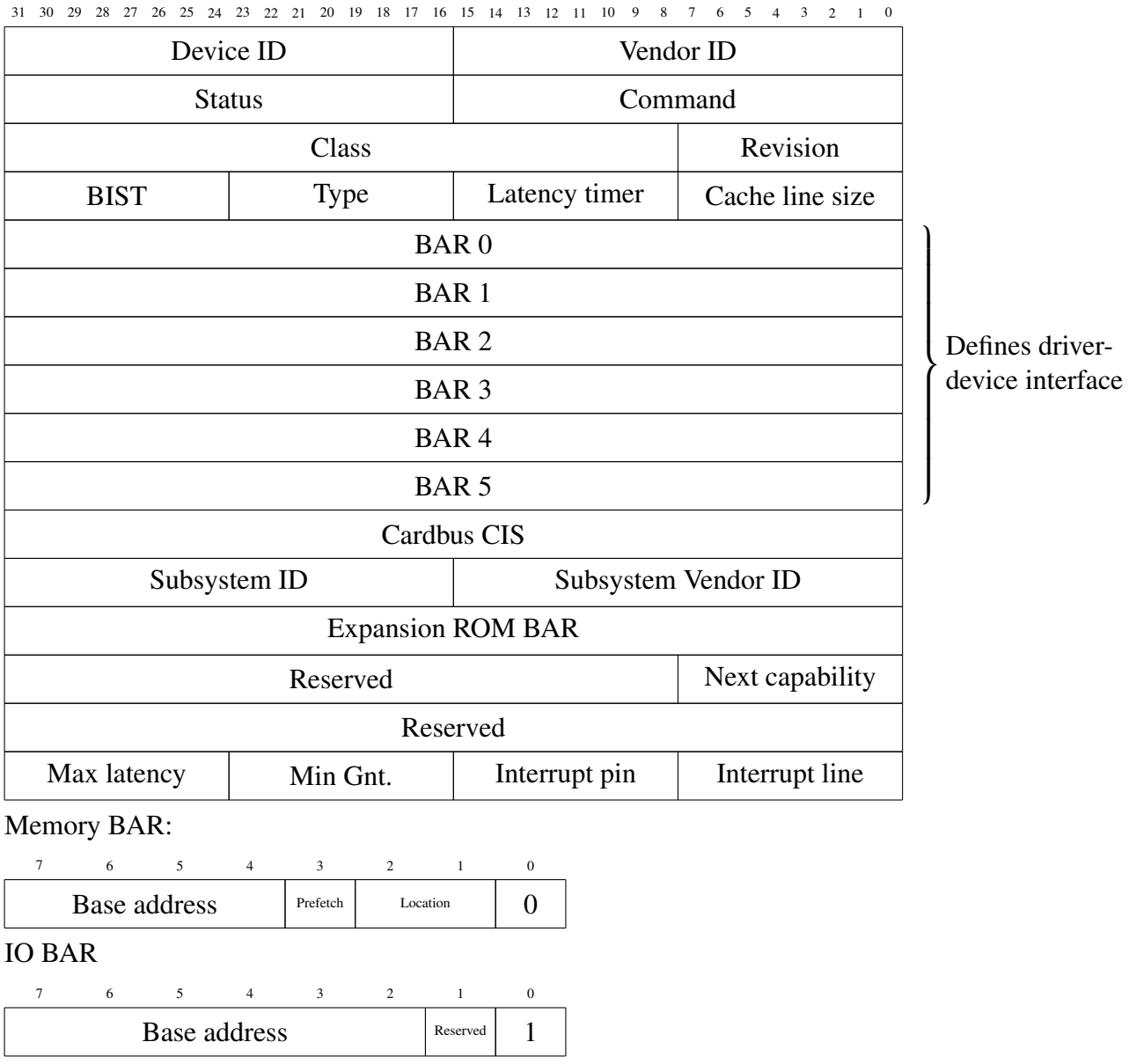


Figure 2.1: The configuration space of a PCI device

configurations will also be set by writing to the configuration space.

The "next capability" register in the configuration space of a device, is a pointer into another place in a device's configuration space, to a linked list of capability structures. Each capability structure has a next pointer and an ID defining its type. Capability structures are used to define additional properties of the device. New capability structures can be defined in the future. One example of a capability is the Message Signalled Interrupt capability seen in figure 2.6. In PCI-X 2.0 ("PCI extended", not PCIe), the configuration space was extended from 256 bytes to 4KB. The extended configuration space is also present in PCIe devices and it is used for an additional linked list of capabilities. A PCIe device will show the system that it is a PCIe device by implementing the PCIe capability structure [20].

The configuration space is not the main interface for a device, but rather, it allows the device to define its interface and its requirements. The BAR registers inform the system of the memory-mapped (MMIO) registers and IO port ranges of the device. Port mapped IO is however deprecated in the PCIe standard so we will not spend much time discussing it. The system uses the MMIO BAR registers to discover the size and number of device-defined memory-mapped areas containing device registers. These areas are used by a device driver to communicate with the device. Unlike the configuration space, the contents of the BAR areas are device-specific and not part of the PCI or PCIe standard. Once the BAR registers are programmed, the system can read and write to their memory addresses to interact with the device.

Each MMIO BAR can be either prefetchable or non-prefetchable. The prefetchable areas behave a lot like normal memory because they are guaranteed to have no side effects on read operations. This allows the system to do preemptive reads and merge operations to increase performance. The MMIO range defined by a non-prefetchable BAR cannot be prefetched by the system. Also non-prefetchable memory can only be placed in the lower 4GB of memory since the addresses are 32-bit only. Non-prefetchable memory is only intended to be used for control registers and such, and not for large storage spaces. Since non-prefetchable BARs are not used for large memory areas, the 32 bit limitation is not significant.

The BARs and the configuration space allow the CPU to interact with the device, but the device can also interact with the host machine. First of all, it can raise interrupts to tell the CPU that it has completed a task or some event has occurred. Also, devices can, like the CPU, access memory by using Direct Memory Access (DMA), to reach the RAM and MMIO registers. Without DMA, the CPU would have to write to MMIO registers of the device to transfer data, but this would occupy the CPU while the transfer was in progress. Devices with DMA engines can instead be instructed to read buffers directly from main memory.

2.2 Transparent bridges and PCIe switches

In traditional PCI, most devices shared a single bus. On this bus, only one device could communicate at a time which caused the total bandwidth on this bus to be shared. Some PCI systems however, had two or more buses that were connected to each other by a PCI-PCI bridge. Each bridge has two distinct ends: primary and secondary. The bus connected to the primary side is closer to the chipset of the system and the secondary is closer to the endpoint devices. The bridges forwards traffic according to a set of rules. The bridge will forward traffic it receives on the primary bus to the secondary bus, *downstream*, if the destination of the traffic is within the address ranges configured for the bus. In the same way, traffic on the secondary bus is forwarded upstream (secondary to primary) if the destination is not within the range of the bridge. The bridges forward all PCI traffic including interrupts, MMIO and configuration space accesses. In PCIe buses and bridges are reused to model the PCIe switches, see figure

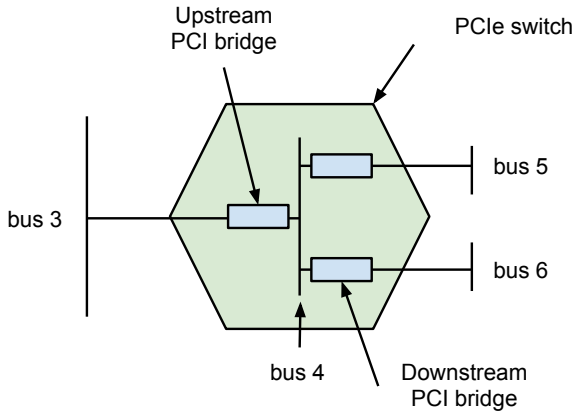


Figure 2.2: A PCIe switch is composed of multiple PCI-PCI bridges and a bus.

2.2. All switches have an internal PCI bus. A single bridge, also referred to as the upstream port, connects the internal bus to another bus closer to the chipset / root complex. All the other bridges are downstream and are called downstream ports. At the top level of a PCIe fabric are the *root ports*. These ports are directly connected to the root complex of the machine. PCIe switches and PCI bridges route memory accesses directly to its destination. A transaction from a device to another device's MMIO registers does not go through the chipset or the CPU, but takes the shortest path through the PCIe fabric. This is sometimes referred to as Peer To Peer transactions (P2P) and can be very efficient, for instance data can be transferred from one GPU to another without involving the RAM or the CPU.

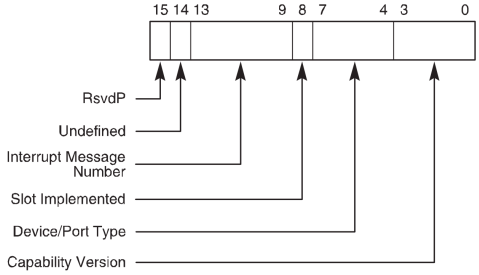
2.2.1 Hotplug capable PCIe slots

The PCIe capability structure declares various PCIe specific properties of a device or bridge including the PCIe capabilities register which can be seen in figure 2.3(a). For bridges, this includes whether the bridge is the upstream or downstream bridge in the switch and downstream bridges can have an additional bit set to indicate that it is a "slot". Slots are physical connectors that other devices can be connected to. Each slot has a dedicated slot capability register that can be seen in figure 2.3(b) and for the most part declares hotplug related capabilities. During the scan of the PCIe fabric, the OS will read these registers and learn the features and capabilities of the slot.

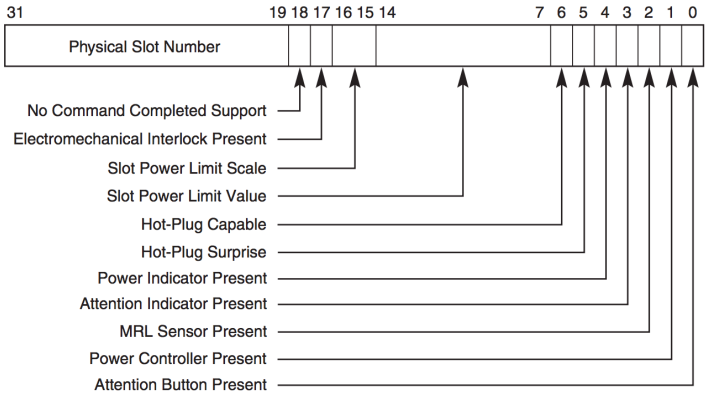
A slot indicates to the system software that is capable of hotplugging by setting the hotplug bit in the slot capability register. This signals that the tree beyond the bridge can be removed and replaced while the system is running. Section 6.4 of the PCIe specification specifies three parts of hotplug and specifies how both the software and the bridge should behave to implement hotplug.

1. The operating system needs to detect and react to the hotplug events.
2. The physical connector needs to allow a device to be removed or added without damaging any components.
3. The PCI-bridge associated with this port needs to communicate the status of the slot to the OS.

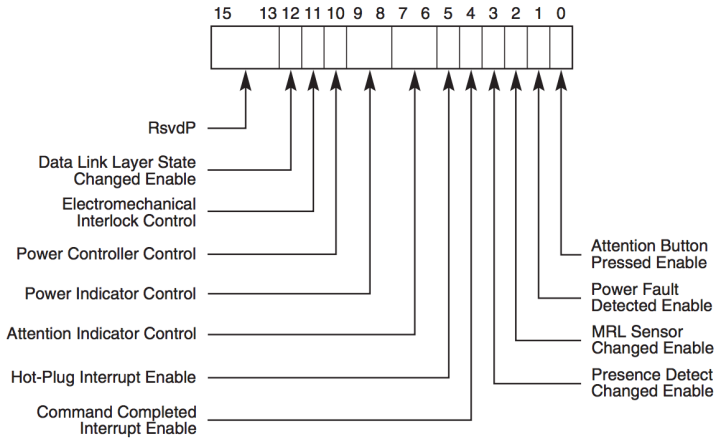
The various hotplug related features of a slot can be enabled and controlled by the system software with the control register seen in figure 2.3(c). Finally, the status register, in figure



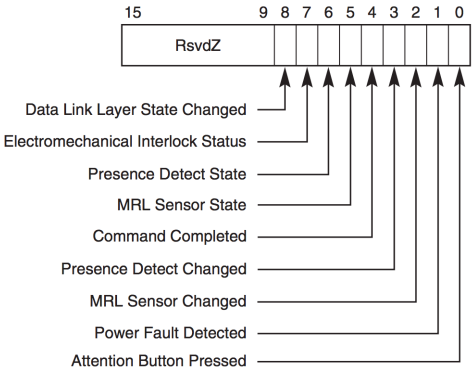
(a) PCIe capabilities



(b) Slot capabilities



(c) Slot control



(d) Slot status

The figures are borrowed from the PCIe specification [20].

Figure 2.3: Registers related to hot-plug in the PCIe Capability Structure of a downstream port

2.3(d), is used by the OS for reading the state of a slot.

For the OS to detect when the slot is disconnected or connected on a hotplug-capable slot the OS first needs to set a bit in the control register: "hot plug interrupt enable". When this bit is set, the slot will generate an interrupt when a hotplug event occurs. When such an interrupt arrives, the OS can read the status register of the slot to detect if anything is plugged into or removed from the slot. When a hot-add event is detected, the OS will perform a scan of the devices behind the bridge, the same scan as when the system boots, by reading the new device's configuration space.

For a slot with the surprise hotplug bit set, a hotplug event can happen at any time. In both, slots that are capable of surprise hotplug and those that are not, there are multiple mechanisms for negotiating between the OS and user before the slot is disconnected. For instance, to assist in human interaction, two indicator lights may be present on the slot: the attention light and the power light which are controlled by the operating system. The OS can enable the power indicator to indicate that the device is powered and may not be removed and the attention light is used to allow the OS to indicate to the user the identity of a slot or call attention to it. For instance, when the OS has prepared a device for removal, the OS may blink the attention light to indicate that the device is ready for unplugging. On the other hand, the attention button may be pressed by a user, for instance to request removal of the device and the system can indicate a response with the attention light. Additionally, the specification says that a "software user interface" can be implemented. To help the user in knowing which physical slot corresponds to a slot in the OS user interface, the slots may be numbered and the slot manufacturer will have user-visible number on the slot and the same number will be programmed into the slot capability register. This makes it easier for the user to see, for example, the slot that contains a malfunctioning card, in much the same way as the indicators.

The manually-operated retention latch (MRL) is a mechanical mechanism that holds a device securely in place and a sensor may be implemented to alert the system of an imminent disconnect or of a new connection. If the power controller is also implemented, the MRL sensor will automatically cut power to the device when a disconnection is sensed. An electromechanical interlock can also be present and used by the OS to physically lock the device in place when the device is not ready to be disconnected.

2.2.2 Hotplug from a software perspective

When a device is hot-added, its position in the PCIe fabric is always behind the bridge of the associated hotplug slot. Adding a new device introduces new bridges, buses and end point devices with associated MMIO areas and port IO ranges, depending on the kind of device that is added. Because of how bridges work in the tree-structured PCIe, all of the new device's resources must fit within the resources of the slot. Adding a new device becomes a problem when the bridge associated with the slot does not have sufficient space for the new device. The windows allocated to the slot must be within the windows of the bridge upstream of the slot. This applies to bus number, IO space, prefetchable and non-prefetchable memory which each have separate windows that all need to fit. Fixing this is hard, as the additional space must come from the upstream bridge and this applies recursively. Depending on the configuration of the tree upstream, the resource windows of the slot can be expanded by expanding the windows of all upstream bridges. Often however, this conflicts with the ranges of an entirely different part of the tree. Figure 2.4 shows a device behind a slot and the resource windows allocated to the slot as well as what is needed by the device behind the slot. In the figure, removing the NIC and replacing it with a device that consumes more memory resources poses a problem. For the connecting bridge to expand to accommodate this, there needs to be free space on either side of

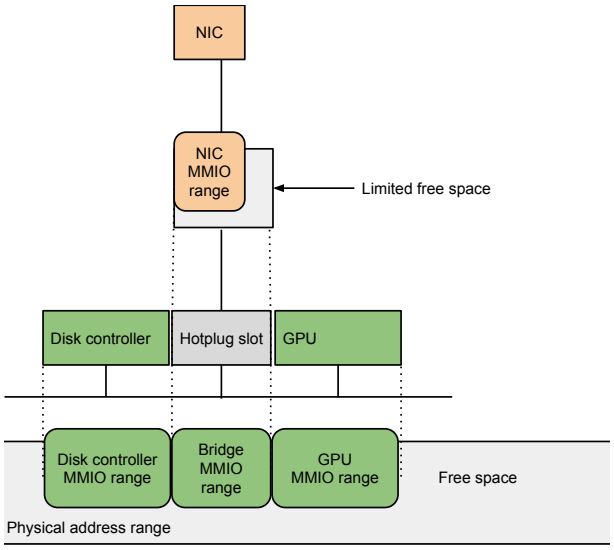


Figure 2.4: Devices behind a bridge must have their BARs within the resource windows of the upstream bridge

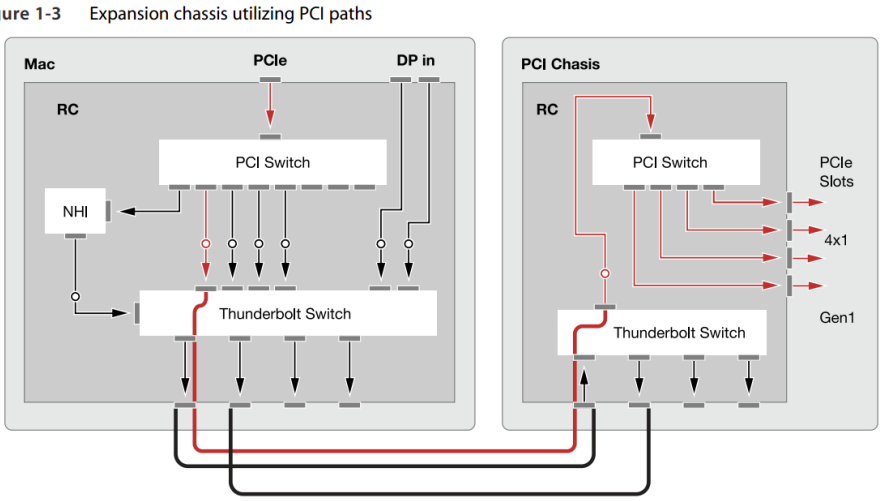
its range. In the figure however, both sides are occupied. One of the devices' resource windows must be moved for the system to be able to allocate the required resources to the new device. Moving the resource windows of a part of the PCIe fabric is however, not trivial as it would disrupt traffic for all downstream devices and associated drivers. If hotplugging is allowed, you can end up in situations where the total available capacity of the system allows the device to be added, but the resources available for the hotplug slot is insufficient and the new device cannot be used. To avoid the lack of resources in a hotplug slot, the system can allocate more space at boot time than what is required for any device connected at boot time.

2.3 Thunderbolt

Thunderbolt is a consumer centric, high speed, external interconnect developed by Intel and Apple. The technology repurposes the Mini DisplayPort formfactor and are backwards compatible, falling back to normal Mini DisplayPort if a DisplayPort device is detected. Native Thunderbolt communication is a custom protocol that can tunnel both Mini DisplayPort traffic and PCIe traffic at the same time (it can also tunnel SATA, Ethernet and others). The traffic is routed by a Thunderbolt switch at each end of the connection as can be seen in figure 2.5. This switch multiplexes and demultiplexes the traffic into PCIe and Mini DisplayPort. It also enables the devices to be daisy-chained (Device A is connected to device B which is connected to device C and so on). A typical scenario would be a Thunderbolt display (not pure Mini DisplayPort) with multiple Thunderbolt ports which allows a computer with only a single port to attach multiple devices. Since Thunderbolt is an interconnect which can tunnel PCIe and allows hotplugging and complex structures (daisy chaining etc.) and is easily available, it is an interesting research topic for us.

2.4 Virtualization support in PCIe

Virtualization is a technology that allows multiple OSs to run on a single machine by separating them into virtual machines. This is achieved by tricking the OSs into believing that they are



Borrowed from Apple's Thunderbolt Device Driver Guide [5]

Figure 2.5: Thunderbolt architecture

running on bare metal and trapping privileged instructions and emulating them. A virtual machine will then be unable to access the other virtual machines or the host OS or any applications running outside its own virtualized environment. To increase the performance of virtualization, newer CPUs have hardware support for virtualization which greatly increases performance by handling more of the isolation in dedicated hardware on CPU instead of in software.

Still, there are some things whose behaviour the host must emulate in software. This carries significant overhead and decreases performance for the guests. This applies for instance to IO, where a VM cannot be allowed direct access to a device because software running on the VM may use this access to break out of isolation. This is because devices typically have direct access to the main memory of the host. Software running in a guest can order a device to read and write to physical addresses owned by the host or other VMs. Malware that has infected a guest OS can use this to break out of the VM and infect other VMs and the host. In addition, the memory layout of the host will be different from the hosts due to each VM having its own address space separate from the host's. Since the devices work with the host's physical address space and not the VMs this would make device access impossible without significant support in the guest OS.

To have both isolation and more direct hardware access, newer computers have an IO Memory Management Unit (IOMMU). IOMMUs are explained in detail in the next section (2.4.1). This hardware has the ability to give the virtual machine more direct access to IO devices such as PCIe devices. The IOMMU assists in doing this while still keeping the necessary isolation between the virtual machine and its connected devices and the rest of the system. To do this, the IOMMU needs to translate the memory accesses from a device to the guest's actual physical memory location. The guest OS might also configure the devices' address spaces that conflicts with the address space of the host. To solve this, the IOMMU translates all memory accesses between the guest OS and the device. In addition, the device may also perform DMA to other devices, which also requires address translation. Furthermore, interrupt from the devices must also be redirected to the VM. All of this must be implemented in such a way that isolation between the guests and the host is not compromised.

