UiO : Department of Informatics
University of Oslo

Virtualized SISCI

Extending Virtual Machines With a Shared Memory API Utilizing PCIe Networking

Halvor Kielland-Gyrud

Master's Thesis Spring 2017

Virtualized SISCI

Halvor Kielland-Gyrud

February 15, 2017

Aknowledgements

First of all, I would like to thank my supervisors, Håkon Kvale Stensland and Hugo Kohmann, for providing me with the opportunity to do this project and guidance along the way. Furthermore, I would like to thank Pål Halvorsen for giving me valuable feedback on my work. I would also like to thank the employees at Dolphin, particularly Lars Bjørlykke Kristiansen and Roy Nordstrøm, for insightful discussions and suggestions on different matters.

Last, but not least, I would like to thank Sarah for her patience and support throughout the process.

Abstract

Today, virtualization technology plays an important role in the computer industry. Virtual Machines (VMs) are deployed in a number of different scenarios to meet the requirements of a market that expects service delivery to be fast, dynamic and transparent. An area where VMs have been more cautiously used is High Performance Computing (HPC). In HPC use cases, there are often strict requirements to the performance throughout a system of several compute nodes. Traditionally, communication between VMs have been limited to standard networking mechanism such as TCP/IP. There are approaches for high performance Inter-VM communication (IVMC) between co-located guests, however, to the best of our knowledge, few solutions combine this with high bandwidth and low latency remote communication.

In this thesis, we propose a unified IVMC mechanism based on the Software Infrastructure for Shared-Memory Cluster Interconnect (SISCI) API. SISCI enables local and remote applications communicate through shared memory, which for the remote case is exposed through a PCI Express (PCIe) interconnect utilizing Non-Transparent Bridges (NTBs). In our proof of concept implementation developed for the Qemu/KVM hypervisor, we extend the SISCI API functionality to VMs, allowing them to seamlessly communicate through shared memory, regardless of whether they are co-located or reside on different hosts. Binary compatibility with existing SISCI applications is achieved, and no significant modifications have been made to Qemu/KVM or the guest operating system.

Our prototype shows that an approach based on SISCI allows for a near native memory bandwidth between co-located VMs. In addition, guest applications can utilize the performance potential of the underlying PCIe interconnect. Depending on the PCIe adapter card used, a remote bandwidth between 2.9 and 6.9 gigabytes per second is achieved, with a memory latency as low as 0.7 microseconds. This corresponds to an overhead of less than 1 percent when compared to the remote performance of native SISCI applications.

Contents

List of Figures

List of Tables

List of Code Examples

Chapter 1

Introduction

1.1 Background and motivation

As a concept, virtualization has been explored since the late 1960s. IBM was one of the first companies to release a virtualization solution for commercial use with their VM/370 OS in 1972. Later, several other companies and research institutions took interest in the technology. Because of decreasing hardware costs and the introduction of the x86 architecture by Intel in 1978, which would dominate the market and offered no support for virtualization, it was somewhat forgotten in the 1980s. Since the start of the 1990s, interest for virtualization has again been increasing, and in the last decade, it has become an important part of the computer industry.

Virtualization technology opens up a variety of possibilities, but primarily, it allows running one or more Virtual Machines (VMs) on a physical computer. A VM is a logical abstraction that runs its own Operating System (OS), but shares the underlying physical hardware on the host with other VMs and host applications. As a result, each VM can be tailored to specific requirements and deployed on demand without making changes to the underlying system. Compared to running a native system, VMs offer benefits such as increased resource sharing and efficiency through server-consolidation, and better isolation between the different tasks in a system. It also leads to a larger degree of fault-tolerance through checkpointing and VM mobility. For these reasons, virtualization has been embraced by the current market, where cloud computing has gained a dominant position, and service delivery is expected to be fast, dynamic and transparent. By 2014, it was estimated that around 70 percent of servers were virtualized [4], and today some prognoses show that the rate has reached 80 percent [5].

However, virtualization is not without drawbacks. It inherently adds a performance overhead compared to a native system, as VMs do not run directly on hardware, but rather through hypervisors. Hypervisors are software components that is responsible for creating, running and managing VMs. This adds a layer between the OS in VMs and the physical hardware, which impacts the overall performance negatively.