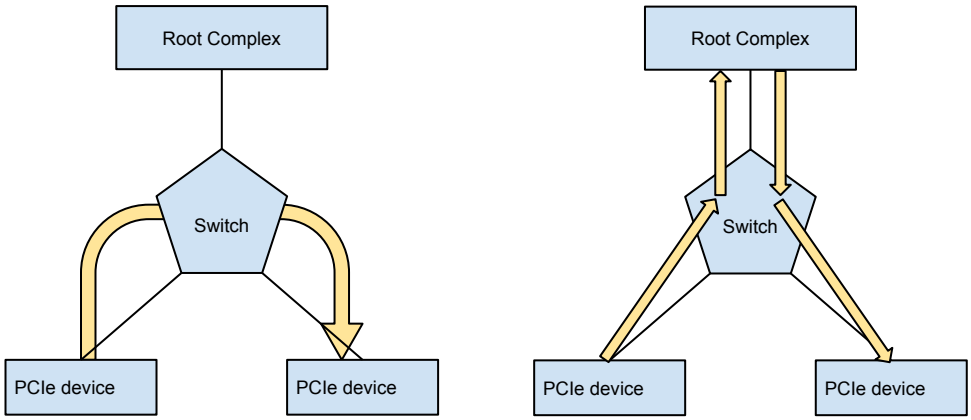


Figure 2.6: Left: normal peer to peer transaction through a PCIe switch. Right: transaction is routed through the root complex and to the IOMMU

2.4.1 IOMMU

In modern operating systems, processes have their own linear and isolated address spaces. This is enabled by the Memory Management Unit (MMU) of modern CPUs. It has the ability to create a virtual address space by translating the addresses used by the CPU before passing them on to the chipset and memory controller. The addresses are translated following the software defined *page table*. The translation works on a page size granularity (4KB by default on x86). Since the table would be very large if every single page must be present, the table is organized in multiple levels. Each entry in the top-level table points to another table. Also, each entry can be marked as invalid.

The I/O Memory Management Unit (IOMMU) is similar to the MMU, but is located between the chipset and the PCIe fabric (or other external buses). The most important feature of the IOMMU is the DMA remapper (DMAR). It translates the addresses of memory operations from the CPU to the PCIe fabric and from the PCIe fabric to RAM. Like with an MMU, access to unmapped addresses is denied. This provides isolation between the PCIe bus and the rest of the machine. An IOMMU can group PCIe devices into domains. Each domain has separate mappings and it's own address space similar to processes with MMUs. When an IOMMU is combined with normal CPU virtualization support, IOMMU domains can be overlapped with a virtual machines address space, which allows a VM to interact directly with a device and the device with the VM's virtual RAM. The access control features and domains of the IOMMU maintain isolation between the VMs and the host. When dealing with an IOMMU domain, there are multiple address spaces that need to be considered. The virtual memory address space, CPU physical address space, bus physical address space and the domain address space. An illustration of the address spaces can be seen in figure 4.3. In addition to isolating virtual machines, an IOMMU can be used to isolate device-driver pairs from the rest of the OS and other devices and drivers. This limits the potential damage an error (or malicious activity) that a device can do to the system. Not all CPUs have an IOMMU, but it is supported by more and more CPUs from Intel, AMD, ARM and IBM. Intel documents the features and workings of their IOMMU technology, VT-d, in their "Intel Virtualization Technology for Directed I/O" [6]. Since Intel CPUs are very common, we used the Intel IOMMU as a sample of what an IOMMU is capable of in the rest of the thesis.

As with most abstractions, DMA remapping brings a performance overhead. The memory translation tables are located in system memory in the same way as with an MMU. When a memory access passes the IOMMU, it must look up in the IOMMU's page table to translate the

access. This can be a performance issue since this can mean going through multiple levels of indirections in the page table. Although cache is present on the remapping engine to mitigate this. However, the cache depends on a good heuristic that predicts future accesses to lower the latency of checking the table in main memory. Peer to peer transactions are also heavily inhibited when remapped, since all transactions will be routed through the root complex instead of taking the shortest path. In external devices, this could involve a much lower bandwidth link which is unnecessary.

In an effort to increase performance of DMA remapping, PCI-SIG has developed the ATS specification [20] . It allows endpoint devices to translate the addresses themselves by having their own cache of their own most used remappings. This is implemented as an extension to the PCIe transaction level protocol. Each memory access will have a bit set if the address is already translated. This is done on a packet basis so the device may send one read that's pre translated and one that is not immediately after. This can be beneficial if the device knows what addresses will be accessed often. To learn the mapping of a page the ATC in the endpoint device can send a translation request to the translation agent in the root complex. The result of this request can be stored in the the devices own ATC and is valid until it receives an invalidate request from the root complex translation agent. The device is encouraged to locally cache addresses that will be accessed in the near future, especially if the access will be repeated. This allows for much more intelligent cache behaviour as the device itself often knows more about its own access pattern that the IOMMU in the root complex will. This could also greatly increase performance of peer to peer access since the traffic no longer needs to be routed through the root complex, see figure 2.6.

An IOMMU might be used to fulfil other purposes than to assist in virtualization. One example is to isolate a device and its driver from the rest of the system to prevent a malfunction or bug from affecting the rest of the system. Another example is to use an IOMMU to assist in scatter-gather operations to devices not capable of scatter-gather operations. This can be useful because a buffer in a virtual memory address space is seldom linear in the physical address space.

2.4.2 Single-Root IO Virtualization

An IOMMU allows a virtual machine to directly control a physical IO device in a safe and isolated manner. This VM will have full control over the device. This leads to increased performance compared to emulated hardware typically used in virtual machines. For instance, a VM typically has an emulated network card. When the VM OS sends a packet using this emulated hardware, the hypervisor will pass this packet through to the network stack in the host OS. Here the packet will get routed as any other packet, typically to a NIC and out of the host. If the VM is instead given a physical NIC, all packets will be sent out to the cable attached to the NIC. This removes a lot of the software overhead, and lowers the amount of times the data is copied before it is sent out on the cable. IBM has shown that a 10Gb SR-IOV capable Ethernet NIC can be almost saturated by a VM [15].

In addition, the VM will have full access to the bandwidth of this device since no other VM or the host can use it at the same time. It will not have to share it with other users. This is also the weak point of direct hardware assignment.

Much of the advantage of running virtual machines is the increased efficiency of having a few powerful machines instead of a lot of weak or under-utilized ones. Giving each guest direct access to its own IO device conflicts with this since each the host hardware now needs multiple IO devices. In addition, these IO device might no longer be fully utilized all the time. The additional number of devices will also use more power, need more room and have a higher

up-front cost.

To get both the performance of direct hardware control and the efficiency of device sharing, a new standard was created, Single Root IO Virtualization [18] (SR-IOV). SR-IOV solves this by allowing a single physical function (PF) to act as multiple virtual functions (VF). The device itself emulates the multiple VFs. The virtual devices are isolated from each other and the device itself shares its own physical resources across the VFs in a way that makes sense for the device type. For instance a network card can act as if an Ethernet switch connects the VFs and the outbound connection. A storage device can be partitioned or provide concurrent access to the same storage space, but with some synchronization features. The VFs can be directly assigned to a virtual machine in the same way as any other function. The virtual devices can have some features disabled to prevent them from being able to negatively affect the operations of other VFs or the physical device. Features of the device that can affect all VFs and the physical funtion are typically only available for the PF. For some devices, potentially unsafe functionalities are needed by the VFs for correct operation, or desired by users. For instance, a NIC might want to set its own MAC address, but it would be unfortunate if one VF, intentionally or unintentionally, set to the same MAC address as another VF or the PF. Potentially, the device could deny this, but this might not be practical for all purposes and it's not very flexible. In Intel's NICs with SR-IOV, there is a communication channel between the drivers for the VFs on the VMs and the driver for the PF on the host. This allows the PF driver to control the requests by the VF drivers in a safe manner that can be defined by software. Since the VMs are isolated from each other and the host, there is no guaranteed way for them to communicate. A hypervisor could implement support for a communication channel between the driver, but to avoid having to support various hypervisors, the drivers will communicate with the help of the device itself. A mailbox and doorbell mechanism on the NIC itself is used by the VF drivers and the PF drivers to communicate [13] and allow a VM to perform potentially unsafe operations under the control of the host.

A device that implements SR-IOV has the SR-IOV capability structure in it's configuration space. The host can control the SR-IOV related features of the device by setting the desired values in the SR-IOV capability. Before the enable bit is set in this structure, no virtual functions will be present. Before enabling this, the host can set the number of desired virtual functions. There is a limit to the number of VF a given device support which it reports through a register in its SR-IOV capability structure. Various other parameters dealing with the BARs for the VFs are also present.

Since the number of function numbers for a given device is fairly limited, Alternative Routing Interpretation (ARI) was created which allows for more functions for a single device. It raises the number of functions to 255, but if the number of VFs exceed this, an additional bus number can be used.

In the same way as hot add, increasing the number of VFs, or enabling it can require more resources than available in the upstream bridge. This will require the OS to expand these resources. This includes memory resources and bus numbers. IO space is not needed as VF cannot have IO BARs. To avoid the difficulties with this, the platform firmware should also be SR-IOV aware. If the firmware is SR-IOV aware, it can reserve space for the additional devices at boot time, for instance by using the maximum number of VFs for a given device. Such preallocation is much simpler as it can be performed before any use of other PCI devices starts.

2.4.3 Multi-Root IO Virtualization

PCI and PCIe are strictly for use by a single host machine. Multi-Root IO virtualization (MR-IOV) [17] is a PCI-SIG developed standard for allowing multiple hosts to be connected to the same PCIe fabric. MR-IOV allows the connected hosts to have devices dynamically assigned to them and to concurrently share devices like VMs can with SR-IOV. Each host connected to the MR-IOV fabric has their own own virtual hierarchy (VH). The VH is consistent with the traditional PCI model as there will be only one root complex in each VH. Each host only sees its own VH and it does not need to know that it's part of a MR-IOV fabric as the VH operates identically to standard PCIe. For MR-IOV to be utilized, new PCIe switches must be used which are called Multi Root Aware (MRA) switches. Normal PCIe switches can be present in a MR-IOV fabric, but a VH cannot span such switches, only MRA switches. A figure showing a MRA switch with two VHs can be seen on the right side of figure 2.7.

While the MRA switch can assign any single device to a VH, normal devices cannot be used concurrently by more than one host, even if it's SR-IOV capable. However, specially designed MRA capable endpoint devices can be concurrently used by multiple hosts in the same way as SR-IOV devices can be used concurrently by multiple VMs. The device will present individual configuration spaces to the different hosts similar to SR-IOV. A device can implement both SR-IOV and MR-IOV to allow it be used by VMs in different hosts at the same time. If it does, each host has control over the SR-IOV capability of its own virtual device.

In the MR-IOV hierarchy a host is assigned the Multi-Root PCI Manager (MR PCIM) role. It is responsible for setting up the MRA switch and the other hosts VH's. This host will be the only host aware / need to be aware of the fact that this is a MR-IOV switch. The MR PCIM will scan the switch and all connected devices. When its done it will assign the devices to a VH. The final step is connecting a host to its VH. The other host will then discover their own VH and scan it for devices, set the devices and operate as normal.

MRA switches have two distinct forms of hotplug: the normal physical hotplug and the virtual hotplug. The virtual hotplug is the one seen by the non-PCIM hosts. When such a host interacts with this capability, the MR PCIM is notified. The MR PCIM acts as a layer between the hotplugging controlled by the other hosts and the physical hotplug control. When a device is removed, the event is propagated by the MR PCIM to the virtual hotplug interface of all VHs with the affected devices. The same applies to other events and statuses such as button presses. The MR PCIM also has the ability to reassign devices from one VH to another. This is presented to the other hosts as a hotplug event in the virtual host.

MR-IOV is can be very useful in solving the problems we introduced in this thesis as it allows great flexibility by allowing for dynamic reassignment as well as concurrent use of devices. With the exception of the MR PCIM, no modification is required to the hosts hardware or software which is useful for easy adoption. Unfortunatly, the MRA switches require new chips to be designed. The same applies to the MRA devices which are the only devices than can be shared by multiple VH at the same time. At the moment however, we have not found any MR-IOV capable switch available for sale and the same also applies to MRA devices. As it stands MR-IOV cannot be used for this reason. Considering that the MR-IOV standard was finished in 2008, its lack availability is not a good sign for its future adoption either.

NextIO

NextIO was a start-up developing IO virtualization products. Their product was a competitor to the traditional top-of-the-rack network switch. Their main product was an external cabinet with room for PCIe devices. This cabinet was connected to multiple hosts with a 10Gb/s PCIe

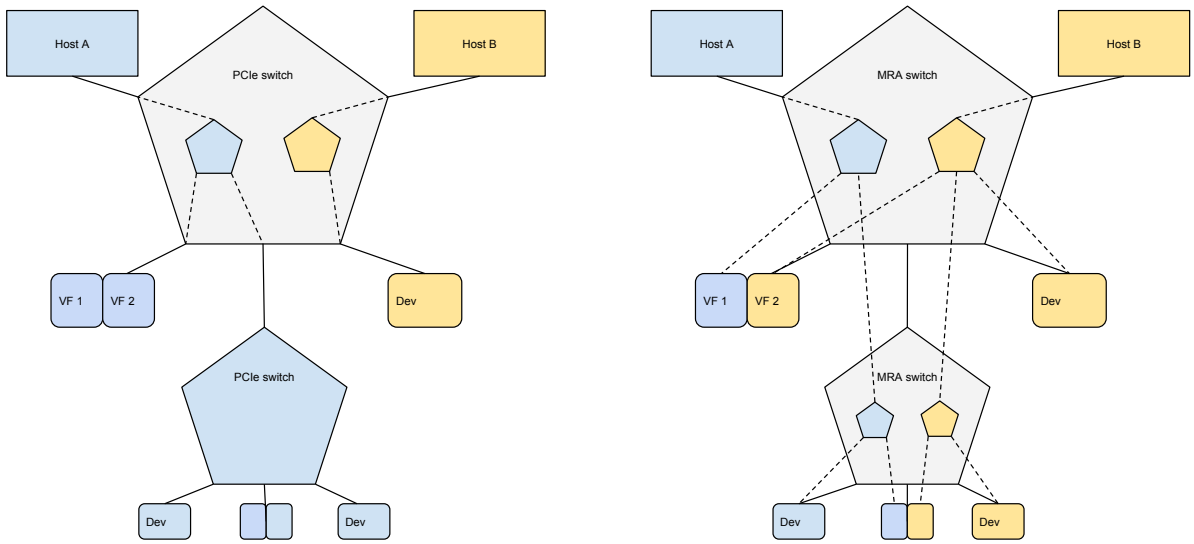


Figure 2.7: Left: A PCIe switch with two partitions. Each host sees its own fabric and switch. Right: An MRA switch. More finely grained device assignments.

cables. The cabinet was fitted with a 10 or 40 Gb/s Ethernet card or a Fiber Channel card. Their product allowed all of the hosts to access the single NIC at the same time, sharing its bandwidth. In addition, the cabinet supported other PCIe devices, but as far as we know, they could not be shared, but only be assigned to a single host. Although the details are a little vague, we believe that the cabinet was an implementation of a PCIe MRA switch. This fits well with the Ethernet and Fiber Channel cards that could be shared by the hosts, but not other cards. We believe these cards, delivered by NextIO themselves, where MRA cards, capable of being controlled by multiple hosts. If so, NextIO was one of the few vendors that produced MR-IOV products, but in 2013, NextIO went bankrupt.

2.5 PCIe switches with support for partitioning

Some PCIe switches have support for so-called partitioning. Partitioning is not part of the PCIe specification and is vendor dependent. It is a feature that allows multiple hosts to be connected to a single physical switch somewhat like MRA switches. The switch will partition the traffic in such a way that the each host sees its own virtual switch and PCIe fabric. An illustration of a partitioned switch compared MRA switches can be seen in figure 2.7. Each partition have a single root complex and zero or more devices, and each port of the physical switch is assigned to a partition. Unlike a MRA (MR-IOV) switch, the different root complexes cannot share devices or overlap in any way. The different partitions can be connected to one another with NTBs, however, which can provide communication between the partitions, (see section 2.7). These switches are inferior to true MRA switches, but are readily available. Our findings so far indicate that the partitioning is limited to a single switch, and cannot span multiple switches like VHs in MRA switches. This limits the granularity of the assignments to the entire sub tree of downstream port of the switch. In the case of an external switch and an external expansion chassis, a single host would be in control of the entire chassis. If an expansion chassis has an internal switch that supports partitioning, it is possible to assign individual devices to the hosts, but each host must then be connected to the chassis by a separate link. It is perhaps possible for two switches to be connected and have the virtual switch span the physical switches or have something like the VH in a MRA switch. Combining multiple such switches, one "main" switch

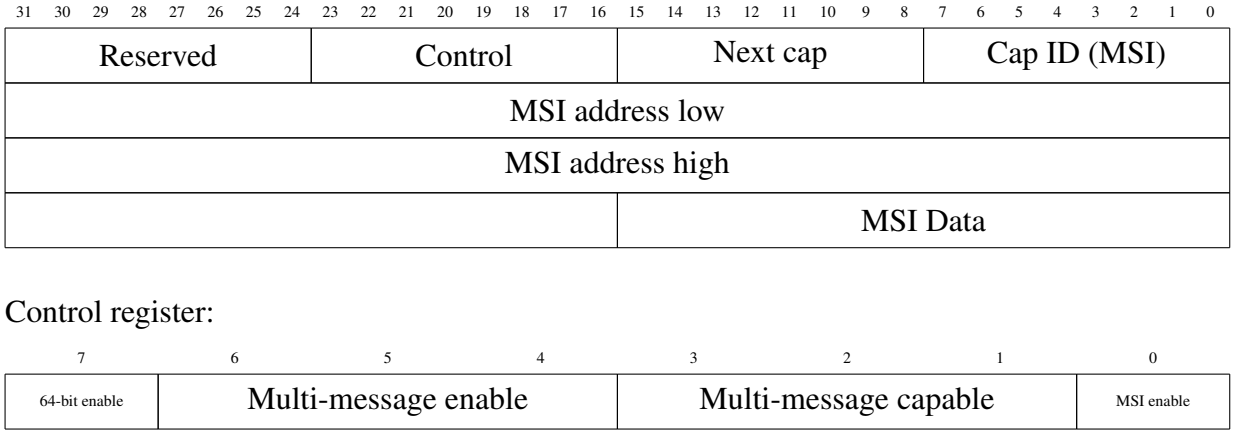


Figure 2.8: The MSI capability structure

as well as one in each expansion chassis, could allow each host to be assigned individual devices for separate chassis.

Possibly, a switch port could be reassigned from one partition to another on the fly, without interfering with the traffic in unchanged ports. If so, it would be possible to use such a switch to allow PCIe devices to be dynamically assigned to different hosts in a cluster. In theory, this should look like and behave as a physical hotplug event. If the switch also reports these events as hotplug events, there would not necessarily be any need for modification to drivers or the OS to support it. A master's thesis from MIT titled "PCI Express Multi-Root Switch Reconfiguration During System Operation" [24] tested the repartitioning of a PLX switch while the hosts where live. In their test setup, two hosts and two NICs were connected to the switch. In one experiment, one of the NICs was moved from one partition to the other. They reported no ill effects from this migration. Unfortunately, measurements of the possible effects of this seem to be lacking. Possibly the operation of the non-moved device could be affected. It would also have been interesting to see a running bandwidth benchmark from one NIC while the switch was being repartitioned. This should show if the repartitioning had any negative negative effects on performance. There is also little discussion on the time it takes to migrate a device, but presumably it is fast.

2.6 Message Signalled Interrupts

In traditional PCI, a device signalled an interrupt by driving a dedicated interrupt pin. When the CPU received the interrupt, it polled all devices on the same interrupt line to find the sender. There was no way to know what device asserted the interrupt without this because multiple devices shared the same interrupt line. This same mechanism still exists in PCIe, but while it is supported, it is considered legacy. The preferred way to raise interrupts in PCIe is with Message Signalled Interrupts (MSI). Support for this is required for all PCIe devices that can generate interrupts [20]. The MSI is specified in the legacy PCI specification [19] with some modification for PCIe in the PCIe specification [20]. When a device issues an MSI interrupt, it will do a normal write to a given address. This is received by the chipset which generates a interrupt to the CPU. On x86 a write to `0xfeeXXXXX` will trigger an interrupt.

A device that can generate MSI interrupts will have the MSI capability that can be seen in figure 2.6, in its configuration space. The address that the device writes to, to generate the interrupt is set by configuring the MSI capability structure. In addition a data field specifies what

the device will write. The combination of address and data makes up the *interrupt vector*. The device has control over the lowest bits in the data so that it can generate 32 unique interrupts. Some devices also have another capability called MSI-X or MSI extended, which gives the device up to 2048 different interrupt vectors. In addition, drivers can mask our individual interrupts with MSI-X. This requires a much larger amount of configuration than a single capability structure. Some of the MSI-X configuration is therefore mapped into a BAR area instead, which BAR, and the offset inside that bar is stored in the MSI-X capability structure.

2.7 Non-Transparent Bridges

PCIe was designed as a single host fabric with a single root complex. It was however discovered that it would be useful to use the strengths of PCIe to create an interconnect solution. One such solution is a Non Transparent Bridge (NTB) that allows multiple PCIe devices to communicate with the help of PCIe technology, because this gives the hosts a link with very high bandwidth as well as extremely low latency. This is done by allowing the hosts to read and write to parts of each others' memory, creating a shared memory architecture. Since the NTB can let any memory operation through it can perform operations not only to RAM, but to other devices as well. NTB devices are not standardized, but all have similar capabilities.

Despite its name, the non-transparent bridge is not a PCI bridge, but an endpoint device with BAR areas like any other device. However, memory operations to the BAR areas are forwarded across the NTB link to the other side where it's emitted by the local NTB. Since the hosts do not share address spaces, the NTB provided a simple address translation. The translation is done by a simple single-level page-table-like mechanism: The BAR area is divided into around 20 equally sized pages and each page can be directed into any part of the other hosts memory. This is done by replacing part of the incoming address with a per-page offset into the other host's address space. This will typically allow an NTB equipped host to access about 20, 32MB memory segments in the other host. Each page can be translated to any part of the other host's address range, but cannot be fragmented.

New Intel Xeon CPUs has built-in support for NTBs [11] which makes NTBs very widespread.

2.7.1 Dolphin NTB Software

Dolphin Interconnect Solutions (Dolphin) sells NTB PCIe devices that are bundled with a software suite that allows user applications to use the shared memory and Remote DMA (RDMA) capabilities provided by the NTB. Included in the software suite are the low-level drivers, an application layer API called SISCI and a TCP/IP implementation. The SISCI API [3] is the best way to utilize the benefits of the NTB and is the only way for an application to use the shared memory capability of the NTB device. Using SISCI however, can require a program to be redesigned with SISCI in mind. The network implementation as well as a MPI implementation allows existing programs to take advantage of the performance benefits of the NTB without modifications. In addition to the application level APIs, there is also a kernel level API on which SISCI is built called GENIF.

Supersockets and TCP/IP implementation

Dolphin has implemented two separate network implementations for unmodified programs to take advantage of the performance benefits of an NTB that are not designed to use shared

memory or RDMA. The first is a virtual network device that the OS uses as any other network interface, but instead of sending the traffic with Ethernet, it is transferred using RDMA to the destination host. This increases performance over normal Ethernet, however, since the OS handles it like a network interface, it will still split the data into packages, use TCP and other mechanisms which are not needed for RDMA. This, including various other mechanisms that are needed or are beneficial on a packet based network can be detrimental to the performance of RDMA. The second implementation, called Supersockets, allows the packet overhead to be bypassed and performance to get closer to the performance of a native RDMA API. Supersockets replaces the standard Berkley interface provided by the OS with an implementation that uses RDMA and allows the data to bypass the OS and be passed directly to the other host using the NTB hardware. To use Supersockets, the dynamic library is linked in with the application at runtime and no modification is needed to the source code of the application.

SISCI API

The SISCI API is built upon the concept of memory segments that the user can allocate by interacting with the SISCI API. A memory segment can be set as shared memory or data can be transferred from a segment on one host to a segment on another host using RDMA. Shared memory segments are accessed by creating and connecting a remote segment by the other host. Memory accesses to a remote segment are passed across the NTB and directly into the RAM of the remote host without the need for the application to treat it in any other way than local memory buffers.

An advanced feature of the SISCI API is its ability to create mappings not just to SISCI allocated memory buffers, but to arbitrary physical addresses. This is done by creating an empty SISCI segment and binding it to a physical address using the `attachPhysicalSegment` function. If this segment is made available for remote hosts, any access to this segment will access the attached physical address. Since a segment can be bound to arbitrary addresses, SISCI can be used not only to access a process' memory, but kernel memory and most importantly, MMIO address ranges.

Kernel level API

SISCI itself is implemented on top of a kernel level API called GENIF which is again implemented on top of the lower level device drivers for the various devices supported by Dolphin. GENIF is similar to to SISCI and many API functions in SISCI have a one-to-one correspondence to a function in GENIF, the primary difference being that SISCI adds support for user level processes and in some cases provides additional error handling. GENIF API users can interact with a SISCI application, but must be aware of the additional functionalities and any abstractions provided by the SISCI API and copy them.

2.8 Page Attribute Table

Page Attribute Table or PAT is a feature of the virtual memory page table in the x86 architecture [12] that allows attributes to be set on each page enabling or disabling certain features. One such feature is caching which is very desirable to enable for virtual memory mapped to RAM. Caching however, should normally be disabled for memory-mapped IO as MMIO can have side effects and the order of the operations can be important. MMIO is also not cache-coherent, so any changes to MMIO registers by a device will not be reflected in the cache.

Another attribute that can be set is write combining (WC) which is a lighter form of caching. When it is enabled, the CPU can merge smaller reads or writes to this area into larger memory operations. This allows for great speed increases when more than a trivial amount of data is written to a MMIO mapped area. It works by having a small cache that buffer up writes or reads so that multiple small operations are merged into one large before the CPU issues the memory operations to the bus. This can cause memory write operations to be delayed while the CPU waits for additional writes to the same area.

It is possible to map the same physical address to multiple virtual addresses which the Intel Software Developers manual [12] refers to this as "aliasing". When a physical address is aliased, it is possible for it to have multiple conflicting attributes which is why Intel recommends avoiding aliasing:

> The PAT allows any memory type to be specified in the page tables, and therefore it is possible to have a single physical page mapped to two or more different linear addresses, each with different memory types. Intel does not support this practice because it may lead to undefined operations that can result in a system failure. In particular, a WC page must never be aliased to a cacheable page because WC writes may not check the processor caches.

2.9 Application level distribution

Most computers have support for at least two GPUs, so a GPU-bound application can gain performance by adding additional GPUs to the computer. However, only expensive high-end computers have room for more than a handful of GPUs so scaling an application's performance beyond this point can be difficult. Computer vendors like Dell and HP deliver rack servers with room for additional GPUs or external cabinets dedicated to GPUs that can be attached to computers and work as local GPUs. This can however be expensive and in addition, all the GPUs might not always be needed in a single computer. A more flexible and less expensive solution is to distribute the program across multiple servers, but this requires large changes to the application if its not designed with distribution in mind. Unless the problem is trivially paralyzable, the key to achieving high performance in a distributed application is efficient communication between the nodes, especially as more and more nodes are added. This can require expensive hardware and careful design of the program. Often a high performance communication protocol like MPI is used, frequently in combination with dedicated high bandwidth links like Infiniband.

2.10 Related work

MR-IOV requires new hardware switches and MRA devices for sharing of devices, but since the availability of MRA switches and endpoint devices is low, multiple vendors have tried to create MR-IOV like solutions that can be used without MRA devices. Most of the other solutions attempt to provide similar functionality to MR-IOV, but focuses on using existing devices and computers.

Xsigio was an early contender, developing a network virtualization solution that used Infiniband or Ethernet and virtualized network cards. Xsigio was bought by Oracle in 2012 and integrated into their virtualization product. A more relevant technology was developed by Aprius. They developed a hardware device that tunneled PCIe traffic over Ethernet that made for a very flexible solution that could utilize existing technologies and Ethernet gear. Unfortunatly,

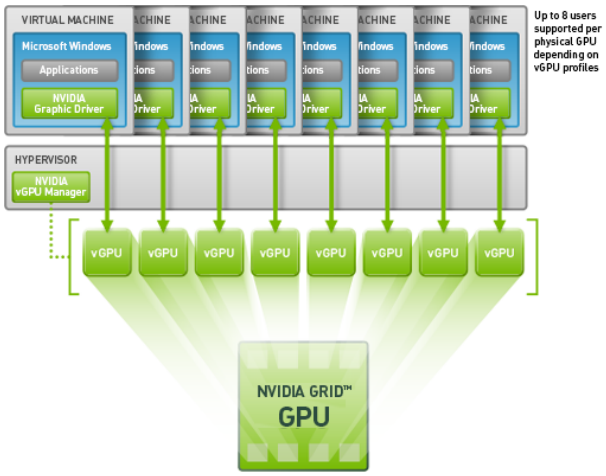


Figure 2.9: Nvidia GRID promotion material showing virtual GPUs

Aprius has seemingly gone bankrupt or been bought out, with their website now owned by an unrelated company.

2.10.1 Nvidia GRID

Nvidia has developed their own solution for virtualizing Nvidia GPUs, "NVIDIA GRID™ vGPU™" [2]. Their solutions allows a single GPU to be shared by multiple VMs with full 3D acceleration and CUDA. They support multiple hypervisors including VMWare, Citrix, Nice and Microsoft RemoteFX. While its sounds like something that could be acheived with SR-IOV, this is not mentioned anywhere in their materials. In addition, there is no mention of support for Nvidia GRID vGPU in any Open Source hypervisors. This would suggest that it's a propritary solution specific to Nvidia.

2.10.2 Micron IO virtualization

Micron's IO virtualization comes from the startup Virtensys, which Micron bought after Virtensys got into financial troubles. Their IO virtualization solution supposedly requires no custom software or devices and is fully transparent to the server OS. The closest we got to technical details about Microns IO virtualization product is the following quote comparing their solution to MR-IOV which confirms that their product does not implement MR-IOV, but is an alternative.

> With Micron's IOV (and PCI MR-IOV), the PCI switch (IOVE) has been enhanced to allow multiple servers to connect to the switch and for the I/O devices to be shared across many servers. The sharing in MR-IOV requires the I/O devices to be modified. However, the sharing functionality in Micron's IOV has been built into the switch fabric in the form of a virtualization proxy controller (VPC), which is a Micron hardware device that works with the switch to virtualize multiple standard PCIe cards.

Our best guess is that their "VPC" emulates a physical network card and multiplexes the traffic in hardware to the real network card. This probably limits their product to a few supported devices, altough this is not clear. They claim to at least support Ethernet, fiber channel, some harddrive controller and SSDs.

2.10.3 Sharing SR-IOV devices with multiple hosts

SR-IOV is designed to allow VMs on a single computer to share a single hardware device and MR-IOV expands this feature to multiple computers. Since MR-IOV is not really available, there has been some effort to use SR-IOV devices as MR-IOV devices. The paper "Multi-root Share of Single-Root I/O Virtualization (SR-IOV) Compliant PCI Express Device" demonstrates that a SR-IOV device can be shared across multiple physical hosts not just VMs [21]. To connect multiple hosts to the same device, they developed a custom FPGA device that translated MMIO address spaces similar to an NTB. When sharing a SR-IOV NIC, performance reached 99% of that of the same device attached locally.

2.10.4 Ladon

Ladon [22] is a system developed by researchers at Stony Brook University aimed at data centre settings. The goal is to decouple the devices from the host and put them into a pool shared by multiple hosts. Tu et al claim that this offers greater flexibility in assigning devices to the VMs. This shared device infrastructure can be enabled by the PCIe technology MR-IOV, however, as the paper recognizes, this hardware is not readily available and so, attempt to implement a software alternative. Ladon builds on virtualization technologies, specifically, it extends the ability of hypervisors to assign a physical IO device to a virtual machine and uses SR-IOV devices to enable sharing. The devices, including SR-IOV devices are placed in a device pool accesible to VMs on multiple physical hosts.

The authors makes multiple claims for how this can increase performance per dollar and reduce power usage. One example given is a traditional VM setup where multiple hosts each have a 10Gb NIC. This NIC is shared by the VMs in on each host and the NICs are connected through an Ethernet switch. The paper makes the claim that all of the NICs and the switch can be replaced by a single shared NIC and a PCIe network between the hosts. The device itself will be able to ensure fair sharing of the bandwidth between the VMs. When multiple NICs are replaces with a single shared NIC, the total bandwidth available is lower than the total with multiple NICs. However, if the switch is connected to the rest of the network with a single 10Gb link, the total outbound bandwidth is unaffected. Tu et al also make the claim that this has a lower power usage than having multiple NICs. It is also easy to see that when the NICs are shared and a single NIC is added, the additional bandwidth benefits all VMs instead of only the VMs on a single host. Finally, the paper shows that the performance overhead of accessing the devices in the pool versus local devices for a VM is low (5%).

Ladon uses a PCIe interconnect with Non-Transparent Bridges to allow multiple hosts to be interconnected. All devices are placed in a central host called the Management Host (MH). Each host that should have the ability to access the devices in the pool must be connected to the MH with an NTB.

The paper recognizes that others have experimented with MR-IOV and NTB as a substitute for the lack of MR-IOV hardware. They however claim that their interrupt delivery mechanism is novel: While others have used sideband communication to deliver interrupts to the VM, Ladon allows the MSI interrupts to pass directly through the NTB to the active VM. The paper also claims that the VMs are securely isolated from the hosts and other VMs. Since Ladon allows direct control of hardware devices, a VM can use an assigned device as an attack vector by ordering it to read and write to memory accesible by the device but not the VM that controls it. For instance the device can be told to write a network packet to an address located in the RAM of the physical host. To isolate the devices assigned to a VM from the other VMs and their devices as well as the physical hosts, Ladon uses a combination of the IOMMUs in the

VM host and in the MH as well as the control over memory mapping provided by the NTBs.

Ladon provides a modified Linux kernel on both the physical host as well as on the VMs. The physical host kernel is modified to intercept the VMs' accesses to the virtual device. This is used to set up the virtual device and assign a device from the pool. In the VM, part of the kernel is modified so that the DMA addresses the driver gives to the device are correct. This is needed because the address of a buffer in the VM's RAM is not the same from the driver side as from the device side both because the driver is running in a VM with its own address space and because the device is in a physically different host which also has its own address space. Because it's the software in the VM that gives the address to the device, the software is required to know about the address space of the MH and how the MH's address space maps to it's own address space. According to the paper, the kernel modification allows the devices in the pool to be accessed by the VMs without modifying the device drivers.

For the device to be able to access the VM's RAM, the VM's virtual memory needs to be mapped into the address space of the device and translated. In Ladon, 3-4 GB or the entire VM RAM is mapped and available for the device. Due to the way NTBs work, this requires the BARs on the NTB to be at least the size of all VMs' RAM combined. The developers report problems with the BIOS of the MH with such a large BAR and their solution was to manually assign this BAR after boot.

We have identified multiple limitations with Ladon. First of all, we believe that having all devices in a central machine is a weakness as this is a single point of failure for all hosts. This is, however, improved in later versions (Merlin) by having a failover scheme for the MH [23]. We are, however, not convinced by the effectiveness of this mechanism because we believe that a failure of the MH would still interrupt the operation of all VMs using a device in the pool. For instance, if the MH were to reboot, all of its devices would be reset and all communications between drivers and the devices would be interrupted. The central pool also means that all device IO from a host must cross a single PCIe link, which has 8 lanes in Ladon, which means that the IO device bandwidth available for all VMs on a single host is shared. In addition to being shared, the bandwidth provided by a single 8 lane link is lower than what some devices use, a single GPU for instance can have 16 lanes alone. The use of a single x8 link will give lower performance than local devices that have dedicated links and with heavy IO traffic and multiple VMs this can be significant. If higher combined bandwidth was available multiple 10Gb NICs could be shared by the VMs, but with only a single 8-lane link per host, the total network bandwidth for all VMs on a host is limited.

While Tu et al's performance benchmarks show good speeds for device-initiated DMA, their performance for CPU-initiated DMA is disappointing. It is at least 30 times slower than device initiated DMA, which is probably due to the lack of caching and write-combining. The paper gives no reason for the lack of these features. The lack of other devices in their tests and benchmarks is also disconcerting. We fear that this is due to a technical limitation of Ladon and that this may indicate that the modifications to the kernels are device-specific. It is also possible that Ladon is limited to SR-IOV devices. If so, it is unfortunate, because there is only a limited number of SR-IOV devices and non-SR-IOV devices are far more common. Ladon supposedly supports using the devices directly from the host machine and not only in VMs, this is however quickly glanced over and it is not stated if this is possible without driver modification. Finally, we believe that the various software modifications are a limitation. The modifications are not limited to a single location, but both the VM OS as well as the hypervisor and host kernel need to be modified. With guest VMs with closed source kernels, Ladon cannot be implemented. In addition, a complex central manager is implemented to set up the shared devices.

2.11 Chapter summary

We know of no easily available solution for Multi-Root PCIe: No vendor produces MR-IOV products and other proprietary solutions seems to be limited or unavailable. Ladon seems to be a promising alternative to MR-IOV, but it is heavily focused on virtualization, has multiple flaws and its not, to our knowledge, available as a product or open source.

Since NTB devices forwards memory operations, it can forward most of the traffic to a device including MMIO, DMA and MSI interrupts. The memory accesses through an NTB is directly transferred to the target on the other side and does not need to involve the CPU which should offer very high performance. Unlike MR-IOV, NTB devices are easily available. Combined with the correct software and SR-IOV, an MR-IOV-like solution can be created with existing hardware.

Chapter 3

Linux Kernel

When working on this thesis, we wanted to be able to see the source code of the OS we developed for. This combined with the ability to modify the OS itself is very useful when working at the low level required for implementing remote PCIe device access. Linux is by far the most widely used Open Source kernel and it has good support for PCIe including hotplugging and SR-IOV, so it was a natural choice

3.1 Device and driver subsystem

The Linux kernel supports all kinds of different devices including USB, Firewire, and PCI, all of which are part of the device subsystem. The central parts of the device subsystem are the devices, which are internally represented by a device structure as seen in code snippet 3.1, and the device drivers. Each device has a parent device, which in bus based devices such as PCI is a bus device, or it is a top level device with no parent.

Each device can be controlled by a single device driver which is *bound* to the device. The currently bound driver is stored in the member `driver` in the `device` structure. Device drivers in Linux have a filter list that match devices they support that the Linux kernel uses to bind the device drivers to the individual devices as they are discovered. Before the the driver is successfully bound, the driver's `probe` function is called. In this function, the driver can further examine the device for support and set up the necessary structures, mappings and interrupts necessary for the correct operation. If a driver return success for the probe function, the driver is bound to the device and now has control over the device. The various device subsystems have their own more specific device structures, device drivers and API functions. An inheritance-like model allows these specialized device types to be handled by the generic device subsystem.

3.2 PCI subsystem

3.2.1 PCI device structure

The PCI subsystem in the Linux kernel scans for PCI devices when the kernel boots or every time a hotplug event is signalled. It performs the scan using configuration requests and when a valid configuration is found, the corresponding device is added. When a endpoint device is discovered a `struct pci_dev`, which can be seen in code snippet 3.2, is created. This structure represents a single PCI device and contains all relevant information about it. After

```
struct device {
    struct device            *parent;
    struct kobject            kobj;
    const struct device_type *type;
    struct bus_type          *bus;      /* type of bus device is on */
    struct device_driver *driver; /* which driver has allocated this device
        */
    void                  *driver_data;  /* Driver data, set and get with
                                            * dev_set/get_drvdata */
    int                    numa_node;    /* NUMA node this device is close
       to */
    u64                   *dma_mask; /* dma mask (if dma'able device) */
    struct acpi_dev_node   acpi_node; /* associated ACPI device node */
    struct class          *class;
    void                 (*release)(struct device *dev);
    struct iommu_group    *iommu_group;
};
```

Code snippet 3.1: Cut down version of the device structure [1]

the device initialization is done, the configuration space of the device is mostly left alone. Drivers and kernel code is not supposed to read values from the configuration space itself if it is stored in the `pci_dev` structure, because the values might not be identical if a "quirk" has been applied to the device due to some device defect. After the device has been discovered and initialized, it is given to the generic device subsystem where a matching driver will be found. All `pci_dev` structures are connected to the `pci_bus` structure of the upstream bridge the device is connected to unless they are connected to the root of the PCIe fabric.

The PCI device structure contains the generic `device` structure which allows the generic device subsystem to handle the PCI device structure by passing a pointer to the contained `device` instead of a `pci_dev` pointer. In the same way, when a `device` pointer is a pointer to the `device` contained in a `pci_dev`, a pointer to the `pci_dev` can be acquired by using the `container_of` macro. This is analogous to casting in object oriented programming.

3.2.2 Configuration space access

Different platforms have different ways to access the PCI configuration space. In older PCI based x86 systems, access is done using the port IO addresses `0xCF8` and `0xCFC`, but on PCIe systems, the extended configuration space must be accessed with MMIO. The address of the MMIO area of configuration space is implementation defined and the firmware informs the OS of the location using ACPI. The Linux kernel also support other platforms with various other configuration space access methods. To abstract away the differences, all accesses to configuration space are done using a family of functions including `pci_bus_read_config_dword` and `pci_bus_write_config_byte`. Since the method for performing configuration space accesses can be different from one PCI bus to another on the same computer, the correct way to access the configuration space behind a given bus is stored in the bus device structure. This is implemented as a structure with a set of function pointers which the configuration space access functions calls to perform the required low-level functionalities.

```
/*
 * The pci_dev structure is used to describe PCI devices.
 */
struct pci_dev {
    struct list_head bus_list;  /* node in per-bus list */
    struct pci_bus  *bus;       /* bus this device is on */
    struct pci_bus  *subordinate;   /* bus this device bridges to */

    void        *sysdata;   /* hook for sys-specific extension */
    struct pci_slot *slot;      /* Physical slot this device is in */

    unsigned int    devfn;      /* encoded device & function index */
    unsigned short  vendor;
    unsigned short  device;
    unsigned short  subsystem_vendor;
    unsigned short  subsystem_device;
    struct pci_driver *driver;  /* which driver has allocated this device */
    u64     dma_mask;   /* Mask of the bits of bus address this
                       device implements.  Normally this is
                       0xffffffff.  You only need to change
                       this if your device has broken DMA
                       or supports 64-bit transfers.  */
    struct  device  dev;        /* Generic device interface */

    int     cfg_size;   /* Size of configuration space */

    /*
     * Instead of touching interrupt line and base address registers
     * directly, use the values stored here. They might be different!
     */
    unsigned int    irq;
    struct resource resource[DEVICE_COUNT_RESOURCE]; /* I/O and memory regions + expansion ROMs */
    /* keep track of device state */
    unsigned int    is_added:1;
    unsigned int    is_busmaster:1; /* device is busmaster */
    unsigned int    no_msi:1;   /* device may not use msi */
    unsigned int    msi_enabled:1;
    unsigned int    msix_enabled:1;
    unsigned int    is_physfn:1;
    unsigned int    is_virtfn:1;
    unsigned int    reset_fn:1;
    unsigned int    is_hotplug_bridge:1;
    union {
        struct pci_sriov *sriov;    /* SR-IOV capability related */
        struct pci_dev *physfn; /* the PF this VF is associated with */
    };
    struct pci_ats  *ats;   /* Address Translation Service */
    phys_addr_t rom; /* Physical address of ROM if it's not from the BAR */
    size_t romlen; /* Length of ROM if it's not from the BAR */
    char *driver_override; /* Driver name to force a match */
};
```

Code snippet 3.2: Greatly cut down version of pci_dev structure in the Linux kernel [1]

3.2.3 Driver interface

Drivers in the Linux kernel communicate with the devices they control using an interface that is different depending on the bus and device type. For PCIe devices, the main interfaces are MMIO, interrupts and DMA.