Throughout the years, virtualization technology has become more mature, and many of the performance limitations have been reduced by techniques such as hardware-assisted virtualization and paravirtualization. However, one area that traditionally has been, and still is, a bottleneck for VM performance, is Input/Output (I/O) [6]. In most scenarios, I/O must pass through the hypervisor or similar software that has direct access to hardware. For this reason, communication between VMs, dubbed Inter-VM Communication (IVMC), is often slow

compared to communication between regular machines, even if the VMs reside on the same physical computer. While efforts have been made to let co-located VMs use shared memory to communicate more efficiently, often the VMs are limited to using TCP/IP or similar standard networking techniques in remote scenarios. This has reduced the viability of VMs in use cases where high performance throughout the system is crucial. Typically, such applications fall under the High-Peformance Computing (HPC) definition, where high bandwidth and low latency are key to achieving the predefined goals.

Outside the virtualization area, however, a number of specialized networking solutions tailored to HPC exist. These could significantly increase the performance of IVMC if adapted to operate in a virtualized environment. One such specialized networking approach is Infiniband[1], which is commonly applied as a high-speed interconnect in HPC scenarios. Infiniband is tailored to remote communication between physical computers and employs Remote Direct Memory Access (RDMA) as a core mechanism. While Infiniband supports virtualized environments through Single-Root Input/Output Virtualization (SR-IOV), it is not optimal for co-located VMs, as communication must still pass through the full network stack [7].

Another approach is PCI Express (PCIe) based networking. PCIe is the successor to Peripheral Component Interconnet (PCI) and natively offers high bandwidth and low latency through its point-to-point topology. By utilizing PCIe adapters with Non-Transparent Bridges (NTBs), PCIe can be used as an interconnect between remote systems, allowing high performance communication through shared memory. If successfully virtualized and sufficiently lightweight, a solution based on PCIe could act as a high-performance IVMC mechanism, increasing the viability of running VMs in HPC scenarios. Dolphin Interconnect Solutions[2] (Dolphin) is a provider of both hardware and software that allows employing PCIe for networking purposes. They offer several software solutions to utilize the underlying interconnect, among others a shared memory API called Software Infrastructure for Shared-Memory Cluster Interconnect (SISCI). This API can be used to share memory both between local and remote applications, allowing communication through (R)DMA and Programmed Input/Output (PIO). Possible use cases for the API include any scenario where high bandwidth and low latency are required, for instance, it is currently employed in high-availability - fast failover systems and latency focused applications such as electronic trading.

Support for high performance IVMC solutions that allows both local and remote communication is scarce in the current market. A mechanism based on extending the SISCI API to VMs, would allow VMs to utilize local shared memory when they are co-located, and remote memory exposed through a PCIe interconnect when they are located on different physical computers. Potentially, such a solution could satisfy the bandwidth and latency requirements in HPC scenarios, and represent a unified IVMC mechanism that would increase the viability of VM deployment in such use cases.

1.2 Problem Definition

The scope of this thesis is to investigate the potential of a shared memory IVMC mechanism based on the SISCI API, that employs PCIe networking for remote communication. Our goal is

[1] www.infinibandta.org
[2] www.dolphinics.com

a unified, high performance mechanism, that allows VMs to communicate with both co-located and remote VMs in a seamless manner. In an attempt to realize this goal, we will:

- Investigate into how VMs can be extended with SISCI API functionality to let them utilize local and remote shared memory for communication purposes. This includes exploring alternative ways for VMs to access shared memory stemming from their hosts. The most suitable approach based will then be determined based on our goal of high performance.
- Implement a proof of concept of a new IVMC mechanism based on the findings. Optimally, the prototype should allow binary compatibility with existing SISCI applications, and otherwise conform to the API and the virtualization platform.
- Review performance aspects of the implemented mechanism to assess its viability when utilized by co-located and remote VMs, respectively. This includes evaluating memory bandwidth and latency, and the performance of synchronization between SISCI applications.

1.3 Limitations

There are a number of different virtualization solutions currently available, many of which use significantly different approaches to perform virtualization and run VMs. For this reason, our proof of concept is implemented for use with one specific open-source hypervisor solution, namely Kernel-based Virtual Machine (KVM) with Qemu for hardware emulation. As a consequence, our proof of concept is also currently limited to the Linux OS.

Furthermore, our implementation only supports a subset of the features offered in the SISCI API. Most of the omitted functionality is not necessary to assess the viability of our mechanism, as they do not limit the communication potential. We have, however, found it necessary to leave DMA mechanisms out of the current implementation. In addition, migration support is not present in our proof of concept. Although mobility is a major advantage of virtualization, the focus of this thesis is to explore how VMs can communicate more efficiently, and the implications of migration are so numerous that it falls beyond the scope of this thesis. For a similar reason, the security implications of our mechanism is only briefly touched upon.