Interrupts

A device driver receives the interrupts from a device it controls by setting up an interrupt handler which is a function that is called when the OS receives a given interrupt. For the OS to know what interrupt it should send to which driver, the driver needs to tell the OS. The driver registers an interrupt handler by calling the `request_irq` function with the device's interrupt number as well as a function pointer to the handler and multiple other arguments.

Depending on the device type, the interrupt number may be fixed or assignable and may or may not be known. In PCI, the interrupt number for a device is stored in the `pci_dev` structure and is assigned by the PCI subsystem. Since there are a limited number of interrupt numbers, some devices may have to share an interrupt number, but with the introduction of MSI, shared interrupts are no longer an issue. When using legacy interrupts and a shared interrupt is received, the kernel will need to figure our which device fired the interrupt by probing the devices. This will allow the correct interrupt handler to be called.

An interrupt hander will often have to communicate with the device, for instance the driver can use MMIO to read the cause of the interrupt from the device. This is especially true for more complex devices where there can be multiple causes for the interrupt. MSI allows the device to generate different interrupts for different events which allows a driver to have multiple different interrupt handlers, one for each possible interrupt. One example is a device with two ring buffers, one for sending and one for receiving and the device sends two different interrupts indicating if a command was received or sent. This saves the driver the need for probing the device to discover why the interrupt was generated.

The driver enables MSI by calling `pci_enable_msi_range` and in addition to the device, a minimum and maximum number of interrupts the driver requests is given. The function will return the number of interrupts it was able to allocate and the driver then calls `request_irq` on all of them, presumably with different handlers.

Most devices that support MSI also support legacy interrupts and since MSI is supperior to legacy interrupts, device drivers for devices with MSI support use MSI instead of legacy interrupts [14]. However, some platforms may not support MSI so drivers sometimes support both in order to provide greater compatibility. The same applies to MSI-X as a device can be capable of MSI-X, regular MSI as well as legacy interrupts. All of them have different capabilities and features and a device that supports all will probably work best with the most capable, MSI-X, and have simpler interrupt modes as a fallback. Depending on the device driver, it might not support the less advanced interrupts mechanisms.

MMIO - ioremap

Memory Mapped IO (MMIO) is a way to interact with devices by issuing memory accesses. Instead of reading and writing to RAM, the accesses are passed to the device which can respond to the reads and writes as it wants. This can be used to expose device registers, memory chips (like RAM on a device) or trigger actions on the device. In PCI devices, the MMIO ranges of a device are determined by the BAR registers of the device which the device uses to specify how much MMIO it needs and the system to assign this amount.

When a driver wants to use a device's MMIO areas the first thing it needs to do is to reserve the memory range. This ensures that this memory range is not concurrently used by different parts of the kernel. Reserving a memory range is done by calling `request_mem_region` or one of the other memory request variants. Non-concurrent access is not enforced in any way, but a call to the reserve function will fail if the area is already reserved. When done, the driver must also free this region.

After reserving the region and before the region can be used, the driver needs to call one of the `ioremap` functions. These functions take a physical address and a size and return a `__iomem` type. According to the comments on the function [1] `ioremap_nocache`, this is required in order for the device to access it:

> `ioremap_nocache` performs a platform specific sequence of operations to make bus memory CPU accessible via the `readb`/ `readw`/ `readl`/ `writeb`/ `writew`/ `writel` functions and the other MMIO helpers. The returned address is not guaranteed to be usable directly as a virtual address.

The last sentence refers to the fact that on some platforms, including x86, the physical addresses given to `ioremap` are mapped into the virtual address space by the `ioremap` functions. On these platforms it would be possible to use the returned value from `ioremap` as a normal pointer. For the driver to be platform agnostic however, it needs to use the MMIO helper functions.

3.2.4 DMA API

Some devices, including PCI devices have direct access to the address space of the host. In PCIe, a device can issue a read or write transaction level packet (TLP) with any address. While a device could theoretically access any part of the host RAM it wants to, it has no good way to know where it should read or write and this information is usually given to it by the driver. This is communicated in a device specific way, usually with MMIO to a devices BAR. In proprietary devices such as GPUs we cannot know the protocol used.

There are however multiple platforms where the device does not have access to the full address space of the host. For instance, a platform can have a 32 bit bus but the host address space is 64 bit. Also, in platforms with an IOMMU, the device and host will have separate address spaces and the host is required to do some set-up to enable access to parts of its address space even if the address spaces are mapped in such a way as to be identical. There are also devices that are only capable of 32-bit addressing, and can only access the lower 32 bits of the host.

The Linux kernel supports both platforms with IOMMUs and devices only capable of 32-bit addressing. Since the Linux kernel runs on both platforms with and without IOMMUs with the same drivers, the drivers need to support this. This is done with the DMA API which provides a unified interface that works on all platforms where DMA is supported. For devices not capable of 64-bit addressing, it can do so called bounce-buffering where the buffer is copied to a buffer in the lower 32 bits before the device starts the transfer.

The DMA API introduces a new data type, `dma_addr_t`, which stores pointers in the devices address space. This makes it clear to the driver developers that the addresses are not interchangeable and cannot be converted to and from the other without the help of the DMA API. When the driver gives an address to the device it will be a `dma_addr_t` address.

The DMA API provides the driver with the ability to:

- Allocate coherent DMA buffers
- Set up streaming DMA mappings

- Make a single page available for the device
- Make scatter-gather list available for the device.

Coherent buffers can be accessed by both the device and the host at the same time. Any caching systems are set up so that the writes by one side will be reflected by a read on the other side immediately.

When a file is read or written to a disk, you want to minimize the amount of times the buffer is copied from one RAM location to another before written to the disk. If the only way to do DMA was to allocate DMA-able buffers, there would need to be at least one copy operation from the original buffer to the DMA buffer. To avoid the copy operation, the DMA API provides the ability to create a mapping of existing memory buffers - streaming DMA. The function `dma_map_single` takes a virtual address and size and return a bus address. However, contiguous virtual memory is often not contiguous in physical address space and to overcome this, the DMA API also provides scatter-gather capability. Unlike the allocation, the streaming mappings are not coherent which means that the driver needs to explicity synchronize the devices view with the CPUs view. These buffers also have an explicit direction given to the kernel, either from device to CPU, from CPU to device or bidirectional. This can be useful for the kernel in order to set up caching and synchronization. The DMA API documentation specifies when the buffer may be modified by either side depending on the direction given.

Devices that are unable to address memory above 32-bit can present problems with streaming mappings as a memory buffer will not be accessible if it is located above the 32-bit limit. For allocations the allocator can take this into account and allocate the buffer below 32-bit. This however places restrictions on the size and number of buffers as the space below 32-bit is limited and can be crowded. This also presents a greater challenge for streaming DMA mappings. If the buffer to be mapped is above 32 bit, the device will not be able to access it. With the help of an IOMMU, the buffer can be mapped into the lower 32 bit of the devices address space. However, there are many machines without an IOMMU, especially older machines. For a 32 bit device to use streaming DMA on such machines, a technique called bounce buffers is used. This technique is provided by the Linux kernel and is automatically used when necessary. It works by copying the mapped buffer to another memory buffer in the lower 32 bit of memory before the transmission is started. This requires the driver to collaborate with the kernel for the kernel to know when to perform the copy operations. This applies in both directions, from RAM to device and from device to RAM. This is part of the DMA API and is performed by the driver by using the synchronization functions that can be seen in code snippet 3.3. They provide the driver with the ability to synchronize the original buffer with the bounce buffer. The two "sync for cpu" functions must be called to update the view the CPU have of the memory buffer. When the device modifies a buffer, this must be called to allow the CPU to see the changes. When the CPU modifies the buffer the corresponding "sync for device" must be called before the device accesses the memory. [7] The same functions can also be used by the kernel in platforms where caches must be flushed and so on.

On a single host, different PCI devices can have different requirements. For instance, an IOMMU may not be present in all parts of the fabric, and an IOMMU has the ability to dynamically separate the devices into domains with their own address space. Since the different PCI devices may have different ways to set up DMA, the DMA API's core functionalities is implemented on a per device level. This also has to be dynamic since in platforms such as a PC, the PCI fabric is not known before runtime. The per device core implementation is controlled by the `struct dma_map_ops`. It contains function pointers to the implementation for a given

```
void
dma_sync_single_for_cpu(struct device *dev, dma_addr_t dma_handle,
                        size_t size, enum dma_data_direction direction)
void
dma_sync_single_for_device(struct device *dev, dma_addr_t dma_handle,
                           size_t size, enum dma_data_direction direction)
void
dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg, int nelems,
                    enum dma_data_direction direction)
void
dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg,
                       int nelems, enum dma_data_direction direction)
```

Code snippet 3.3: Functions in the DMA API to synchronize buffer between CPU and device

device.

3.3 BIOS, firmware and ACPI

When a computer boots up, platform specific software is responsible for initializing the hardware and getting everything ready for the OS to boot. On older PCs this was done by the machine's BIOS and other platforms had other forms of platform specific firmware. Modern PCs have Unified Extensible Firmware Interface (UEFI) instead of a BIOS. UEFI is a specification that defines a standard set of interfaces that are compatible accross platforms, for instance, UEFI has drivers for some devices and file systems that can be used to boot the OS. One of the tasks of the firmware is to scan the PCI fabric and configure the devices which is needed for instance when the boot disk is attached to a PCI based disk controller.

In addition to bootstraping the OS and setting up hardware, modern platform firmware provides various services to the OS. One such service is control of power on the system, allowing the system to for instance go to sleep. This requires support from the platform to set everything up correctly when waking up. The interface between the platform firmware and the OS is defined by the Advanced Configuration & Power Interface (ACPI). Originally ACPI was its own standard, but it has been absorbed by the UEFI standard. ACPI provides the OS with a set of tables which contain various useful information for the OS including devices the firmware has found and so on. These tables can contain objects, and these objects can, like object oriented languages, include methods. These methods are made using executable ACPI bytecode called AML that the OS runs these using its own AML interpreter. AML code has ring 0 access and can access any part of RAM or perform IO operations. ACPI methods allow the OS to query the firmware for information or perform platform specific operations such as power controls.

In addition to the ACPI bytecode, in most x86 systems, the firmware keeps running after the OS has booted. The firmware runs in a super privileged mode of the x86 CPU called System Management Mode (SMM). The kernel, which runs in ring 0 is not able to control SMM as SMM is completely isolated from the OS. The platform firmware can use this to intercept certain actions performed by the OS, for instance, this is used to emulate PS/2 mouse and keyboards and allow legacy OSes without USB support to be used with USB mouse and keyboards. Some AML code can trap into SMM to perform super privileged operations.

Since the firmware keeps running after the OS has booted, parts of the computer might be in the control of the firmware and not the OS, for instance, a USB controller when legacy PS/2

emulation is enabled. ACPI most therefore be used to negotiate control over hardware.

3.3.1 Hotplugging

In the Linux kernel, the `pcieport` driver handles all PCIe PCI-PCI bridges. Various optional extensions a bridge can have, including hotplug and AER (Advanced Error Reporting), are separated out into "service drivers" which attach to bridges that have the associated capability which includes both ACPI and native hotplug. All hotplug drivers have a common core that that handles the kernels reaction to hotplug events and covers everything from rescanning a bridge to controlling the various indicator lights.

3.4 IOMMU support

The Linux kernel provides a common interface for all the different IOMMU variants in different platforms (Intel VT-d, AMD-Vi or ARMs SMMU). This API, which is defined in the `iommu.h` header, allows devices to be added to domains and the domains to be controlled in various ways, including setting up address translation.

3.4.1 VFIO

Virtual machines has had the ability to directly control PCIe devices thorugh the use of IOMMUs for some time. However, there was not a unified interface used by the different virtual machine monitors, and it only supported x86. Virtual Function IO (VFIO), is an interface designed to fill this hole. The goal is to have a common interface for all platforms with IOMMUs capable of providing pass-through of devices. In addition, the interface is entirely user space, so no kernel modules or modifications are needed for a VMM to support device pass-through. VFIO will also allow device drivers to exist in user space as the IOMMU will provide the isolation between the kernel and user space. Since hardware virtualization can have very high performance, it is feasible to create user-space driver for high performance devices.

VFIO accepts configuration space accesses, while guaranteeing that this will not break the isolation. To do this safely, it needs to intercept some of the accesses and handle them in a safe manner. To do this, it will fully emulate parts of the configuration space, for instance BARs so that the guest OS believes it has control of the device BARs, but in reality it will change the IOMMU mapping and leave the BARs unchanged. In addition, the configuration space it exposes can have some capabilities entirely hidden.

A VFIO device is presented to user space as a file device. Various IOCTL operations can be performed on this device to interact with the hardware device. This includes configuration space accesses, setting up interrupts handlers and mapping MMIO areas using `mmap`.

3.4.2 Sysfs interface

Sysfs is a special filesystem used to control system parameters in the Linux kernel from user space. By mounting this filesystem (typically at `/sys`), user can perform various kernel related operations and see various parameters. Sysfs is populated by kobjects and attributes present in the kernel, including devices, PCI devices, drivers, modules and so on. By interacting with the sysfs filesystem, a user can for instance inspect the devices the kernel is aware of and perform actions on them.

```
irq
subsystem_vendor
class
power
sound
resource
consistent_dma_mask_bits
modalias
dma_mask_bits
local_cpus
config
device
driver
enable
subsystem
msi_bus
local_cpulist
remove
rescan
uevent
vendor
resource0
subsystem_device
numa_node
d3cold_allowed
```

Code snippet 3.4: Output from running ls on a PCI device in sysfs

The path `/sys/bus/pci/` contains attributes related to the PCI subsystem including the connected devices which are contained in the sub folder `devices`. Each device has a folder within the `devices` folder, named by its BDF. Inside the device folder are various properties of the device as seen in code snippet 3.4. This includes a link to the driver's folder, the interrupt number and so on. There are also writeable files that can be used to perform actions on the device, for instance, the file `remove` can be used to remove the device from the kernel as if the device was hot-removed and a full rescan of the PCI fabric can be initiated by writing 1 to the `rescan` file.

3.5 Chapter summary

The PCI subsystem in the Linux kernel abstracts away the hardware details from the structures and APIs used by the drivers. This provides us with a platform for allowing an unmodified driver to interact with a remote device. Multiple of the APIs provided to the drivers are hook-able at runtime which is useful for us in order to make the driver interact with a remote device.

Chapter 4

Design and Implementation

When we were starting out, we has little knowledge of how PCI and IO virtualization worked. We decided that in an effort to gain more knowledge of the subject, would start by reading about and testing related technologies, MR-IOV in particular. We also looked into the support for these in the Linux kernel and material on how they work.

We speculated that hot plugging of PCIe devices was a central part to being able to share or reassign devices. Assuming that the two would be similar from the software perspective, we decided to start with hot plugging. From what we had seen so far, hotpluging was not something that was normally used in PCI systems, we however discovered various mentions of it while researching. One of the technologies we read about was Thunderbolt which is a relatively new interconnect for consumer products. The key to our interest was discovering that Thunderbolt's main mode of operation was tunnelling of PCIe traffic, for instance, a Thunderbolt Ethernet adapter, is a PCIe based NIC. Thunderbolt provides us with the opportunity to test PCIe hotplug in easily available systems and to experiment with this functionality. Our first step in the practical part of the thesis work was to experiment with Thunderbolt. The hope was that this would allow us to learn about PCIe and how hotplug works in PCIe.

After a while testing Thunderbolt systems, a native PCIe hotplug set up became available to us. The set up was provided by Dolphin. Unlike with Thunderbolt, there was no layer beneath PCIe. In addition, the set up could be used with any normal PC. This allowed us to learn more about the differences between PCIe and Thunderbolt tunneled PCIe. Dolphin also have an external PCIe switch they use in their products. This switch was not MR-IOV capable, but had the ability to create partitions. This was however not something Dolphin had used much. Still, this provided us with an opportunity to create a setup where PCIe devices could be dynamically reassigned from one host to another as described in section 2.5. Without access to MRA switches, this was as close as we would come. At this point we had however discovered problems with the native hotpluging used in such a set up in Linux and we looked into this and various ways to improve it. While working on getting this set up to work we had to read and learn the inner workings of the PCIe switch in both Dolphin's NTB device as well as their PCIe switch.

While we were able to make hot plugging work partially, it was not enough. During this work an alternative came up. Dolphin's main product is their NTB hardware and related software. They believed that this hardware and software could be used to remotely access a PCIe device. After the problems we had with hotplug, we decided to look into this. This lead to the implementation of what we call "device lending". It allows one host to borrow a device from another host in a way that allows the native device drivers to be used without modification.

4.1 Experimenting with Thunderbolt

Thunderbolt is an interesting platform for us because it allows the hot plugging of PCIe devices in easily accessible hardware. We looked into it to learn how PCIe hot plugging worked and the current support for in the the Linux kernel. We had access to a Thunderbolt ethernet adapter, multiple Apple Macbooks and later a Intel NUC, a small form factor PC with Thunderbolt. We started by testing Thunderbolt on Apple computers. Hotplugging worked, as expected in OS X, and also as we expected, things were not so simple in Linux. Testing revealed that Thunderbolt devices that were inserted from boot worked correctly. The device showed up as a normal PCIe device behind a PCIe switch. The Linux kernel did apparently not detect that the device was removed when the cable was unplugged. The devices in `lspci` were now corrupted and displayed as all `0xff` as expected from a non-existing device. When the device was re-plugged, unsurprisingly the kernel did not detect it. However, a manual rescan of the devices did not yield any result either. This was surprising to us as our understanding of PCIe at the time suggested that this should not happen.

We also read articles, forum posts and comment sections to learn about other people's experience with Thunderbolt. The reports seemed to agree with our findings: Devices present at boot worked, but hot plugging did not. We suspected that since Thunderbolt exists at a lower layer than PCIe that some additional work was required to make hot plugging work. We attempted to look into the XNU kernel which is the open source kernel powering OS X. To our disappointment, the PCI and Thunderbolt code was not part of the open source version.

After a while we noticed that a patchset for enabling Thunderbolt support on Macs had been created and submitted to the Linux kernel. The patch was eventually included in the mainline kernel. The patch aimed at supporting hot plug events on Apple computers, but apparently it was currently simplified and did not support daisy-chaining. We tried out the patchset and as promised it enabled hot plug of Thunderbolt devices on the Macs we tested. This driver revealed a lot of information about Thunderbolt which we had previously been unable to gather. One thing we noticed were that when Thunderbolt devices was connected, the driver explicitly sat up the PCIe tunnel to the new device. This explained why rescanning did not work without the driver, a path to the new device was not set up. Thunderbolt was not in itself our main target and once it was working, we began looking at the Linux PCI subsystem involved and how it handled hot plugged devices.

Later, we got access to a PC with Thunderbolt. Unlike the Apple computers which needed the patchset, it worked flawlessly without any Thunderbolt specific patches to the Linux kernel. Our first thought was that this was due to the PC Thunderbolt variant being strictly PCIe hotplug. We however recalled the routing needed to be set up for new Thunderbolt devices. This is especially obvious since Thunderbolt does not only support PCIe, but also legacy DisplayPort using the same connector and no Thunderbolt support on the monitor. A closer look at the various outputs from the kernel revealed that the hotplug events for the Thunderbolt slot was handled by the `acpihp` driver. This driver delivers hotplug events it gets from the platform firmware via ACPI to the PCI subsystem in the kernel. We believe that the firmware implements the same as the functionality as Thunderbolt driver in the Linux kernel. The `acpihp` driver is not Thunderbolt specific but applies to any firmware controlled hotplug scenario and the firmware on a platform can handle other PCI hotplug setups in the same way. We followed the function calls in events from the Thunderbolt driver and the ACPI hotplug driver. This helped us identify the common code in both setups to learn how the kernel sets up new devices.

One thing that both the PC and the Macs had in common was that they reserved space for additional devices on the hotplug slot as well as all upstream bridges as described in 2.2.2. This is in contrast to what most firmwares do, and what the Thunderbolt systems did for all non-

Thunderbolt bridges. In these other cases, only the exact amount necessary is allocated. The extra resources for the Thunderbolt bridges allows the kernel to find room for the new devices.

4.2 Experimenting with Native PCIe hotplug

After a while, Dolphin approached us with a hotplug setup they had. They had access to PCIe expansion boxes, an external PCIe interconnect and a slot for the external cable. The slot was their NTB device configured to operate in a transparent mode. In this mode it works as a normal PCIe switch that shows up as bridges in `lspci`. They had a problem that sounded familiar to us, the devices worked fine when present under boot, but problems arose when hot plugging occurred. First of all, the hotplug event was not registered by the kernel. This caused the devices removed to not be cleaned up, but stay in the system in a broken state. This was similar to what we saw with Thunderbolt on Apple computers before the Thunderbolt patch was applied. In addition, hot add events was not detected. They worked around this by removing the devices manually with the Linux sysfs interface before physically unplugging the cable. Similarly, when the cable was re-plugged, a rescan was triggered with the same interface. Unlike with our testing with Thunderbolt, this worked and the devices were correctly discovered. However, the devices did not work as they should. While every device behind the slot was correctly discovered and identified, the kernel had trouble reinitializing them. We discovered that the problem was in allocating room for the device BARs. While this should not be a problem in theory, obviously there is room, everything was set up correctly before the removal, it turned out to be a significant issue. We believe this to be caused by the allocation algorithm used by the kernel not being able to fit the memory areas in the same windows as the firmware or that it has more strict requirements. The two problems with this setup was then the missing events and the resource allocation.