1.4 Research Method

Overall, our research method corresponds to the *design paradigm* in the ACM classification [8]. We have proposed a set of requirements and goals that we have used to design and implemented a proof of concept. This prototype has been evaluated against our initial requirements. However, our design, implementation and evaluation phases have overlapped at times. The reason for this is that our goal of high performance has made us review different approaches in more detail before making a conclusion about their suitability.

1.5 Main Contributions

Our proof of concept shows that an IVMC solution based on shared memory over PCIe is a viable option in scenarios that require high bandwidth and low latency. Moreover, it proves that

shared memory mechanisms, such as the ones offered through the SISCI API, translates well to a virtualized environment when using the KVM and Qemu.

We achieve a memory bandwidth and latency that is more or less identical to the native SISCI performance, both when VMs access remote memory over PCIe and shared memory stemming from their host. Applications in co-located VMs can communicate through shared memory with little or no overhead, in practice only limited by the native memory bandwidth and latency in their host. The maximum overhead in local scenarios are just over 2 percent, while often staying around the 1 percent mark. Correspondingly, communication between remote VMs are able to utilize the full performance available through the underlying PCIe interconnect. With the x8 PCIe adapters primarily used in the project, this equals a bandwidth of around 2900 megabytes per second and a memory latency as low as 0.7 µs. Compared to native, the bandwidth overhead introduced in remote scenarios are less than 1 percent. Preliminary tests have also shown that a remote bandwidth of around 6300 MB/s is achievable when using newly released x16 PCIe adapters.

In both scenarios, the near native performance stems from remapping memory from the host to VMs through the device BARs of a Qemu virtual PCI device. We use a dynamic mapping procedure, where both local and remote shared memory is made available to VMs on demand, i.e., no memory is inherently shared. This avoids the need to define shared memory regions statically during VM startup, and allows variably sized segments.

The shared memory mechanisms is made available to VM applications through a slightly modified SISCI API. All interfaces of the latter has been kept intact, and only minor modifications have been performed on the API to take the virtualized environment into account. As a consequence, VM applications utilize the API as any other native application. This also allows binary compatibility with existing SISCI applications.

1.6 Outline

This thesis is structured as follows. In chapter 2 we will describe various topics relevant to the project, such as a more detailed look at virtualization technology. Chapter 3 comprises how we proceeded to create a design for our proof of concept, more specifically the approaches we considered and the choices we made during the process. In chapter 4 we describe the details of how we implemented our IVMC mechanism based on the shared memory mechanisms in the SISCI API. We evaluate our proof of concept in chapter 5, more specifically, we compare the performance of our mechanism to the one achievable in a native environment. In chapter 6 we make our concluding remarks about the project.

Chapter 2

Background and Related work

In this chapter, we introduce some subjects that are central to our goal of implementing a high performance IVMC mechanism based on the SISCI API. First of all, we look at virtualization in detail, describing various aspects of it, and techniques employed to realize virtualized environments. We then review some popular virtualization solutions that we have deemed suitable for our project. Moreover, relevant parts of the Linux OS is introduced, as this is our chosen development platform. In addition, we look at the PCIe, the features therein that allows sharing memory remotely, and the SISCI API and its features. Finally, we review some of the previous research that has been performed in the IVMC field.

2.1 Virtualization

As a concept, virtualization opens up a variety of possibilities. In the context of this project, the most important aspect is that virtualization allows creating *Virtual Machines* (VMs) that run on a physical computer. VMs are logical and exist only in software, but act as physical computers to the operating system (OS) running on them. By letting the VMs share underlying hardware resources, it is possible to concurrently execute multiple, and potentially different, OSes on one physical machine. In the following, we will describe different aspects of virtualization, the techniques employed to realize it and some of virtualization solutions that exist today,

2.1.1 Virtualization components

There are three main components in a virtualized system; the *host*, the *guest(s)* and the *hypervisor*. The host is the main OS of the physical machine that a VM is being run on. This machine is commonly referred to as the host machine. Guests are VMs being run on a host, while any OSes being run in VMs are called guest OSes.