Examining the `lspci` output of the Hotplug slot, we discovered that it did not report itself as being hotplug capable. With this hotplug slot however, we had access to both the specification for the internal PCIe switch as well as it's EEPROM. We looked into the specification and the current EEPROM in an attempt to make hotplug detection work. We also hoped that by declaring itself as hotplug, the slot would be allocated extra resources by the firmware or the kernel.

4.2.1 NTB EEPROM modification

The EEPROM in the PCIe switch on the NTB device we got from Dolphin can be modified. By modifying this, we can control the configuration space of all bridges in the switch. In an attempt to make hot plug events work, we attempted to mark the slot as hotpluggable as specified in the PCIe specification. The first we needed to do was to mark the downstream bridge where the hotplugged devices appeared behind as a slot. This is because only slots have the slot capability register where the hotplug capable bit resides. In the same manner, only downstream bridges can be slots. The upstream bridge was marked as upstream. The other was marked as both downstream and a slot. The slot capabilities register was now valid. In this register, the slot was marked as hotplug capable in addition to hot plug surprise capable. After applying this EEPROM modification and rebooting the machine, we examined the output from `lspci`. We could now clearly see that our modifications allowed the slot to be identified as such and that the status register changed when the cable was unplugged. Still, hotplugging did not work. Looking at the `lspci` output more closely we saw that the OS had not enabled any features in the control register such as interrupt on hot plug events. We also noticed that `pcieport`'s

service driver for native hotplug was not loaded for the slot.

4.2.2 Hotplug - ACPI

Burrowing ourselves deep into the Linux kernel we discovered a suspect for the hotplug service driver not loading and placed some debug prints there to confirm. The kernel will for every bridge create a service bitmask of what the bridge is capable, this is the basis of what service drivers are used. In addition to this, a permissions bitmask also exists. It is the logical and of these bitmasks that decide what service drivers are used for a given bridge. These permissions bits signify if the kernel was allowed control over the associated device and feature for a given bridge by the platform firmware. Each PCI device is represented by a ACPI object that is available to the kernel and each of these objects have a `_OSC` member object. According to the PCI Firmware Specification, the `_OSC` object is:

> A Mechanism for Exposing PCI Express Capabilities Supported by an Operating System

Looking at this specification we were able to confirm that the `pcieport` driver in the Linux kernel correctly requested native hotplug control from the firmware. For some reason however, the firmware on our test machine denies the request for native hotplug. At this point, the kernel has no choice but to not use native PCIe hotplug. Since hotplug does not work, and no hotplug event is delivered via ACPI, we can assume that the firmware does not support hotplug. Possibly there is a known hardware compatibility issue that prevents hotplug. We also assume that on ACPI hotplug platforms like Thunderbolt, native hotplug would be denied in the same way.

Feeling brave, we enabled an option in `pcieport` to ignore any denied requests to the firmware. After this, the kernel detected hotplug events correctly. In addition, the slot control register was now changed: Presence detect change enable and hot plug interrupt enable had been set. We have tested this on our machine without any adverse effects. This would suggest that the refusal to hand over control for hotplug is a firmware bug. After all, it seemed to work just fine. It might be that the firmware developer simply did not bother to test hotpluging and deny it to be on the safe side.

When a device was hot-removed this was correctly detected by the Linux kernel. It cleaned up the devices and associated drivers correctly. While this seemed to work, we noticed that some drivers were unable to handle an actual hot remove as opposed to the software only remove when using the remove operation in sysfs. The difference is that when using the sysfs remove, communications with the device will continue to work even after the event, but when the link is abruptly severed, any transaction to the device will fail. For instance reads will result in `0xffffffff`. This can also happen before the driver is notified by the kernel that the device is no longer present.

While working with the PCIe switches we discovered their ability for partitioning. As described in section 2.5, this feature allows the PCIe fabric to be split into multiple separate fabrics. When this is active, multiple root complexes can be connected to a single switch and and have their own PCIe fabric. If this could be changed dynamically, we could possibly use this to allow for device sharing. This however still required us to fix the device resource allocation problems.

4.2.3 Resource issues

In short, the problem with allocating resources to hot-added devices is that the resources allocated to the bridge may be insufficient. There are two ways to solve this: Make sure the

bridge has enough resources from the start or grow the resources of the bridge. The first is the approach taken by the Thunderbolt implementation on the NUC and the Apple computers we tested with. We also suspect that this is done in most platforms with support for native PCIe hotplug due to the lacking support for any other technique in the Linux kernel. This solution is very simple and has few issues, the main one being not allocating enough resources. This can happen if the machine has multiple hotplugable slots, in such a case, the fairly limited bus numbers can be a problem (255). Unfortunately, this technique requires support from the platform firmware, something which the platforms we tested on did not have.

Since the Linux kernel has to follow the setup done by the platform firmware, hot-add will probably not work without the support from the firmware. Our PC with thunderbolt suggests that a platform with hotplug friendly firmware will work with Linux. If this is the case, hotplugging, including hot-add should work without any modifications required to the Linux kernel. If hotplug works, PCIe switches with partitioning and MR-IOV switches should work as well.

When a bridge does not have the necessary resources either because no extra space was pre-allocated, or because the pre-allocated resources were insufficient, the only solution is to grow the resources of the bridge. However, growing the resources of a bridge is significantly harder than pre-allocation and comes with a number of issues. While in theory the bridges can be reconfigured dynamically while the system is running, the reconfiguration will have an effect on the devices behind it. Also, for the bridge widows to expand, other bridge windows may need to be moved, further affecting other devices. Drivers currently bound to a device that is moved will cause issues, the main issue being that the addresses it uses will be incorrect after the move. We have explored the idea of using an IOMMU to abstract away the fabric windows from the addresses used by the drivers. This would make it possible for a device to change it's BARs without the driver knowing. This still does not solve the problem of the device losing traffic or being reset. Timing would be extremely critical if communication could not be suspended somehow.

In theory, the entire PCIe fabric can be reconfigured by removing all devices from the PCI subsystem, rearranging the fabric and then rescanning it. This is however made very complex by the fact that the disk and other critical systems can depend on these PCI devices. This would probably also affect user space programs currently interacting with a device, including file IO, network traffic and GPU computations.

Apple's "Thunderbolt Device Driver Programming Guide" [5] introduces driver writers to the special considerations that needs to be taken when the device is a Thunderbolt device rather than simply a PCIe device. This sheds some light on how Thunderbolt hotplugging is implemented in OSX. In addition to the perhaps obvious, that the driver needs to be prepared to lose all connections to the device at any point in time, drivers are also encouraged to support "PCIe pause". It is optional for a driver to support this, but it is highly encouraged. This functionality appears to allow the kernel to pause the driver, change the address spaces of devices and notify the driver of the new address spaces. At this point the driver must check all of its address spaces as they may have changed. We believe however that this may be hard for some drivers to implement. In the section on "PCIe pause", the guide suggests that the developer disable another feature called "spread" to increase the likelihood / frequency of the need for "pause" to be used. We speculate that this spread feature increases the address spaces of the bridges upstream of the thunderbolt ports to make room for new devices. The output from lspci in OSX appears to support this theory. Possibly the Linux kernel can implement something similar to OSX's PCIe pause. For instance we have theorized that existing suspend functionality can be used by suspending the device, moving it and then resuming it. Hopefully, when the device is suspended, the driver saves the device's state in order to restore it when

power is returned. When the state is saved and the driver has prepared for device power down, there should be no traffic to and from the device and moving it should be safe. If the new values can be communicated to the driver when the device is resumed, the driver should be able to continue normal operations. Suspend will probably affect user space programs currently interacting with the device or it's driver however.

4.3 Experimenting with NTB

While working on hotplugging and resource allocation, the idea of accessing a PCIe device on the other side of an NTB link occurred to us. Dolphin thought this was possible, and had done some preliminary testing but had not actually tried to interact with a device beyond probing the bar areas. This was done using their API function `AttachPhysicalMemory` which allowed us to map arbitrary physical memory addresses. We tried mapping the BAR areas of a PCIe device with this function and it allowed us to read and write to the a device's BAR areas in one host from another host. We believed that this could be used to essentially remote control a PCIe device from another host. The memory operations that are forwarded from the NTB to the device will be directly sent to the device, and not through the other host's CPU or chipset. This will not only ensure high performance, but also not affect the performance of the other host by incurring a overhead on the CPU or chipset. Instead, the only affect to the other host is the traffic between the NTB and the device as well as the traffic on the NTB itself which it might be using.

Using an NTB was a promising idea because it side-stepped the problems we had met so far. First of all, resource allocation should not be a problem as the remote device would not be part of the users PCIe fabric. Instead the resources would be consumed from the NTB itself. Secondly, it should allow for greater flexibility than both MR-IOV and PCIe switch partitioning since it's entirely controlled by software. This is because it's built on top of PCIe instead of into PCIe itself and can co-exist with existing hardware and software.

The NTB idea showed lots of promise, and Ladon showed that it was possible (see section 2.10.4). We however, had not discovered this at the moment. We feared that we would not be able to complete such an implementation in time or that it would be impossible due to some technical detail we had not thought of. When we later discovered Ladon and Merlin, we were convinced that this would be possible. Still, there was much to do as implementation specific details are scarce in Ladon and Merlin.

For this idea to offer a good alternative to normal PCIe switches and MR-IOV, existing drivers would need to be supported. We also wanted the needed requirements and modifications to the kernel to be minimal. We decided to attempt to implement a proof of concept with the goal of using an unmodified device driver to use a PCIe device behind an NTB on another host. The proof of concept was developed for Linux and uses Dolphin's NTBs and software.

4.3.1 Device lending

We invented the term "device lending" to describe our NTB remote access solution. Compared to MR-IOV, device lending offer many of the same capabilities and advantages. Unlike MR-IOV with the Virtual Hierarchies (VH) however, there is no change in the hierarchy of the PCIe fabric with our device lending. Instead, everything stays standard PCIe, but with the added ability of a device to be accesses remotely. A remotely accessed device will stay in the owner host's PCIe fabric, and will not be part of the other hosts PCIe fabric.

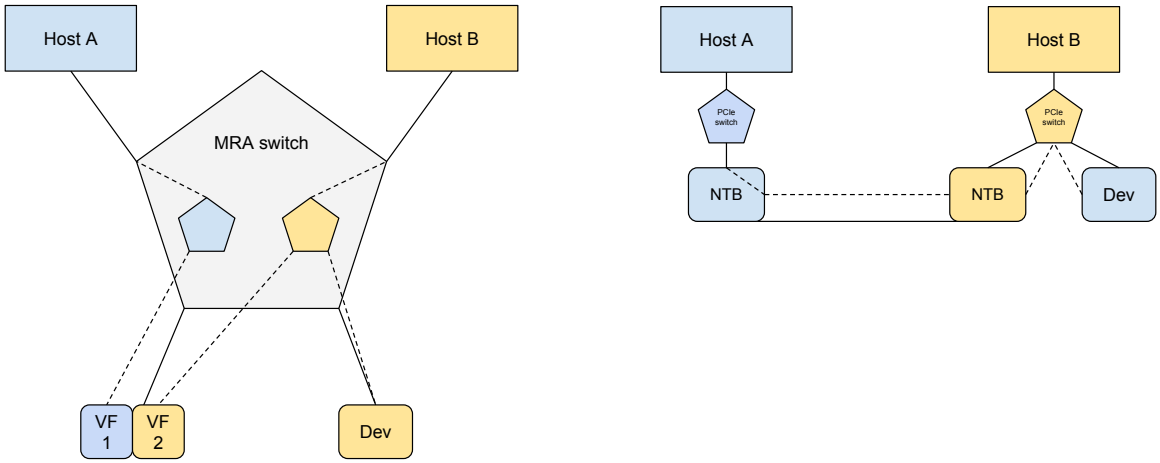


Figure 4.1: Left MR-IOV. Right: A host borrowing a device from another host using an NTB.

Additionally, the NTB based device lending is more flexible in the devices that can be borrowed or shared than MR-IOV. A comparison can be seen in figure 4.1. MR-IOV requires all devices to be assigned to the VH to be downstream of the MRA switch the host is connected to. This makes MR-IOV incapable of sharing a device locally connected to a host since a local device will probably be upstream of the MRA switch. While sharing local devices is not possible in either MR-IOV or with a partition-able switch, it has some distinct advantages. A locally connected device can have a 16 PCIe lanes directly to the CPU. The bandwidth and latency between this device and the CPU will be at the absolute maximum the machine can provide. A device that is externally connected will often have limited bandwidth to the CPU. For instance the cable used by Dolphin is only 8 lanes, half of what an internal slot can provide. While 16 lane external cables exist, the bandwidth of this external link is still shared by all devices connected behind it. In addition, externally connected devices will have higher latency, if still very low. We believe this provides a compelling argument for having as much as possible of devices with high bandwidth requirements connected locally. Our NTB based device lending scheme allows such devices to be utilized remotely, but still gives the devices the optimal bandwidth when used locally.

A significant advantage MR-IOV has over PCIe switches with partitioning is the MRA capable devices. These devices can be controlled by multiple hosts at the same time, like SR-IOV for multiple hosts, as described in the section about MR-IOV, 2.4.3. This advantage is however limited by the fact that currently few or no available devices are MRA capable. Further, we hope that the device lending can apply to the virtual devices created by an SR-IOV device. If so, such device could be shared among the hosts. This has also been demonstrated to work, see section 2.10.3. This would make our solution one of the very few that can share PCIe devices between multiple hosts. We believe it also has the potential to be one of the cheapest.

For our NTB based device lending to work with unmodified drivers we would need to fool the drivers into thinking they were interacting with a local device. Since the drivers directly interacted with the PCI subsystem we needed a way to get the PCI subsystem to play along.

```
/* Low-level architecture-dependent routines */

struct pci_ops {
    int (*read) (struct pci_bus *bus, unsigned int devfn, int where,
                 int size, u32 *val);
    int (*write)(struct pci_bus *bus, unsigned int devfn, int where,
                 int size, u32 val);
};
```

Code snippet 4.1: pci_ops structure in the Linux kernel (include/linux/pci.h)

4.4 Injecting remote device into Linux PCI subsystem

We aim to inject a non-existing device into the kernel. This leaves us with two alternatives. The first is to duplicate the device creation in the kernel and create our own `pci_dev`. Since most information about the device comes from this structure, we should have full control over all parameters of the device. The device scanning and management code in the kernel is however not trivial. It would be a substantial amount of work to duplicate and understand all parts of the code involved. In addition, the device would, after creation, need to be handed off to the kernel. Because of this, the device would still need to be set up so that the kernel can handle it in the same way as the physical PCI devices. Also the memory management of this device would be difficult as the PCI subsystem might decide to modify or free memory associated with the device. Since the PCI subsystem did not allocate it itself it could possibly be confused. We also fear that future changes to the kernel would render this code unusable without modifications, possibly creating subtle bugs.

The alternative would be to allow the PCI subsystem to think that it sees a new device that it will initialize itself. The Linux kernel acquires its information about the devices using configuration reads and writes. If we could emulate or intercept these we could make the kernel see a virtual device and have it initialize its structures.

4.4.1 Intercepting configuration space accesses

As explained in 3.2.2, each PCI bus in the kernel has its own set of configuration space access operations. They are stored in the `pci_ops` structure which can be seen in code snippet 4.1. To intercept the configuration accesses performed by the kernel, we can change the function pointers in the `pci_ops` structure on a PCI bus we want control over.

When the functions are hooked we have full control over all accesses. This allows us to trick the kernel into thinking there is a device on a bus that is not physically there. To do this, the read function will need to return the same values as a physical device. The configuration space of a device is specific to each device type as each can implement different capabilities and so on. In order for the configuration space accesses to be correct, we need to recreate the same behaviour as the physical device we want to access remotely. Sending the configuration space accesses to the physical device across the NTB link is the obvious solution. However, configuration space accesses cannot be forwarded by an NTB.

One way to implement the configuration space is to emulate it, similar to the way VFIO does. This means that we attempt to return the same as the device would in the same situation, but in software. A good first step is to read the configuration space of the target device into

a memory buffer. When we intercept a read operation, we can redirect it to the buffer. For much of the configuration space, this is sufficient as it is read- only. Writes are however, more challenging. Even if the writes are not important to the operation of the device, such writes cannot be simply ignored as there are places in the kernel where the kernel will read back the result of a write to confirm its success. This requires us to allow write operations to modify the in-memory buffer.

However, some writes are part of a more complex protocol, such as the BAR registers to which the OS writes all `0xFF` and reads back another value. In addition, some registers can be changed by the device, for instance to report errors. In these cases the in-memory buffer will be insufficient. VFIO has good support for this. It sometimes ignores writes, reflects them to the emulated configuration space or passes it through to the physical device. This is however quite complex. Also, since some writes are passed through to the device, we still need access to the physical device.

In an attempt to avoid the complexities of configuration space emulation, we first redirected configuration space accesses to an identical local device. If the local device is identical, the kernel will correctly identify the device. In addition, the local device will correctly respond to the configuration space accesses. This was however never more than a stopgap solution. First of all, having an identical local device can be a problem and this will decrease the flexibility of the system. Furthermore, if some of the accesses change the behaviour of the device in any way, we will affect the local device, and the remote device will not be configured as expected by the local kernel. One example of this is the enable bit in the MSI capability register. If we don't update this on the physical device when the write occurs, the driver or the kernel can be confused when a interrupt comes when it doesn't expect it or doesn't come at all. This implies that we need a way to do configuration space access across the NTB link to the target device. While these accesses cannot be transparently bridged across the NTB link, we identified three alternatives:

Some of the NTBs we have access to has the ability to generate configuration requests. This can be controlled from a connected host. In theory we could use this to send requests to the physical device. This will however not work in our set-up where the device is in a host machine. This is because configuration requests can only be sent downstream. In a host, the path from the NTB device to the target device will always include at least one upstream bridge. If however the target device was located in an expansion box, the NTB would be placed upstream of all devices as if it were the root complex.

In a similar vain, there are two other ways for us to inject configurations space requests into the target machine. The first is to take advantage of the fact that in modern machines configuration space accesses are done using MMIO to a platform dependant memory location reported via ACPI. We could possibly access this register from the other side of an NTB. We dismissed this as a poor solution for two reasons. First, we would likely run into concurrency issues with the local CPU. Secondly we believe the chipset or other hardware might block the access to these registers from anything but the CPU. The other way to inject configuration requests is to utilize the packet generator present in some PCIe switches. This is a vendor specific feature not available to the public. It is unclear to us how we would be able to receive the responses.

The final method, which we ended up using, is to have the other host generate the configuration requests. With the help of a communication channel, we proxyed the reads and writes through to a user space daemon on the other host. The daemon perform the accesses by utilizing a *sysfs* file provided by the kernel (`/sys/bus/pci/devices/*/config`). This daemon also implements a basic emulation to allow us to ignore some writes but still reflect them in subsequent reads. This is done by having a copy of the configuration space in a memory

buffer. Depending on the write it might be written only to the buffer or both the buffer and the device. All reads will go through the buffer.

4.5 Access to device BARs

A devices registers exist in the range specified by the BAR registers. In Linux however, drivers will not read the devices configuration space themselves, but use the values stored in the devices `pci_dev` structure. This is described in detail in 3.2.1. This means that we have control over the addresses seen by the driver as the device is injected by us.

Examination of the MMIO functions on x86 revealed that they are all small functions or macros and are very likely to be inlined. Since our goal is to avoid driver modifications, any tricks needed to make sure that the driver interacts with the device on the other host needs to be at the level of these MMIO helpers or lower. Since the functions are inlined, there is now way for us to hook them at runtime. We could however modify the source and recompile the kernel. The helper could then be made hookable. This would likely decrease performance of MMIO heavy code.

Another alternative would be to set up the virtual memory mappings in such a way that the memory accesses could be intercepted in hardware. A page fault handler could possibly perform the necessary steps to communicate with the device. This would also negatively affect the performance of MMIO, but could be limited to only the remote devices. The drivers would also not need to be recompiled. We are however unsure exactly how the page fault handler would work. This would possibly work better as a virtual machine extension.

The solution we went with however, was at PCIe level. Before the device was injected into the PCI subsystem, we mapped the device BARs on the owner host to physical addresses on the loaning host using an NTB shown in code snippet 4.2. These addresses lay within the BARs of the NTB on the loaner host. While the device is injected into the PCI subsystem, we make sure that the BAR addresses stored in the `pci_dev` structure of the virtual device is our mapped addresses into the actual BARs on the other side as shown in code snippet 4.3. When the driver loads, it will use these addresses to access the device. This should ensure that the driver communicates with the remote device. When the driver probes the device, it will reserve the BAR regions and perform an `ioremap` call. This required us to make some changes to the NTB driver, see chapter 4.10.2. Other than this, the driver was now able to use the MMIO registers of the device as if the device was local.