The hypervisor is the component responsible for creating, running and managing VMs. For this reason, it is also referred to as the Virtual Machine Manager (VMM). Hypervisors can either run directly on hardware or run on top of an existing OS. The former is commonly called *type-1* or *bare metal* hypervisors, while the latter is referred to as *type-2* hypervisors. This distinction is not always clear; some hypervisors require an existing OS but run directly on hardware. While it is common for type-1 hypervisors to require modified guest OSes, there are exceptions. Type-2 hypervisors generally always allow running completely unmodified OSes in the guest.

In order to run guest OSes, VMs must offer a similar hardware interface to that of physical machines, to the OSes. This means that the Central Processing Unit (CPU), memory and Input/Output (I/O) must be virtualized. These issues will be introduced in sections 2.1.2 and 2.1.3.

2.1.2 Virtualization techniques

There are several approaches to virtualization. Commonly, three types are employed to run VMs; full virtualization, hardware-assisted virtualization and paravirtualization.

Full virtualization

Full virtualization allows running completely unmodified guest OSes. In other words, the guest OS is unaware that it is being virtualized. To do this, one must provide an environment for the guest OSes where they can execute as if running on a physical machine. This is done by virtualizing the CPU and memory and emulating I/O devices in software. In the following paragraphs, certain issues related to this process will be introduced.

To understand how the CPU is virtualized, it is appropriate to introduce the protection scheme in the x86 architecture. This scheme implements a number of protection "rings" used to control access to hardware resources and certain system functionality. There are four rings, ring 0 to ring 3, each corresponding to a privilege level. In ring 0, any CPU instruction can be issued and full access to hardware and system functionality is provided. It is commonly referred to as *kernel mode*. Rings 1 and 2 are rarely used, but ring 1 can be configured to allow access to certain hardware, like I/O devices. Ring 3 is the least privileged level and referred to as the *user mode*. Here, direct hardware access is restricted and a number of instructions can not be executed. These instructions are restricted to kernel mode and are typically called privileged instructions. OSes takes advantage of these modes to enforce security. The kernel of an OS must be able to execute any instruction to perform memory management, device I/O and other system critical tasks, and therefore runs in kernel mode. User applications, on the other hand, should have limited control over the system and run in user mode. This is, however, usually also the case for guest OSes. This means that while the kernel of the OS expects to be running in ring 0 and have full privileges, it does not. Any privileged instructions from the guest OS will cause a protection fault if issued. This scenario can be handled relatively easy by letting privileged instructions from the guest trap to the hypervisor. The hypervisor can emulate the effects of the privileged instruction and return control to the guest. What complicates matters is that some instructions are privileged in nature, but does not cause an exception when executed in user mode, rather they fail silently [9]. Such instructions are referred to as *sensitive* or *critical* instructions, with an example being the *popf* instruction which modifies the *FLAGS* register of the CPU. To be able to run an unmodified guest OS, one therefore need a mechanism to allow such instructions, while keeping the protection scheme intact. One of the approaches to this problem is to use *binary translation*. This is a technique where the non-virtualizable, privileged instructions are dynamically intercepted by the hypervisor and translated to code that can be run directly on the CPU [10]. The solution was popularized by VMWare Inc. in the 1990s, and initially creates a significant overhead, but performance has been helped by using various caching schemes. An overview of full virtualization with binary translation can be found in

figure 2.1. Later, the problems related to privileged instructions were more or less solved by the introduction of virtualization extensions in x86 CPUs (see section below).

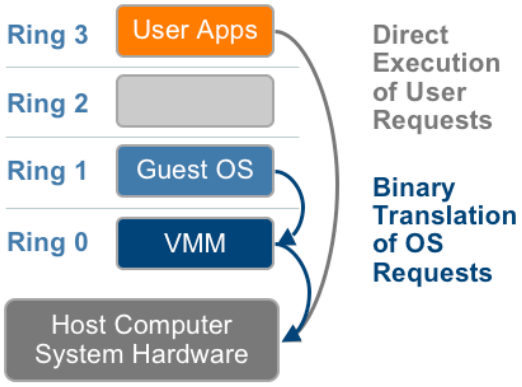

Figure 2.1: Full virtualization with binary translation [1]

Memory management is another area where virtualization introduces complexity. Since the guest kernel (normally) runs in user mode, the memory which the kernel regards as physical memory is in reality virtual memory on the host. This means that an additional translation scheme is required to perform correct memory accesses. A software technique that is commonly used for this purpose is *shadow page tables*. Shadow page tables reside in the hypervisor and map the virtual pages of the guest to the actual physical pages that were allocated by the hypervisor [11]. The host Memory Management Unit (MMU) (see section 2.3.1) then performs lookups in these page table when translating, meaning that correct mappings are achieved while the guest remains oblivious to the fact that it does not deal with real physical addresses. The guest does still have its own page tables however, which need to be synchronized with the shadow page tables. Since the page tables of the guest reside in userspace, there is no inherent mechanism that makes the host aware of changes to the page tables. To handle this problem, various techniques exist, all which create overhead compared to page table management in a non-virtualized environment. A common approach is to make the guest page tables *read-only*, which causes a trap to the hypervisor upon guest modifications. Other hypervisors rely on different techniques, of which some will be introduced later (see section 2.2.2).

I/O virtualization is initially not as complicated as virtualizing the CPU or memory. When using full virtualization, I/O devices are traditionally emulated in software, which can be done in a relatively straightforward manner. Usually, the emulated hardware presented to the VMs are well known generic devices, for compatibility reasons. I/O requests are trapped by the hypervisor and translated there or in host userspace, in order to conform to the actual physical hardware. If any Direct Memory Access (DMA) capable devices are to be emulated, it is important the implications of DMA are handled. DMA allows devices to transfer data directly to or from memory, without involving the CPU. Since absolute addressing is used, a DMA transfer will fail, or worse, allow the VM to access memory regions it should prohibited from, if initiated by a guest OS, and no technique is in place to translate guest physical memory addresses. Such translation can be done in software with a performance penalty, but it is important to have mechanisms that ensure proper isolation is in place.

Regular emulation of I/O devices can introduce a significant overhead compared to the performance of native I/O. We will introduce alternatives to emulation in section 2.1.3.

Hardware-assisted virtualization

Hardware-assisted virtualization (HVM) involves using hardware features to aid the virtualization process. Because of the lack of hardware support in the x86 architecture, the technique was not widely used until Intel and AMD introduced virtualization extensions in their CPUs. Intel released the first processor with their *VT-X* technology in late 2005, and AMD followed with *AMD-V* in the beginning of 2006. Although the two technologies are not compatible with each other, they introduced similar functionality, mainly by adding a new CPU mode; the *guest mode*. Guest mode is similar to user mode with the exception that one can choose to trap certain instructions issued from it [9]. By setting up the VMs to run in guest mode, any privileged instructions from the guest kernel can thereby be handled by the hypervisor. Normally, when the CPU is faced with a privileged instruction in guest mode, it simply returns control to the hypervisor, which handles the instruction. Figure 2.2 provides a basic overview of this approach. Virtualization support in x86 CPUs means that techniques such as binary translation are no

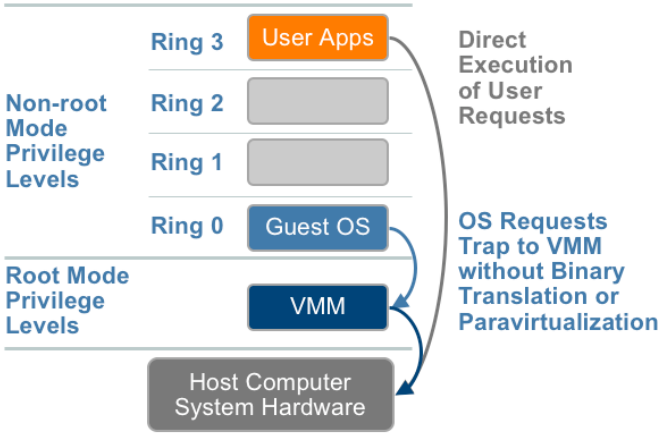

Figure 2.2: Overview of hardware-assisted virtualization [1]

longer necessary to run unmodified guest OSes, and most solutions that use full virtualization take advantage of these features today.

In 2007 and 2008, hardware support for virtualizing memory followed. Intels solution is called *Extended Page Tables* (EPT), while AMD named their solution *Rapid Virtualization Indexing* (RVI) (originally Nested Page Tables (NPT)). Both approaches introduce an additional set of page tables that contain mappings from the physical addresses of the guest to actual physical addresses on the host. The hypervisor is responsible for setting up and maintaining these mappings. When a lookup is performed to initially translate a guest virtual address into a guest physical address, an additional page walk is performed by hardware in the second set of page tables. By doing this, the corresponding physical address on the host can be found without requiring software techniques such as shadow page tables, at the cost of an extra page walk.