4.6 Device interrupts

In traditional PCI a device signalled an interrupt by driving a dedicated interrupt pin. The pin was shared by multiple devices so the host had to query the devices to see which device currently requested service. While this is supported in PCIe, it is considered legacy. The preferred way to do interrupts in PCIe is with message signalled interrupts (MSI). Support for this is required for all PCIe devices that can generate interrupts. There is also MSI-X, which is an extension to MSI. When a device issues an MSI interrupt, it will do a normal write transaction to a given address. This is received by the chipset which generates a interrupt to the CPU. On x86 a write to `0xfeeXXXXX` will trigger an interrupt.

```
/* The segment ids we export in this function are used by the other side to
 * connect and map our BARs. When the other side connects, it will get a
 * local IO address that is valid from it's own side of the NTB that point
 * to the relevant BAR. This will be the new "BAR" values to be used by the
 * machine borrowing the device
 */

for(int i = 0; i < number_of_bars; ++i) {
    /* Map the BARs local IO address using NTB software. */
    if(bar[i].ismmio)
        export_area(SEGMENT_ID_START + i, bar[i].start, bar[i].size);
}
```

Code snippet 4.2: Exporting the BARs of a device

```
int map_bars(segmentid_t start, size_t number_of_bars, bar_t* bar) {
    for(int i = 0; i < number_of_bars; ++i) {
        /* Connect to the remote BARs and get the local io address for this
            host using NTB software. */
        bar[i].start = map_area(start + i);
    }
    return 0
}
```

Code snippet 4.3: Mapping the BARs of a remote device

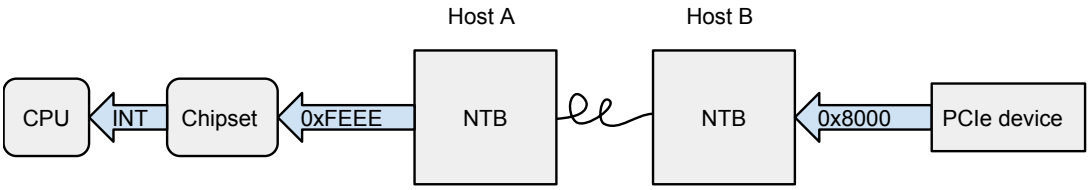


Figure 4.2: An MSI interrupt delivered from a device on one host, through an NTB and to the CPU on the remote host.

When a driver sets up the interrupts for a device, it is the kernel itself which sets up the interrupts, decides parameters and configures the device. When the kernel that loans the device configures the interrupts on the device, it will not be aware of the fact that the device is not local. The addresses set up by it will reach the CPU of the machine in which the device is physically located. We need to somehow redirect this interrupt to the driver on the correct host. This also needs to be done as fast as possible as interrupts can be performance critical.

One solution would be to hook one of the functions that the driver calls to set up interrupts. We could then do the setup on the host with the device physically present. On this host the interrupt handler would simply pass the interrupt on to the host where the driver is running. The interrupt could be send via an interrupt feature provided by the SISCI API. The kernel module on the other side would then call the interrupt handler of the driver when it received the SISCI interrupt.

Another way to support MSI is to allow the interrupt to go directly through the NTB and into the chipset of the host with the driver. With this solution the interrupt is handled entirely in hardware and should give better performance. This works because MSI are signalled in the same way as memory writes in PCIe. These memory writes can go across the NTB and to the chipset of the host where it will generate an interrupt for the CPU as illustrated in figure 4.2. MSI is configured in the configuration space of the device. This includes the address to which the write is performed. Since we have already hooked the accesses to the configuration space, we can intercept this easily. By mapping the address required to generate interrupts on the loaner host and using the mapped address as the interrupt address, the interrupts will be delivered across the NTB. This technique will not work for either MSI-X or legacy interrupts. Legacy interrupts because they are not memory accesses and cannot be transferred across the NTB and MSI-X because we are unable to intercept the addresses used. This is because the addresses used by MSI-X are placed in a BAR and are direct MMIO operations. Support for legacy interrupts would need to be implemented using our first technique. MSI-X could however be supported by a hybrid approach where the interrupt setup was hooked so that we could change the address, but still directly delivered across the NTB.

4.7 Device initiated DMA

In the same way that the host can access a device's mapped memory, by issuing TLP memory read and write, devices can also access the host's memory. The target address can be anything within the address space of the device, the bus physical address space. The RAM is normally located within this address space. In addition, all other devices are normally within the same address space. This means that the device should be able to access the BAR areas of other devices as well. When the device is located in another host as in our case, it will have access to a different machine's address space. This includes this machines RAM and other devices MMIO areas.

Take a look at figure 4.3. On the left side, the device is local. When the driver use the DMA API, the dma address it gets will be the address that hits the correct buffer in RAM in the devices address space. On the right side however, the device is in another hosts address space. The driver is on Host A. Consider a buffer the driver has allocated in RAM. It's bus physical address on host A is `0x1000`. The driver gives this address to device: "Read address `0x1000` - `0x1100`." When the device reads this address, it will be within the address space of host B, not host A. Obviously this pointer will point to something else here. For correct device operation, the device and driver needs to know of the difference in address space or we need to translate

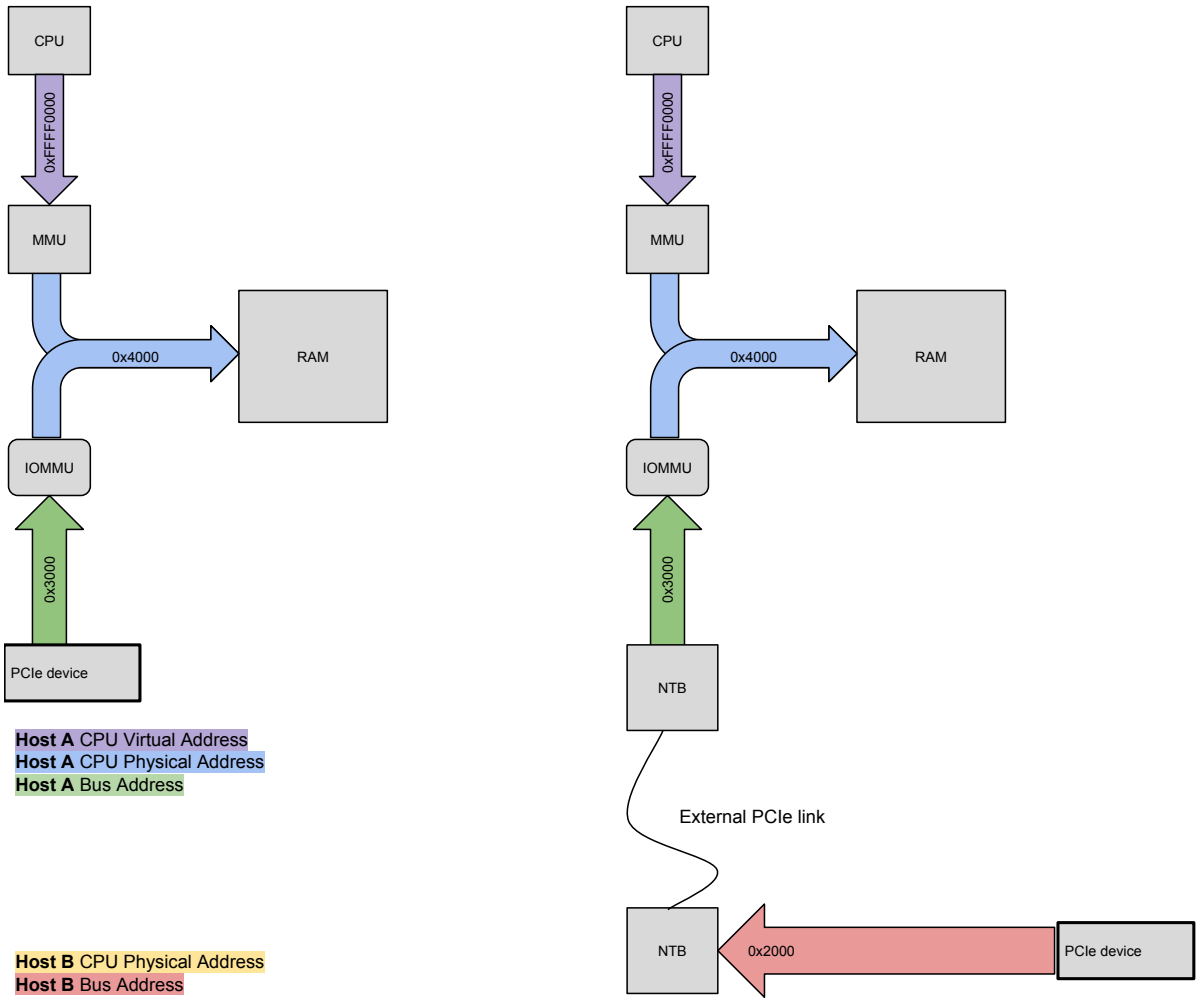


Figure 4.3: A device accesses a buffer in RAM. On the left the device is attached locally. On the right, the device is in another host and the device is in another address space.

```
/*
 * A dma_addr_t can hold any valid DMA or bus address for the platform.
 * It can be given to a device to use as a DMA source or target.  A CPU cannot
 * reference a dma_addr_t directly because there may be translation between
 * its physical address space and the bus address space.
 */
struct dma_map_ops {
    void* (*alloc)(struct device *dev, size_t size,
                   dma_addr_t *dma_handle, gfp_t gfp,
                   struct dma_attrs *attrs);

    void (*free)(struct device *dev, size_t size,
                 void *vaddr, dma_addr_t dma_handle,
                 struct dma_attrs *attrs);

    int (*mmap)(struct device *, struct vm_area_struct *,
                void *, dma_addr_t, size_t, struct dma_attrs *attrs);

    int (*get_sgtable)(struct device *dev, struct sg_table *sgt, void *,
                       dma_addr_t, size_t, struct dma_attrs *attrs);

    dma_addr_t (*map_page)(struct device *dev, struct page *page,
                           unsigned long offset, size_t size,
                           enum dma_data_direction dir,
                           struct dma_attrs *attrs);

    void (*unmap_page)(struct device *dev, dma_addr_t dma_handle,
                       size_t size, enum dma_data_direction dir,
                       struct dma_attrs *attrs);

    int (*map_sg)(struct device *dev, struct scatterlist *sg,
                  int nents, enum dma_data_direction dir,
                  struct dma_attrs *attrs);

    void (*unmap_sg)(struct device *dev,
                     struct scatterlist *sg, int nents,
                     enum dma_data_direction dir,
                     struct dma_attrs *attrs);

    void (*sync_single_for_cpu)(struct device *dev,
                                dma_addr_t dma_handle, size_t size,
                                enum dma_data_direction dir);

    void (*sync_single_for_device)(struct device *dev,
                                   dma_addr_t dma_handle, size_t size,
                                   enum dma_data_direction dir);

    void (*sync_sg_for_cpu)(struct device *dev,
                            struct scatterlist *sg, int nents,
                            enum dma_data_direction dir);

    void (*sync_sg_for_device)(struct device *dev,
                               struct scatterlist *sg, int nents,
                               enum dma_data_direction dir);

    int (*mapping_error)(struct device *dev, dma_addr_t dma_addr);

    int (*dma_supported)(struct device *dev, u64 mask);

    int (*set_dma_mask)(struct device *dev, u64 mask);

#ifdef ARCH_HAS_DMA_GET_REQUIRED_MASK
    u64 (*get_required_mask)(struct device *dev);
#endif

    int is_phys;
};
```

Code snippet 4.4: The dma_ops structure

the addresses somehow. For the memory operations from the device to reach the driver-side host, they will need to cross the NTB. For this to work correctly, NTB mapping must be set up in the same manner as when mapping the BAR areas, only in the reverse direction.

The DMA API will call the function `get_dma_ops(dev)` to get the core DMA implementation for a given device. The function returns a `struct dma_map_ops` which we have included in code snippet 4.4. This function is platform specific and on some platforms it may return the same for all devices. The kernel includes a default implementation of the `dma_map_ops` called `nommu_ops`. It will use other platform dependant functions to convert to and from virtual address space, but will assume there is no IOMMU. On x86 `get_dma_ops` can return a per-device `dma_map_ops` if the kernel compilation time configuration `CONFIG_X86_DEV_DMA_OPS` is set. If so, the function returns `dev->archdata.dma_ops`. `dev->archdata` is a platform dependant structure stored in all devices. On x86 we can provide our own implementation of the DMA API on our virtual devices by changing this member. When the device is injected, this is changed to our own version of the struct that implements the functionality required for correct operation across the NTB link. We will now go through the various operations in this struct that we implemented.

4.7.1 Allocation of coherent DMA buffers

The alloc funtion in the DMA API allows a driver to allocate memory that the device can DMA to and from by returning both a virtual address for the CPU, and a `dma_addr_t` address for the device. The trivial solution to allocating DMA-able buffers would be to wrap the original `dma_ops` implementation and map the resulting buffer with the NTB so that it is usable by the device. However, this is very inefficient as drivers often allocates multiple small buffers (around 4KB), but the NTB is not well suited for this. This is because he original allocator spreads out the allocations over a large area. The result is that when we map this, the Dolphin driver has to use one NTB page per allocation. Depending of the configuration of the BAR size on the NTB, each page will take multiple mega bytes. If the BAR size is configured to be 1024MB, each page will map 32 consecutive MB. This is clearly wasteful. In addition, the NTB is only capable of mapping around 20 pages so we will quickly run out and be unable to map any more memory.

To better utilize the mapping resources, we should allocate the DMA buffers inside a single page. We do this by implementing a memory pool allocator that find an available space within an already mapped pool. As long as the drivers do not allocate a lot of DMA buffers, we can fit all of it within a single NTB page. Ideally this pool should be an integer multiple of the NTB's page size. When the mapping is connected, we get the address to the pool from the device-side. Any address inside the pool can be calculated from both sides by adding the same offset to both addresses.

Our initial pool implementation is extremely simple allowing for only the minimal amount of recycling of memory. Space for a new allocation was always placed after the last allocation and counter is incremented. When a buffer is deallocated, the counter is decremented. When the counter reaches zero, all allocations had been freed and the next buffer to be next buffer pointer was reset to the start of the pool. Obviously this algorithm had a lot of weaknesses. First of all, it was unable to free any memory at all unless all was freed. This saved us much complex book keeping and made the allocator very simple. Depending on the allocation patters of the driver, however, it could easily be unable to allocate more even with lots of free space. A simple enhancement that still kept the book keeping to a minimum was a stack-like model. A check was added in the free operation. If the buffer to be freed was the same as the last allocated, the next pointer could be moved back to the same pointer given to the free operation.

```
/* We get the local io address of the area to be mapped by calling the
   original DMA implementation. */
dma_addr_t local_bus_addr = orig_impl(size);

/* We map this area using the NTB software. */
handle = map_area(local_bus_addr, size);

/* The other side also needs to connect to this map and sends us its local
   io address which will be our remote io address. */
dma_addr_t remote_bus_addr = request_remote_to_map(handle);

/* This address is the one the device driver orders the device to use. It
   is only valid from the device-side */
return remote_bus_addr;
```

Code snippet 4.5: Hooking the DMA API

For drivers that always free their allocations in the reverse allocation order this is sufficient. A more intelligent allocation algorithm was not developed as it was not necessary at the time. Most drivers only allocate a small number of static buffers at device initialization and free them at de-initialization. A driver that dynamically allocates and deallocates and not in the order required by our stupid algorithm to reclaim it will quickly run out of space. To tackle this a better algorithm is required. There are many requirements to such an algorithm. Among the important ones are the fact that it should avoid fragmenting the pool. The memory allocator in Linux is very capable and could serve as a reference implementation. We believe however that fragmentation would not be a huge problem as the drivers we have seen often only allocate only a few equally sized buffers.

4.7.2 Streaming mappings

Streaming mappings allow an already allocated memory buffer to be directly accessed by a device. The alternative would be to copy the contents of the buffer to a dedicated DMA buffer.

Streaming DMA is implemented by the `map_` / `unmap_` functions in the `dma_map_ops` structure. The map page variant maps a single physically contigous memory range. Since virtual memory ranges may not be physically contigous, there is also a scatter-gather (SG) variant. Both variants return a `dma_addr_t` type address, or a list of such addresses in the case of the SG variant. These addresses are in the address space of the device.

When we hook these mapping functions, we will wrap the original implementation and return an NTB mapped address as shown in snippet 4.5. The `dma_addr_t` we get as result from the original DMA core call is the addresses we need to map. This is very similar to mapping the device's BARs, only the other direction. The Dolphin driver's genif interface will be used to create this mapping. After the mapping is created, the device-host must connect to it. The device side host will also be able to fetch the bus address of the mapping after connecting to it. This address is the one the driver must tell the device to use. This address cannot be known on the driver-side. Because of this, we must communicate with the device-host when setting up streaming mappings. In addition to acquiring the bus address, we must also wait for the other host to connect before returning. This is to ensure that the mapping window is set up before the device starts any transfer.

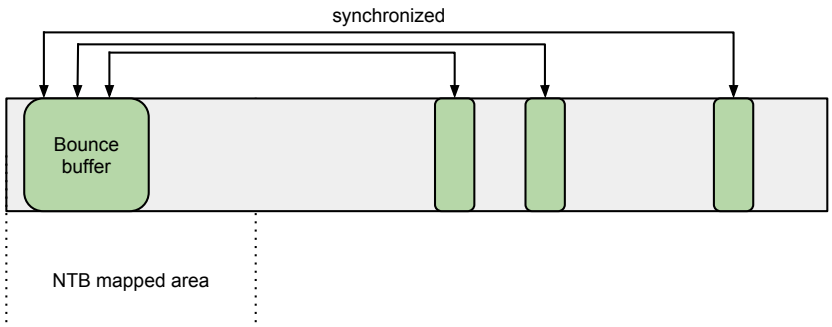


Figure 4.4: Bounce buffer mapped with NTB

The other host may take some time in responding to the mapping request. This can be a performance issue depending on the time this takes and how often new mappings are set up. Since the other side is a user space process, we can expect some additional delays from context switches, scheduling and so on. Not only can this be bad for performance reasons for the device interaction, but it can affect system wide response time since we might be in an interrupt context, this is covered in more detail in section 4.7.4.

Each single mapping will need its own entry in the NTB mapping table and since the number of mappings are fairly limited, this can become a problem. Multiple mappings which are near in physical memory can be mapped by a single NTB mapping entry. When a new mapping is requested by a driver, we check if the area is already mapped. If so, the mapping can be reused. To ensure that this mapping is not tore down before all users have freed their buffers, each mapping has a reference count. This also has the added benefit of not needing to communicate with the other host to set up a new mapping all the time and can improve performance as well as conserve mapping resources. Still, in the worst case scenario, where each mapped area is far apart in physical memory, one mapping entry is needed per mapped area.

The scatter-gather variant in particular will spend a lot of pages to map. It is possible that we will be unable to meet the demands of a driver or for there to be very high contention on the mapping resources. When we are unable to perform a map we can return `0x0` to indicate an error. The DMA API encourages drivers to retry later in case of failure as this is likely due to a temporary problem. In our case however, when a mapping request can never be completed due to the demands being above the capabilities of the NTB, we have no way to report this. A driver might keep retrying forever, hanging the system. The swiotlb DMA implementation described in the next section also faces this problem in addition to drivers that simply ignore the error code. It circumvents this by returning a pointer to an emergency pool or simply issuing a kernel panic [1]:

> Ran out of IOMMU space for this operation. This is very bad. Unfortunately the drivers cannot handle this operation properly unless they check for `dma_mapping_error` (most don't). When the mapping is small enough return a static buffer to limit the damage, or panic when the transfer is too big.

Bounce buffer

Since we had limited mapping resources we looked for alternative solutions. We considered the bounce buffers used for 32-bit devices. The same bounce buffer technique can be used to place all streaming mapped memory areas within the bounds of a single NTB mapping as illustrated in figure 4.4. The bounce buffer in the Linux kernel is implemented in `swiotlb.c`

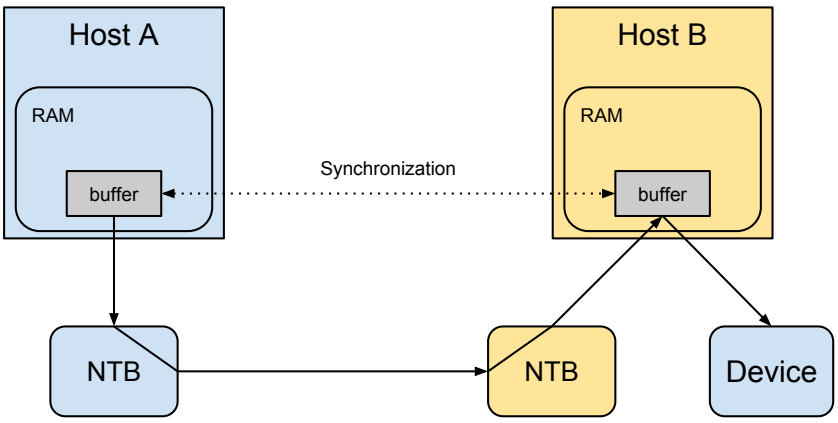


Figure 4.5: Streaming DMA with a bounce-buffer on the other machine.

as a DMA API implementation. Looking at this code, we determined that it keeps a pool that all mappings are placed in that is 64MB by default. We can map this entire area within two mappings. This can greatly reduce the number of mapping entries required. To utilize `swiotlb`, we need to make sure that it handles all mappings and allocations. We first thought that simply using it's DMA core implementation would be sufficient. However, upon closer inspection, we discovered that it does not use the bounce buffer unless it needs to because the device is unable to address the original mapping. Because of this, we need a way to force it to use bounce buffers for our device always. A bootflag, `swiotlb=force`, causes bounce buffers to be used for all DMA operations on the host, always. This is obviously not ideal since bounce buffers have a performance penalty. Another way is to mark our remote device as not 64-bit capable (although the NTB is 64-bit capable and allows a 32-bit device to access memory above 32-bit). This will force bounce buffers to be used when a memory map is requested above the 32-bit limit, but not if the buffer is already below this limit. Still, the later is probably the best choice. The final alternative is to modify `swiotlb` or copy it's functionality into our kernel module.

By default on x86 systems with no IOMMU, `swiotlb` is used. Our implementation so far, that wraps the default DMA core implementation will use `swiotbl` in this case. Combined with the merging of overlapping mappings and reference counting no additional modifications to the implementation is needed. To guarantee that the mappings cover the entire bounce buffer, we map the memory area used by `swiotbl` in the initialization of our kernel module. Using bounce buffers also eliminates the need for communication with the remote host each time a new streaming mapping is needed, at least as long as the bounce buffer is used.