Later, both Intel and AMD have introduced hardware that acts as an Input-Output Memory Management Unit (IOMMU). Intel's technology is called VT-d, and adds what they refer to as *Directed I/O*, while AMDs version is named AMD-Vi. Similar to how regular MMUs perform

address translations for the CPU, IOMMUs translate addresses on behalf of devices. In a virtualization context, this allows translating guest addresses into physical addresses on the host, when devices perform DMA requests initiated by a guest. Thereby, problems related to DMA operations in virtualized systems are mostly solved.

Paravirtualization

Introduced by the Denali project [12] and popularized by Xen [13], paravirtualization (PV) is a technique which leverages some of the original drawbacks of full virtualization. It does this by requiring that guest OSes are modified to be aware that they are being virtualized. The hypervisor presents a similar, but not identical, interface to that of the underlying hardware [13], which the guests uses to communicate with the hypervisor through so called *hypercalls*. Many of the problems related to virtualization of the CPU and memory management may therefore be avoided. For instance, non-virtualizable, privileged instructions are replaced by calls to the hypervisor, as shown in figure 2.3. Device I/O on the other hand, are be simplified and sped

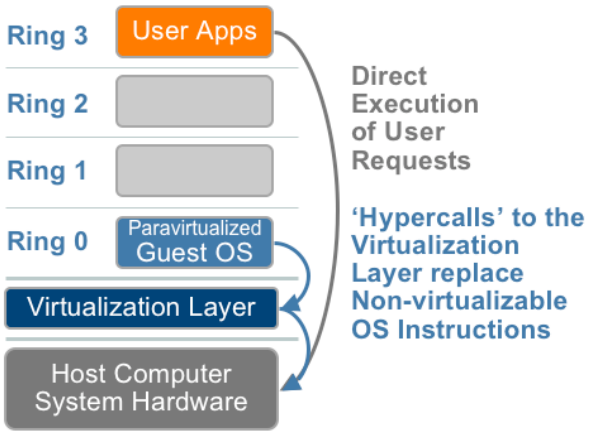

Figure 2.3: Overview of paravirtualization [1]

up by letting the guests and the hypervisor communicate more efficiently. Paravirtualization leads to reduced overhead and improved performance, but is not without drawbacks. Requiring modified guest OSes means that the guest OS either must be open-source or inherently support paravirtualization. Additionally, deployment and maintenance may be more complicated.

In addition to paravirtualizing all hardware, it is possible to use paravirtualization in conjunction with full virtualization or hardware-assisted virtualization. Typically, hardware extensions are used for virtualizing the CPU and memory, while device I/O is paravirtualized for improved performance. Support for this approach is present in many common OSes, and hypervisors that traditionally use full virtualization.

2.1.3 I/O virtualization

Since our project is related to communication between both local and remote guests, a more thorough introduction of different I/O virtualization techniques is in order. As mentioned, the drawback of regular I/O emulation is that it introduces significant overhead. The hardware interface of emulated devices must be kept intact in the guests, and device drivers must be

able to perform I/O operations on the devices identically to how they would perform them on their physical counterparts. In a virtualized environment this does not lead to effective communication between guests and the host component responsible for emulating the device.

As a consequence, *paravirtualization* is frequently used also when virtualizing I/O. By using frontend drivers in the guest and backend drivers in the host that are not limited to the hardware interface of the emulated device, significant performance gains can be achieved. This does entail modified device drivers in the guest, but it is common for modern OSes to include this by default, which means that minimal effort is required by the end-users. A popular solution for paravirtualizing I/O is the *Virtio* standard, released by Russell in 2010 [14]. Virtio is in essence a framework that provides an abstraction of a set of commonly emulated devices. The goal of the framework was to create a fast, general virtual I/O mechanism with cross-platform capabilities [14]. This is achieved through a structure similar to the in one used in Xen (see section 2.2.1). The hypervisor exports interfaces for the devices, which when implemented, creates the frontend drivers in the guests. These communicate with the backend drivers in the hypervisor through *vrings*; ringbuffers which can contain commands and data. Currently, backend drivers and interfaces for five types of devices exist. This comprises block devices, network device, PCI devices, a balloon for memory management and a console driver.

For PCI and PCIe devices, (see section 2.4) one can also employ *PCI-passthrough*. Passthrough involves giving a guest direct access to a hardware device, meaning no emulation is performed and standard device drivers can be employed. This potentially offers native performance but is not without drawbacks. By default, regular passthrough involves assigning a single guest exclusive control over the device. As a consequence, no other guests, nor the host, may utilize the device while it is assigned to a particular guest.

While regular passthrough may be acceptable in certain scenarios, it is also possible to use *Single Root I/O Virtualization* (SR-IOV). SR-IOV is a part of the PCIe specification, and describes how PCIe devices can be shared between guests using *multiplexed-passthrough*. In short, this involves *virtual* device functions, which may be specified in addition to the physical device functions that normally represent each PCI device and its functionality. In practice, virtual function acts as lightweight device with similar properties as their physical counterpart, mediating access to the latter. As a result, these functions can be assigned to individual guests without limiting access to the physical device to one guest. Devices that are to be used with SR-IOV must explicitly support it by defining such functions, with an upper limit of 256 per device. In addition, SR-IOV must be supported by the hypervisor. While some hypervisors allows regular passthrough without hardware support, most require an IOMMU to e.g. correctly handle DMA transfers, and for SR-IOV, IOMMUs are an absolute requirement.

We have now given an overview of virtualization technology, including different techniques employed to virtualize guests. In the next section, we will look at some of the virtualization solutions currently available.

2.2 Current virtualization solutions

In today's market, several virtualization solutions are available. In the following, some the most popular enterprise solutions will be introduced briefly. At the time of writing, Microsoft and VMWare are the two largest suppliers of proprietary virtualization technology. Microsoft's

main product is called Hyper-V and is a type-1 hypervisor. It does, however, rely on existing Microsoft software in the virtualization process. VMware has several products for both desktop and server environments. Among its enterprise solutions are VMware ESX and VMWare ESXi, both regular type-1 hypervisors. Their desktop offerings are generally type-2 hypervisors.

As for open-source solutions, the most popular choices are the Xen Project (Xen)[1] and the Kernel-based Virtual Machine (KVM)[2], both based on Linux. Xen is currently the only open-source type-1 hypervisor, and was the first hypervisor that employed paravirtualization. KVM, on the other hand, is a loadable kernel module that converts an existing Linux distribution into a hypervisor. Whether KVM is a type-1 or type-2 hypervisor has been debated, but it is commonly viewed as a type-2 hypervisor.

For this thesis, Xen and KVM have been considered as the most viable hypervisor alternatives. The reason is that they are open-source and have broad support in the Linux community. During the development of our prototype, it will be a significant advantage to have access to the source code of the hypervisors to understand requirements and for debugging purposes. It is also possible that parts of the chosen hypervisor will need to be modified in order to use the SISCI API, which in practice is only possible if the hypervisor is open-source software. This makes solutions based on Linux a natural starting point. In the following, Xen and KVM will therefore be looked at in more detail.

2.2.1 Xen

Xen was originally released in 2003 by Barham et al. [13], after being developed at the University of Cambridge. At the time, virtualization support in CPUs did not exist, and the developers turned to paravirtualization to achieve their goal of implementing a virtualization solution with good performance and isolation.

To virtualize guests with Xen, two components are required; the hypervisor and a dedicated VM referred to as *domain0* (dom0). The hypervisor is booted into upon startup and is the most privileged entity in the system. Its main responsibilities are setting up the initial environment needed for virtualization, and to handle various requests from the VMs. While CPU and memory management, such as scheduling and page table operations, are handled by the hypervisor itself, I/O requests are passed on to dom0 (see below). This was a design choice by the developers, as they wanted a minimalistic a hypervisor as possible [13].

Dom0 is a privileged VM that is loaded by the Xen hypervisor immediately after initial booting has completed. It runs an existing Linux OS, which must support acting as dom0 or have been modified to do so. The kernel of dom0 executes in ring 1, which means it has direct access to hardware, and in some ways it can be seen the host OS in the Xen system. It is responsible for setting up and managing the regular VMs and providing them with I/O access, the latter meaning that device drivers for hardware reside in dom0 and not in the hypervisor. The regular guests are contained in VMs referred to as *domainUs* (domUs). In order to interface against the Xen hypervisor, the guest OSes must have a paravirtualization-enabled kernel and paravirtualization-enabled drivers. In effect, it means that modified guest OSes are a requirement to be paravirtualized with Xen. As with dom0s, the kernel of guest OSes execute in ring 1, while their userspace run in ring 3.