Unfortunately, the performance reductions for bounce buffers can be a problem. Since each buffer now has to be copied at least once. Possibly since this implementation would not have to communicate with the other host when setting up new mappings, the performance penalty of copying can be outweighed by the performance loss of the required communication. This especially applies to driver that create new mappings all the time, for instance when performing DMA directly from a device to a user space buffer. Possibly, a mix can be used. For instance when mapping resources have run out, our DMA implementation can switch from direct mappings to using bounce buffers.

Cross machine bounce buffer

The bounce buffer technique will still require at least one mapping. We have however, also considered a streaming DMA that requires no NTB mapping resources. The idea was to create

a bounce buffer, but creating the bounce buffer on the remote host instead as seen in figure 4.5. When the buffers are synchronized, the data is transmitted to the buffer on the other host. This synchronization can be done using the DMA engine on the NTB, or even network access.

When a driver allocates or maps a memory buffer, an identically sized buffer will need to be created on the device side host. The pointer to the device-side buffer will be given to the driver as the DMA address. This will cause the device to read the buffer in the RAM of the host its connected to. This buffer needs to be kept synchronized with the buffer on the driver side host. The DMA API features synchronization functions used with streaming DMA mappings as explain in section 3.2.4. Like the rest of the DMA API functions, the sync functions are implemented in the `dma_map_ops`, see code snippet 4.4. These functions allow a driver to tell the kernel when to synchronize the bounce buffer.

In the `sync_x_for_device` functions, the buffer on the driver side must be copied to the device-side buffer and in the `sync_x_for_cpu` functions, the buffer must be copied from the device-side buffer to the driver-side buffer. In the synchronization hooks, a copy from one to the other buffer will need to be started and completed before the function returns. This synchronization can be problematic as it can take some time. To transfer the buffer, the DMA engine on the NTB device can be used, but we can just as well use PIO from the CPU since we need to wait for the transfer anyway. Using PIO also has the advantage of not having to call any Dolphin functions that might not be interrupt safe (more in section 4.7.4).

4.7.3 Utilizing an IOMMU for increased mapping granularity

A typical IOMMU has a much higher granularity than an NTB and it's capable of a much larger number of pages. A way to simplify the mapping is to utilize an IOMMU on the lender side and use it to improve upon the mapping capabilities of the NTB. Combined, the two can allow us to get rid of the pool allocator as well as greatly improve the streaming mappings. We have identified two ways for us to utilize an IOMMU to increase the granularity and total number of mappings. The first is to use the DMA core implementation for the IOMMU present. The other is to do our own mapping.

In much the same way as `swiotbl` can be used to decrease the amount of mappings required, an IOMMU DMA implementation can fit more memory in the same amount of NTB mapping area. Since we wrap the current DMA core already, using our implementation on a system with IOMMU enabled can possibly improve the mapping issues without any further changes. The reason for this is that the IOMMU core can make the bus addresses closer to each other and merge SG mappings into a single, large, contiguous memory segment. It would also be beneficial to us if the bus addresses from the IOMMU mappings are placed close to each other since we also allow an NTB mapping to be reused if an overlap is detected. We have investigated the behaviour of the Intel IOMMU DMA core implementation to see if any of our hopes holds true. First of all, we discovered that like `swiotlb`, the IOMMU core will only remap if a device is not 64-bit capable. This can however be overridden using the boot option `iommu=force`. Also, unlike with bounce buffer, an IOMMU should have substantially lower overhead, so this might be a viable option. Depending on how the IOMMU core places the remapped memory , all mappings can now end up very close to each other and save us multiple mappings. If this holds true, our solution can be substantially improved simply by enabling the required features. An optimization to this would be to pre-map the area the IOMMU used for remapping to get a full coverage in the least possible NTB mappings. If we don't do this, the first mapping inside this area can be reused later, but it would probably not start at the start of the remap area but some offset into it.

A different alternative is to use the IOMMU API to set up our own mappings. Compared

```
dma_addr_t map_area(dma_addr_t local_cpu_addr, size_t size) {
    dma_addr_t remote_io_addr;
    dma_addr_t local_io_addr;
    uint64_t pool_offset;

    /* IOMMU is programmed to translate memory requests between
     * local_cpu_addr and local_io_addr
     */
    pool_offset = pool_map_chunk(local_cpu_addr, size);

    /* This is the address used by the device on the other side
     * of the NTB link. It is calculated from the address of the
     * previously set up NTB mapping.
     */
    remote_io_addr = pool_remote_io_start + pool_offset;

    /* This address is the one emited from the NTB on the local
     * side. We don't really need this.
     */
    local_io_addr = pool_local_io_start + pool_offset;

    return remote_io_addr;
}
```

Code snippet 4.6: Create mapping using IOMMU

to the previous idea, this is far more work. The idea is to directly improve the Dolphin driver ability to map arbitrary memory areas over the NTB using an IOMMU if available. This can also be built on top of the Dolphin driver. First, a single NTB map is set up and the address it translates to is mapped with the IOMMU and set to unavailable or denied. This area serves as our mapping pool. When a new mapping is created, it is consumed from this pool. The offset into the pool can be used to calculate the correct remote bus address and the IOMMU is used to map this chunk of the pool to the desired address. The permissions also need to be set at this point. To find an available chunk in our pool, we can either use something extremely simplistic as our coherent DMA buffer allocator, or something more involved like the Linux kernel memory management (buddy allocator).

The allocator should be aware of the remaining space, preferably taking fragmentation and such into account. When it's running low on space, a new pool can be allocated by creating a new NTB mapping. In the same way, an entire pool should be freed when there are no allocations in it and another pool has sufficient space. Since we cannot move allocations, we need to be smart about where the allocations are placed if we want to be able to free a pool while the system is running or we risk there always being an allocation in every pool.

Our wrapper function need to use the `nommu` DMA core so that we don't end up mapping the area with the IOMMU twice. Similar to our previous implementation, we map the result address using our new IOMMU based mapper instead. The return value from this can be return from the wrapper. This applies to `map_page`, `map_sg` as well as `alloc`. Mapping an already existing memory area should be as simple as seen in code snippet 4.6. There are clear advantages to this approach. If it's implemented in the Dolphin driver, other applications can also benefit from it.

4.7.4 Interrupt context considerations

While developing we discovered that some of the functions we hooked could be called from interrupt contexts. This happens when the function is called either directly or indirectly from an interrupt handler. We first noticed this with the `dma_unmap_sg` function. A device driver we tested with mapped a memory area and started a DMA transfer. When the transfer was done, the device fired an interrupt. The drivers interrupt handler in turn called the unmap function to clean up the mapping for the completed transfer. We assume that the other wrapped functions in addition to the `pci_ops` functions also can be called from a interrupt context.

A function that is called in an interrupt context has quite a few limitations on what it can do according to "Linux Device Drivers" [8]:

- Cannot interact with user space
- Must return quickly
- Cannot invoke the scheduler

Interacting with user space is not something we are interested it, so this should not be a problem. The two other requirements however, are.

There are multiple ways to enter the scheduler in addition to simply calling `schedule` and all of them need to be avoided in an interrupt context. This of course includes any external function called inside the interrupt context. Sometimes, which functions end up scheduling can be surprising. Among them are any function that uses locks as the lock will call schedule if the lock is taken. This does not apply to spinlocks. One perhaps unexpected place to find a reschedule is the memory allocation functions. Unless the flag `GFP_ATOMIC` is specified, kernel allocator such as `kmalloc` might schedule to acquire the memory. Unfortunately, most functions do not have an atomic flag. This includes all networking and disk IO. Because of this, any function call we make needs to be examined so that we can be sure they do not schedule.

The wrappers in our implementation can be accessed concurrently. To protect our data structures from concurrent access some form of mutual exclusion or atomic access is required. Since we cannot schedule, normal locks cannot be used. Because of this, we need to use spinlocks. Since we do not know, for any given call, if we are in an interrupt context or not, we need to use the spinlock functions `spin_lock_irqsave` and `spin_unlock_irqrestore` instead of the plain `spin_lock` and `spin_unlock`. These functions will save the current state of interrupts so that the release function only re-enables interrupts if it was enabled when the spinlock was locked.

While a spinlock is taken, we need to make sure that we don't re-enable interrupts, attempt to re-enter the lock or otherwise cause the spinlock to fail. In the same way as we need to look for `schedule` in functions we call, we also need to avoid functions that re-enable interrupts, for instance by calling `spin_unlock` on it's own spinlock. Since the spinlock disables interrupts we want to avoid holding it for a long period of time. This is also an argument for minimizing the area protected by the spinlock.

We need to communicate in multiple of our wrapper functions. We started off doing the communication using TCP, but the socket API is however, not atomic and we cannot use it when we are in an atomic context. Theoretically, if we could detect that we were in an atomic context, we could attempt to queue operations or similar, but there is no way to know this for sure. In addition, there are some of our wrappers that need to wait for the response from the other side before it can return. In some of them, we have found no other way out either. Because of this, we needed to create a communication channel that was atomic. Still, the problem remains that the interrupt handler should be quick. This can be a problem, depending on how long the other

machine takes to answer. We should attempt to minimize this by ensuring that the messages themselves are delivered quickly. We should also do our best to ensure that the daemon on the other side answers quickly. Finally, in the event that no answer comes due to for instance communication error or that the program crashed, we need to time out and fail gracefully. This is very important as these functions can be called deep in the kernel.

4.8 Shared memory based communication channel

Since network now was out of the question, we either needed a communication channel that can be used in atomic contexts, or we needed to avoid communicating. We however realized that in some cases communication was absolutely necessary. One example of this is when MSI configuration registers are written to. When this happens, the write needs to be directed to the physical device for correct operation. We also believe that queueing such a write can be problematic as we suspect this to cause race conditions when the caller expects the device to behave as if the the write is complete when the function returns.

Using the NTB to get the message itself across we could either use DMA or mapped memory. Since the messages are very small (< 100 bytes), we decided to use mapped memory. To save mapping resources, the message buffer is placed in the same pool as the coherent DMA buffers.

The communication channel follows a request-response pattern. When the host wants to send a request it will write the message contents to a buffer located in its own memory. A request-id is also part of this message. This request-id is incremented each time a new request is made. The other host will respond by writing to another buffer in the first hosts memory. When done, it will copy a request-id from the request into the response buffer. This allows the requester to spin on the id in the response and immediately react when the response is written. A simplified version of this can be seen in code listing 4.7. The communication channel cannot be used concurrently and must be protected by a lock. To keep the channel interrupt safe, a spinlock must be used. The communication channel also have a simple timeout mechanism. If the timer runs out, we will treat it as an error and report it as such. This timeout is very short to avoid potentially blocking an interrupt handler for too long.

The SISCI API has the ability to send interrupts across hosts to another SISCI application. This allowed us to quickly alert the other host that we need to communicate. This is implemented by a function in the Dolphin driver. For us to use the interrupt capability, this function would need to be atomic. We should probably look through the function and the function it calls to verify this. However, a simple test seems to indicate that the function is atomic. If for whatever reason the function behaves in a non-atomic way, we could do without it by having the client spinning for ever. When the host receives the interrupt, the user space program needs to be notified. This can involve a context switch to the process and can take some time compared to the interrupt itself. If the time from the interrupt is fired to the interrupt handler is run is long, the spinning variant can be faster. However, the spinning variant will occupy one CPU core of the device-host. A hybrid approach, which we implemented, might be a good compromise: Each time an interrupt is received, the client runs in a loop waiting for additional messages for a while before returning.

When the client detects a new request, it will read the other hosts shared memory to find the request type and its parameters. When the response is ready, the client writes the response to the response buffer in the server's memory. To signal the completion of the request, the request-id is copied from the request to the response. This is the final step and indicates that the server can now read the response. A simplified version of the interrupt handler in the client can be seen in

```
//Prepare message
request->msg = 42;
request->id = global_id++;

send_interrupt();

//Wait for the other host to respond
while(response->id != request->id) {
    /* Send once in a while, in case an interrupt is lost.
     * Not so often as to storm the other machine and
     * cause slowdown
     */
    send_interrupt();
}

//We have the result
result = response->msg;
```

Code snippet 4.7: Our simple shared memory based communcation channel algorithm

```
void handle_interrupt() {
    if(request->id != response->id) {
        do_stuff();
        response->id = request->id;
    }
    return;
}
```

Code snippet 4.8: The client of the shared memory based communcation channel

code snippet 4.8.

4.9 Device owner daemon

The host that wants to share one of its devices need to provide a few services to the loaner. In our implementation these services are implemented by a user space daemon.

We want to give the loaner kernel access to the devices configuration space. This it vital for a lot of reasons, including identifying the device type and other vital information. It is also vital in setting up the device. On the other hand, there are parts of the configuration space that can disrupt or even adversely affect the owner host. Some of these registers the Linux kernel will write to under normal circumstances to configure the device. One example are the BAR registers. When the kernel on the loaner host discovers our virtual device, it can attempt to write to the BAR registers to configure it to be within its own windows. However, any such address would not be valid within the other host and even if they where, it would disrupt the mappings. We need to prevent this from happening. In some cases we can bypass parts of the kernel to prevent it, but at other parts of the kernel we have no choice. In addition, the kernel will frequently read back the previously written registers to confirm that it succeeded. We need

to allow the kernel to believe it has access to these registers and to emulate it in such a way that the kernel will not detect it.

Our first take on this problem was to have an in-RAM copy of the configuration space and have the configuration space accesses redirected to this. The accesses we needed the user to be able to physically modify on the device we passed on to the device in addition. Access to the configuration space is exposed as a file to user space programs in Linux. It was fairly easy to implement a first version of this solution. After a while however, we saw that correctly implementing this can be hard. To do this we need to understand all the different PCIe capability registers and emulate anything we can't allow. Fortunately for us, we discovered that the VFIO interface does this. The capabilities of VFIO is described in more details in 3.4.1.

Since VFIO is designed for use in virtual machines, user space drivers and not "bare metal" use, it has some short comings that we need to circumvent. The first we discovered was that VFIO virtualizes parts of the MSI capability and completely circumvents it as interrupts are still delivered through the Linux kernel. There is a separate interface for controlling interrupts. This means that for some of our accesses, we need to use another access method than VFIO.

While VFIO has the capability to provide us with increased isolation and better handling of configuration space accesses in general, the need for special handling would have increased the complexity of our proof of concept implementation. Because of this we decided to not implement VFIO support.

Drivers and devices will not be expecting multiple hosts to access the same device at the same time. As we have already stated, this is outside our mission statement. The exception to this, SR-IOV devices, are regarded as separate devices and will get no special treatment. To enforce this, the daemon must always know which other host, if any, is currently using a device. Before loaning the device to another host, it needs to disconnect the device from the current user. The current user can also be the local host. The best way for it to prevent the local kernel from interfering with the device is to unbind any currently bound driver from the device and bind a shim driver. The shim driver will prevent another driver from binding to the device. This is the same method used by VFIO and is required before the VFIO interface can be used on a device.

One of our user space daemons will be spawned per device that is exported over the NTB link. When started, it should unbind the driver that has currently bounded to the device and bind a shim driver instead. This ensures that the local host will not use the device concurrently with a remote host. To ensure that only one remote accesses the device a the same time, the daemon will only accept a single connection. The current implementation does not unbind the host driver or attempt to prevent the host from concurrently using the device. This should be implemented, when the solution gets closer to being finished.

Providing the other host with access to the device's BARs is done using the SISCI API. The API allows us to expose the BAR areas over the NTB. The daemon also provides a reverse mapping service to the loaner. This will allow the loaner to map parts of its own memory and make it available for the device. This is used to allow device-to-host DMA and MSI interrupts. Upon a request, the daemon will connect to the memory mapped by the other host. It will return the local bus address to this mapping. This is important as this is the address is the one the device must use to reach the mapped memory. This is used when the driver orders the device to perform DMA.

4.10 Modifications to the Dolphin driver

During our work, we encountered some issues with the Dolphin software stack. The first was the we were unable to create more than 4 mappings. This became an issue with devices with more than 4 MMIO BARs. Multiple DMA buffers and SG-lists were also a problem. This limitation was due to a feature of the Dolphin stack conflicting with our arbitrary mappings. The other issue was that the device drivers and the Dolphin driver both reserved the same memory region. This is because we map the remote devices' BARs through the BARs of the NTB. We also encountered issues with the `ioremap` attributes due to the double mappings of the BARs.

4.10.1 Lacking mapping resources

As explained in chapter 2, an NTB has a limited number of possible simultaneous mappings. To account for this, the Dolphin software will place multiple segments in a single mapping. To do this, a pre allocated buffer in memory will be mapped. This means that some of the mapping resources are already consumed before any user mappings are created. The remaining mappings are called "on demand" mappings. This number can be edited in a compile time constant. This was necessary to have enough mappings for our purposes. In addition, the largest single mapping as well as the total mappable size is governed by the BAR size configured for the NTB. For some of the devices with large bar areas we need to increase the BAR size of the NTB to fit all mappings within it. This can be configured up to 1024 MB with a Dolphin supplied tool. The NTB hardware itself has the ability to go above this, but this also requires modifications to the dolphin driver. We were able to make this minor modification. Unfortunately the platform firmware for the machines we tested this on where unable to allocate BARs larger than 1024MB. We speculated that this was because the BAR was 32 bit, and that the space below the 4GB 32-bit limit was too crowded. We made some attempt to mark the BAR as 64 bit by modifying he EEPROM of the NTB device. Unfortunately, we could not get this to work either. At this point we gave up increasing the BAR size at it was not critical for all but a few devices. The reason for the 64-bit change not working can possibly be that the machine we tested always allocated all BARs bellow 4GB for some reason or simply could not handle such large BARs (it's fairly uncommon for BARs to exceed 256MB for compatibility reasons). One way to circumvent this for mapping large areas is to use multiple mapping entries. However, because of how the Dolphin driver works, we have no control over if the segments are mapped as contiguous as we need them to be. If they are not, we cannot use it.

4.10.2 Reserved memory regions and ioremap attributes

Before using ioremap to get access to an MMIO area, drivers and the kernel itself will call one of the `request_mem_region` functions. This function is a service provided by the kernel to ensure that there is no concurrent access to the same area. The call will return a value indicating if access was granted or if this area has already been reserved. If it fails, access to the area is forbidden, but there is nothing preventing the driver from accessing the area anyway.

Areas mapped with the NTB are inside the BAR of the NTB. The Dolphin driver will map it's own BAR when the driver is loaded and it calls ioremap. After we have created our own mappings, the driver for the remote device will also call `request_mem_region`. Since this area is overlapping, the call will fail.

To avoid this, either the device driver or the NTB driver will need to avoid reserving the region. Since we aim to use unmodified device drivers, we will modify the Dolphin driver to

avoid reserving the region. A more clean solution would be to have the Dolphin driver reserve the regions it actually accesses or uses, not the regions which are only mapped.

In addition to reserving the memory region of the BAR, the Dolphin driver will ioremap the entire BAR. There are multiple flags that can be used for an ioremap-ed area. They control caching, write combining and other attributes that affect the data flow from the CPU to the area. See 2.8 and 3.2.3 for more details. The Dolphin driver enables write combining for the entire BAR meaning that all mapped memory will have this enabled. This causes issues when used with MMIO to non-prefetchable BARs as writes and reads can be merged and re-ordered. Our experiences with this is detailed in section 5.1. To work around this problem, we modified the Dolphin driver to not enable write combining. This is far from the ideal solution as this negatively affects performance. It also does not change the fact that the same area is mapped in both the Dolphin driver and the device driver, leading to *aliasing*. Intel strongly discourages this [12]. Ideally, the Dolphin driver should only ioremap the areas it needs access to, or that's mapped to user space. In addition, at this point, the attributes needed might be more known. For instance write combining might not always be desired. This would however mean significant modifications to the Dolphin driver, something which is definitely outside the scope of this thesis.

4.11 Usage

The Linux kernel provides a sysfs interface for controlling PCIe devices. It can be used to remove a device in the same way as a hotplug event. Our implementation should support the same remove command, but should in addition respond to hotplug events from the NTB link as well as other errors. Error handling from the NTB is not yet implemented, but most of the drivers we tested with noticed the error itself due to the communications with the device stopping.

To make a device available to a remote, a user will start the device sharing daemon, specifying the device in an argument. On the remote host, the IP and port of the daemon must be specified. This is done by writing to a sysfs file made available when the kernel module is loaded. After this, the device will be available on the remote host. The daemon must be kept running while the device is in use. When the daemon exits, connection to the device is severed. When the control of the device should return to the owner, either the daemon is exited or the sysfs remove is used on the loaner host.

4.12 Chapter summary

This chapter first described what we researched and learned before we arrived at the NTB based solution. Later, our device lending design was described as multiple functionalities, some of which we have implemented. Our current implementation, based on this design , provides the necessary functionalities to allow for our device lending. However, some problems still remain including non-atomic Dolphin functions and lacking mapping resources. We believe, that both can be solved by using an IOMMU as explained in section 4.7.3. The next chapter goes into the details of the problems we faced with both earlier implementation as well as the final one.

Chapter 5

Evaluation and Discussion

To debug our implementation while developing we used a number of test devices and these devices also allowed us to validate the correctness of our implementation and evaluate its performance. While debugging, we mainly wanted devices where it would be relatively simple to understand the interaction between the driver and the device. Because of this, we also favoured devices with open source drivers in the Linux kernel. These devices were used to both trigger any bugs and shortages in our implementation as well as allowing us to clearly verify correct operation. For benchmarking the performance of remote device access, we needed devices where we could design an easily reproducible procedure that we could use to quantify the performance.

The table 5.1 show the devices we used and how well suited we believe they are for debugging, validation and benchmarking as well as the complexity of the driver and the interaction with the device. We have also created a table 5.2 that show the features used by each device/driver in our implementation.

5.1 Intel HDA audio codec

The Intel HDA audio codec is well suited for testing and validation. It has an open source driver in the Linux kernel and the device specification is available [10]. This enables us to better understand the driver as well as the actions performed by the device. Finally, a standard user space program can be used to play audio and we can verify the audio output from the device.

5.1.1 Device-driver interface

The device has a single non-prefetchable BAR that contains various device registers. The device has support for both legacy interrupt and MSI, but only MSI will be used by the driver. Finally,

Device	driver	complexity	debugging	validation	performance
HDA audio codec	ALSA	low	✓	✓	
Intel Ethernet card	Linux kernel	medium	✓	✓	✓
NVME SSD	Linux kernel	medium	✓	✓	✓
Nvidia GPU	noveau	high	✓	✓	✓
Nvidia GPU CUDA	closed	very high		✓	✓

Table 5.1: The various devices we tested and our assessment of them

Device	non-prefetch BAR	DMA alloc	DMA map	DMA SG map	MSI
HDA audio codec	✓	✓			✓
Intel Ethernet card	✓	✓	✓		✓
NVME SSD		✓		✓	✓
Nvidia GPU	✓	?	?	?	✓
Nvidia GPU CUDA	✓	?	?	?	✓

Table 5.2: The features used by the various devices

the device has a DMA engine so that it can access the RAM of the host. The MMIO registers in the BAR are specified in the Intel HDA codec specification [10]. In addition to the MMIO device registers, commands are sent to the device in a set of ring buffers, CORB and RIRB. The command ring buffers are present in host RAM and their addresses are programmed into device registers. The device will directly access the ringbuffers using DMA. The driver and device will communicate their progress in the ringbuffers by reading and writing to a device register. There is also simpler command scheme referred to as "single command mode" in the Linux kernel driver. It is intended for use by BIOS and not for normal operations.

5.1.2 Audio playback

When the device is handed to the driver by the device subsystem, the driver will start initializing the device and set up communications. To do this, it will allocate DMA-buffers from the Linux kernel using the DMA-API [7]. When this device is remotely accessed using our implementation, this DMA allocation will end up in our DMA pool allocator described in section 4.7.1. Our allocator will return a virtual address used by the driver and a bus address used by the device. The bus address, when used from the device on the other host, will go through the NTB and end up in the same CPU physical address as the virtual address. When probing the driver will discover the device's audio input and output streams. Each stream will have a single Buffer Descriptor List (BDL) and multiple data buffers. These allocations will be handled by our allocation in the same way as the ring buffers.

To initiate audio playback the driver will program a BDL to point to one or more of the allocated data buffers. The audio data will be written into these buffers by the PCM library of ALSA. In at least some cases, a user space program will be given direct access to these buffers. This is the case with the tool `speaker-test` The driver will write the address and length of the BDL into the device registers and start the playback by writing to another register.

At this point the device will read the BDL using the BDL address in its own registers. The address of the first audio clip is stored in the BDL. Both of these addresses need to be bus addresses that hit the same buffer as the virtual address used by the driver. In the remote case, both should be bus addresses that go through the NTB and hit the other hosts buffers in RAM from the devices perspective. When the device has completed playback of an entry in the BDL, the device will issue an interrupt if the interrupt bit is set in the entry. To generate an interrupt, the device will perform a write transaction to the address stored in the MSI capability's address field in the devices configuration space. In the remote case, this address should go through the NTB and hit address `0xfeeXXXXX` of the host using the device. The chipset, CPU and kernel will deliver this interrupt to the device driver. The driver can use the interrupts to assess the progress of the device. The device will also write its progress into a host allocated buffer called the DMA position buffer. Alternatively, the driver can read the position from a set of device registers. When the device reaches the end of the BDL it will loop back to index 0. While the device progresses through the BDL, ALSA and/or the user space application will need to

change the already played buffers to new audio data to create a continuous stream of audio.

5.1.3 Testing remote device access

Debuging the Intel HDA audio card lead to the correct implementation of multiple of the functionalities described in chapter 4. Specifically, the device depended on device initiated DMA for correct operation in addition to MSI interrupts. Before we started testing with this device, we had no working implementation of device initiated DMA and testing this device directly lead to the implementation of DMA buffer allocation feature described in 4.7.1 and lead us to realize the need to disable WC as described in section 4.10.2.

When we started testing the implementation using this device, we saw that the driver was unsuccessful in probing the device. To debug this, we enabled single command mode in the driver. This was enough to get the driver to successfully probe the device. At this point, the audio card showed up in user space programs such as `alsamixer` and we could change the volume and see the various capabilities of the card. Since the ringbuffers depended on DMA working, we suspected that our DMA allocator or mapping was broken.

When we attempted to run `speaker-test`, which should produce pink noise, we instead heard mostly silence with short beeping noises. The output was repeating. We speculated that what we were hearing was mostly zero bytes, with a few bytes here and there. This again caused us to suspect our DMA code.

To debug this we dumped the BAR0 of the device, while `speaker-test` was running, to examine the device registers. We did this both when using the device locally and remotely to compare. We expected that the BDL address might be wrong, or that the device was somehow unable to access the buffers. We examined the BDL address and the buffer it pointed to, and both appeared to be correct. We also attempted to manually write the stream descriptor from our dump from when the device was be used locally to the device while the remote host was running `speaker-test`. This produced the correct sound. Careful comparison of the two dumps revealed that, the BDL Last Valid Index (LVI) was set to zero when we used the device remotely, but some non-zero value when used locally. We attempted to write the same value to the register while the device was playing remotely, and correct sound was produced.

We found the function in the driver responsible for writing this register, see code snippet 5.1. The driver was clearly writing a value to this register, so we inserted a print to see what value it was writing. This caused the problem to go away, both audio playback and recording was now working flawlessly. A sleep instead of the print also produced the same result. This suggested some form of a race-condition. Our first theory was that this was because the registers were in a non-prefetchable BAR, but the Dolphin driver had mapped this area through one of the NTBs prefetchable BARs. We feared that this somehow cached some reads or writes, caused a memory race or merged them into larger reads/writes. If this was the case, it would be hard to fix as it would mean substantial changes to the Dolphin driver. To test our theory we modified the driver and inserted a memory write barrier before the write to the register we had problems with, see code snippet 5.1.

This fixed the issue and seemingly confirmed our fears. During a discussion with Dolphin about this, it became clear that the Dolphin driver enables Write Combining for the entire BAR region of the NTB. This could also cause similar issues. In particular the function in question wrote two different values to the same register. Presumably this means that writing to this register has some side effect, possibly enabling the writing of the other registers. With WC enabled, the CPU might merge these writes into one. The order that the registers are written in

```
static int azx_setup_controller(struct azx *chip, struct azx_dev *azx_dev)
{
    azx_sd_writel(chip, azx_dev, SD_CTL, val);

    wmb(); //Adding this fixed the issue

    /* program the stream LVI (last valid index) of the BDL */
    azx_sd_writew(chip, azx_dev, SD_LVI, azx_dev->frags - 1);

    /* set the interrupt enable bits in the descriptor control register */
    azx_sd_writel(chip, azx_dev, SD_CTL,
              azx_sd_readl(chip, azx_dev, SD_CTL) | SD_INT_MASK);

    return 0;
}
```

Code snippet 5.1: This method produced incorrect results for us with WC enabled. Most of the function has been removed to reduce the noise. Notice that the `SDL_CTL` register is written twice.

might also be important. When WC is enabled, the CPU provides very few guarantees about the order the memory accesses occur in. For more information see section 2.8. To make matters worse, the Intel X86 manual states that mapping the same physical area to more than one virtual mapping is not recommended ("aliasing"). If the two mappings have different attributes, the CPU behaviour was undefined. In our case, the Dolphin driver mapped the area with WC, but the driver mapped the area without WC. A quick hack in the Dolphin driver allowed us to disable WC for the entire BAR. This would decrease performance for various other users of the NTB, but sufficed as a test. This not only fixed the audio playing issue, but also the problem we had with the verb ring buffers.

5.1.4 Missing features

With the HDA audio card, both playback and recording works without any errors disernable to us. This suggests that our implementation works and allows the driver and device to work as though the device was local.

5.2 Non Volatile Memory Express SSD

NVME is a specification for PCIe based Solid State Disks. The specification [16] is open and there is a driver implementation in the Linux kernel. It was clear to us from testing the HDA audio codec that having access to both the driver and the specification was a huge benefit. This was large part of the reason we chose to test and develop with a NVME SSD. If we could get it to work, we would also be able to do some performance benchmarks. It would also be more suited for validation than the sound card as a test that reads and writes to a disk and validates the data is easy to create.

5.2.1 Device-driver interface

NVME SSDs have a single prefetchable BAR, supports both MSI-X and MSI and relies heavily on DMA operations. We have not spent time with the details of the NVME specification, but we

suspect that the driver programs requests into the device registers exposed by the BAR which the device performs using DMA.

5.2.2 Testing device

The streaming DMA feature described in section 4.7.2 was implemented using the SSD as it was the first to use this feature and it proved to be invaluable in the development of the mapping code as it helped us uncover multiple oversights and bugs. Unlike the HDA audio codec, which created static, long lived, DMA buffers, the NVME device driver rapidly mapped and unmapped memory during operation. The rapid mapping and unmapping pushed our DMA code to it's limits and it was clear that our code presented a performance bottleneck. Not only were the mappings frequent, but the driver unmapped memory in its interrupt handlers which we were not prepared for at the time. This lead to the implementation of our shared memory based communication channel described in section 4.8 as well as other interrupts context issues we discovered described in section 4.7.4. Finally, the driver also created more mappings than both we anticipated and the Dolphin driver supported. To solve this we had to modify the driver as described in 4.10.1.

5.2.3 Missing features

Our DMA implementation is still not completely interrupt safe as we need to call Dolphin software functions that are not interrupt safe. The Dolphin driver functions uses a non-atomic lock which is a problem when the lock is taken and it attempts to sleep and schedule the system in an interrupt context. We have not been able to avoid using all calls to these functions so this problem still exists.

The results is that the NVME SSD works with our implementation, *sometimes*, but most of the time, the kernel deadlocks or crashes before the disk can be used for anything useful. We have however been able to mount a partition on the disk and successfully run `ls` at least once, most of the time however, everything crashes before we are able to get this far. To solve this we either need to make the Dolphin driver interrupt safe, or implement the IOMMU based mapping described in section 4.7.3 which should be interrupt safe and also not require communication with the other host which we expect greatly decreases performance.

5.3 Intel Ethernet NIC

Intel network cards are widely available and have an open source driver in the Linux kernel. Compared to the audio codec it is however very complex, requires support for more features as well as high performance.

5.3.1 Device-driver interface

The Intel Ethernet cards we experimented with had multiple BARs, some prefetchable and others not. In addition, the cards supports both MSI-X and MSI, but the device driver for some of the cards only supported MSI-X while others supported MSI as a fall back. The contents of packets are transferred using DMA.

5.3.2 Testing remote device access

The first network cards we tested with was at the very beginnings of our implementation when we only supported configuration space accesses and MMIO register mappings. We were able to validate that the MMIO mappings worked, but at this point we had neither DMA support nor interrupts which is needed for correct operation. Only after both of these functions were implemented did we go back to testing network cards. Before they were implemented however, we debugged using the network card as it was the first device for which the driver loaded successfully, even though the implementation was not complete. So although network traffic did not work, the device showed up in user space interfaces such as `ifconfig` and it was correctly identified by the driver. At this point we discovered the program `ethtool` which has the ability to make a light on a NIC blink for easy identification. This feature worked for a remotely accessed network card and since the only features we had implemented was configuration space access and MMIO, it confirmed that the MMIO mappings were correct.

Later in the implementation, after implementing DMA, we discovered that the drivers for network cards we had tested for only used MSI-X, which we do not yet support, but we later found a network card where the driver used plain MSI. This was after testing the NVME SSD as well as the audio card, so most features were implemented. The driver for this network card however, required hundreds of small DMA mappings, far more than we had previously seen. The need for this amount of mappings lead to the development of the merging feature described in section 4.7.2 and the discussion on ways to increase the amount of mappings in section 4.7.3 and 4.7.2.

Even though we supported DMA, running `ifconfig ethX up` did not work as the driver were unable to allocate enough buffers. We however noticed that the amount of mappings required could be decreased by lowering the size of the ringbuffers for the device using `ethtool`. We were however unable to lower the lengths enough for our mapping limit to suffice, until we modified the driver to allow shorter queues. By lowering the queues to a length of 8, instead of the default of 256, `ifconfig ethX up` completed successfully. There was however a problem either sending or receiving frames (or both) as we were unable to ping any machine and we suspected that the queues were to short. At this point we believed that we needed to implement IOMMU based mapping to progress any further, but we were running out of time. When we looked at the addresses that we mapped or attempted to map but failed, we discovered that some of them are very close in memory. Since each NTB mapping always maps a full NTB page, 32MB in our case, we realized that some of them could be covered by a single mapping and that some of our current maps overlapped. We implemented functionality for detecting and utilizing this and were able to increase the size of the queues and progress without using an IOMMU.

5.3.3 Performance and validation

Our final implementation still had issues with the mappings and sometimes we were unable to map enough buffers if the buffers were to far away from each other. We also experienced very high packet loss which we suspect is due to failed mappings. We were however able to perform some simple benchmarks. First, we used a ping test to measure the latency of packets and compared it to the same device but used locally. Secondly, the tool `iperf` was used to measure bandwidth and packet loss. This device, was like the NVME disk prone to crashes, although a lot less. This, in addition to the issues we had with mappings caused the performance benchmarks to be hard to consistently perform.

We were able to perform a ping to another machine directly connected by cable. With

the device used locally, latency was around `0.25`. When the device was accessed using our implementation however, it increased to about `0.45`. We expect that this is not due to the latency incurred by the NTB, but the fact that the driver sets up new mappings for each packet sent. Since our mapping functionality is far from optimized and might need to communicate with the other host, we consider it a likely candidate for the increased latency. We also attempted to use `iperf` to measure bandwidth, but this seemed to cause more issues than ping. The packet loss varied from 0% to 100% and the bandwidth was at least sometimes in the order of tens of KB/s. We expect this to be caused by the inefficient mapping as well as the dropped packets.

5.3.4 Missing features

The most pressing issue with the network cards tested were the lacking mappings that like with the NVME SSD can be solved using an IOMMU. Hopefully, this can increase the performance to same same level as when the device is used locally. In addition, a working MSI-X implementation is needed for the more advanced network cards to function and we suspect that support for MSI-X can also be implemented with the help of an IOMMU although this is more uncertain.

Chapter 6

Conclusion

6.1 Summary

Our goal with this thesis was to look at PCIe multi-root solutions and to create our own using existing hardware. We started by looking at how PCIe works and existing multi-root solutions to provide the necessary background information for us to argue the need for our software based multi-root solution. After this we presented PCIe in the Linux kernel focusing on what is needed as background for our implementation.

The design chapter started by summarizing the technologies we tested while learning about the subject material. The design is described as a series of functionalities that is implemented as well as ideas that did not end up in our final implementation. The design that is described solved the main problem statement of this thesis by providing us with a working proof of concept.

6.2 Main Contributions

About half of the time spent on this thesis was used to familiarize ourselves with PCIe, Linux and NTBs. Through this we have gained not only the knowledge required for solving our problem statement but also general understanding of the Linux kernel and the PCIe architecture as a whole. More directly related to the design, we learned a great deal about how PCIe devices and their device-drivers communicate including MMIO, DMA and interrupts. Specifically we have learned about the low-level details of the memory operations involved including address spaces, caching systems and IOMMUs.

We believe that our device lending design provides a viable alternative to vendor multi-root systems and MR-IOV. We have shown this by providing a working proof of concept that provides the same functionalities as MR-IOV on most points and is superior in other areas. The table 6.1, compares our solutions to the alternatives. Most importantly, our solution has modest hardware requirements, especially if implemented using an Intel Xeon CPU with an NTB. The theoretical performance of our design is very close to other native multi-root solutions such as MR-IOV and locally attached devices. Our current implementation does not achieve such performance, but we believe that some of the alternate designs, especially IOMMU, can close this gap.

	MR-IOV	NTB design	NTB implementation
performance	native	native	low
device sharing	With MRA devices	With SR-IOV devices	✗
share upstream devices	✗	✓	✓
unmodified drivers	✓	✓	✓
unmodified applications	✓	✓	✓
software required	only for management	✓	✓
hardware required	MRA Switch	NTB / Xeon CPU	Dolphin NTB

Table 6.1: Comparison of PCIe device pool schemes

6.3 Future work

We believe that the most important work remaining for our design is performance enhancements as achieving near native performance is critical to the viability of our implementation. We also need to test more devices to verify the correctness of our implementation. In addition to this, there are some areas that require more research.

6.3.1 Multiple devices

One desired functionality that we did not implement or test is to have multiple devices remotely accessed at the same time, either from one machine or multiple in a cluster. We believe that an improved memory mapper is required for this to be feasible for most devices. The design itself should have no problem with multiple devices and this should be fairly easy to implement once the memory mapping has been improved.

6.3.2 SR-IOV device sharing

Sharing an SR-IOV device with multiple hosts has been shown to work [21] [22]. For us, the issues with multiple devices apply for SR-IOV as well, at least if more than one virtual device is desired. In addition, the only SR-IOV capable device we had access to was an Intel network card. The driver for this card, however, only supported MSI-X so we could not use it. Once MSI-X support is implemented, or the driver is modified to support MSI, SR-IOV should work without significant changes to our current implementation.

6.3.3 Improvements to the Dolphin driver

We consider increasing the mapping granularity using an IOMMU to be a feature that should be implemented into the Dolphin driver. If this functionality is implemented, our current implementation should be able to take advantage of it with minimal modifications. We have already described how we believe this should be designed in section 4.7.3. In addition, the current modification to the Dolphin driver that disables WC needs to be removed and replaced with an alternative.

6.3.4 Device control negotiation

To arbitrate the allocation of devices, some sort of software should be designed. By spawning our daemon and interacting with the sysfs interface of our kernel module, it should be able to assign a device to a host without needing to know the inner workings of our implementation.

We think that such software should provide user programs with the ability to query the available devices and reserve devices for a given time period or until returned. Research will be needed into how different drivers handle the frequent hotplug of devices this would cause. Especially interesting is if a user space CUDA program would be able to request or release CUDA devices while running.

6.3.5 Use in virtual machines

We have not attempted to assign one of our fake devices to a VM, but in theory it could work "out of the box". If so, the Linux kernel should be able to create a VFIO device for our remote device that the hypervisor can use to enable pass-through of the device. Another, more direct approach, is for our implementation to mimic the VFIO interface and create a VFIO device file which can be given to a VM, bypassing the Linux kernel and any difficulties it may present.

6.3.6 Isolating borrowed device

To increase security for the host lending away a device, an IOMMU should be used to isolate the affected devices from the rest of the system. This should prevent the other host from using the device as an attack vector or crashing the system if the device is misused.

Appendix A

Accessing the source code

The source code of our implementation can be accessed at `https://bitbucket.org/larsbk/ntb_test`. To gain access to the repository, contact the author at `larsbk [at] ifi.uio.no`.

The program `map_res` is used for mapping the BARs of a device by pointing it to the device's `/sys/bus/pci/devices/X/resource` file and mapping the same resources on the other machine. `configd` is the user space daemon that the kernel module connects to. `fake.c` is the kernel module that injects the remote device and implement the driver-side functionality. It communicates with the user space daemon. New devices are added by writing to `/sys/module/fake/control/create`.

Bibliography

[1] Linux Kernel. kernel.org.

[2] Nvidia GRID Shared Virtual GPUs (vGPU). www.nvidia.com/object/virtual-gpus.html.

[3] SISCI API Documentation. ww.dolphinics.no/download/SISCI_DOC/index.html.

[4] Steam Download Stats. store.steampowered.com/stats/content/.

[5] *Thunderbolt Device Driver Programming Guide*, 2013.

[6] D. Abramson. 2006.

[7] J. E. Bottomley. Dynamic DMA mapping using the generic device. https://www.kernel.org/doc/Documentation/DMA-API.txt, 2015.

[8] J. Corbet and A. Rubini. *Linux Device Drivers, 2nd Edition*. O'Reilly Media, 2001.

[9] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young. Computing as a discipline. *Computer*, 22(2):63, 1989.

[10] Intel Corporation. *High Definition Audio Specification*, 1.0a edition, 2010.

[11] Intel Corporation. Non-Transparent Bridge Enables Connectivity. www.intel.com/content/www/us/en/intelligent-systems/picket-post/xeon-c5500-c3500-non-transparent-bridge-paper.html, 2010.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2015.

[13] Intel LAN Access Division. *PCI-SIG SR-IOV Primer*. 2011.

[14] T. Iwai. MORE NOTES ON HD-AUDIO DRIVER. https://www.kernel.org/doc/Documentation/sound/alsa/HD-Audio.txt.

[15] J. Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[16] NVM Express. *NVM Express Specification*, 2014.

[17] PCI-SIG. *Multi-root I/O Virtualization and Sharing 1.0 Specification*.

[18] PCI-SIG. Single root I/O virtualization and sharing specification.

[19] PCI-SIG. *PCI Local Bus Specification*, 2002.

[20] PCI-SIG. *PCI Express 3.1 Base Specification*, 2010.

[21] J. Suzuki, Y. Hidaka, J. Higuchi, T. Baba, N. Kami, and T. Yoshikawa. Multi-root Share of Single-Root I/O Virtualization (SR-IOV) Compliant PCI Express Device. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 25–31. IEEE, Aug. 2010.

[22] C.-C. Tu, C.-t. Lee, and T.-c. Chiueh. Secure i/o device sharing among virtual machines on multiple hosts. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 108–119, New York, NY, USA, 2013. ACM.

[23] C.-C. Tu, C.-t. Lee, and T.-c. Chiueh. Marlin. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems - ANCS '14*, pages 125–136, New York, New York, USA, Oct. 2014. ACM Press.

[24] H. Wong. PCI Express Multi-Root Switch Reconfiguration During System Operation. Master's thesis, Massachusetts Institute of Technology, 2011.