[1] www.xenproject.org
[2] www.linux-kvm.org

Xen uses paravirtualization to virtualize both the CPU, memory and I/O. Privileged and sensitive instructions in the domains are replaced with hypercalls, while a scheme called *direct paging* is used to handle the memory implications. This involves having the mappings from guest physical addresses to host machine physical addresses reside in the actual hardware page tables. Guest OSes are responsible for creating their own page tables but must register them with Xen after the initial allocation. Afterwards, they only have read access to the page tables; any updates must be done through the hypervisor to allow validation of any changes.

To provide I/O access for guests, a split driver-model is employed. This comprises frontend drivers in domU and backend drivers in dom0. I/O requests from the guests are communicated from the frontend drivers to the backend drivers in dom0 through a shared memory ring buffer, with the hypervisor only validating the requests. I/O is then performed in dom0 on behalf of the domUs, using the actual drivers. Instead of using hardware interrupts an event mechanism is used for notifications [13].

While paravirtualization still is the default mode of Xen, it is possible to run fully virtualized guests by using hardware extensions in the CPU. Support for both CPU-related and memory-related extensions are present, which enables using unmodified guest OSes in domU. In this mode I/O is emulated in software with the aid of Qemu [15], which we will describe further in the next section. One can also use full virtualization in conjunction with paravirtualization-enabled drivers (PVHVM), to increase the I/O performance. In addition, Xen offers support for mode referred to as *ParaVirtualized Hardware* (PVH). The goal with PVH is to exclusively use PV-enabled drivers and hardware extensions, the advantage being that paravirtualization-enabled drivers also are employed for booting.

Xen's paravirtualized approach initially limited the number of Linux distributions that could be used as guest OSes or run in dom0. Either they had to inherently support the roles, or they would need modifications in order to do so. It also meant that after a new release of a distribution or an update to the Linux kernel, modifications had to be made in order to make them compatible with Xen again. Today, most distributions, with the notable exception of newer Red Hat releases, can act as dom0 or be run as a guest OS. In addition, support for both dom0 and guest OS has been included in the Linux mainline tree since the release of version 3.0.0. This means that one of the early drawbacks of using Xen is now virtually non-existent.

2.2.2 KVM and Qemu

KVM was developed at Quamranet and released by Kivity et al. [9] in 2007. As mentioned, it is a loadable kernel module that converts an existing Linux distribution into a hypervisor. The hypervisor consequently runs directly on hardware but requires an existing OS. KVM exploits the similarities between a hypervisor and an OS; OSes are used for running applications and a hypervisor is used to run VMs. By extending Linux to act as a hypervisor and letting the VMs be regular Linux processes, existing functionality in the Linux kernel such as scheduling, can be reused for virtualization purposes. In addition, any relevant modifications or optimizations performed on the Linux kernel, may be easily integrated into KVM. It is also claimed that KVM can utilize any hardware supported by Linux [9]. KVM was developed after the introduction of Intel VT-X and AMD-V and requires these extensions. While its standard mode is hardware-assisted virtualization with emulation of I/O, it does support paravirtualization of the latter.

In order to use KVM to run VMs, two components are needed; KVM and a host userspace component that emulates hardware. When installed KVM exposes a device file or node called

/dev/kvm. This node acts as a regular character device and through its interface the userspace program can set up and run VMs by using different *ioctl*s. More specifically, VMs can be created and allocated memory to, virtual CPU registers can be read and written, and interrupts may be injected into the virtual CPU [9].

By default, the userspace program utilized to emulate hardware is Qemu [15]. Originally introduced by Bellard in 2005, Qemu is by itself a type-2 hypervisor capable of emulating a number of different CPU architectures and a significant amount of hardware. As a result, Qemu does not need KVM to perform virtualization, but commonly uses the features provided in KVM to accelerate guests. The way KVM and Qemu utilize each other has led to somewhat symbiotic relationship, where Qemu is focusing more on emulating hardware and providing I/O mechanisms, while KVM has continued to exploit the advantages that hardware extensions offers. By default, Qemu emulates I/O devices, providing a range of more and less generic device interfaces and mechanisms to guests. It does, however, also support paravirtualized I/O through the Virtio standard, implementing all the device interfaces offered by the framework.

In addition to providing I/O to KVM guests, Qemu initially sets up the guests' physical memory and calls the KVM kernel module to continue execution in the CPU's *guest mode*. Execution of guest code then continues until an I/O instruction is issued or an external event, such as incoming network traffic, triggers a signal. When exiting from guest mode into the kernel, referred to as a *VM exit*, it is checked what caused the CPU to exit guest mode. If the reason was an I/O instruction or a signal, the kernel exits to Qemu which performs the I/O request or handles the signal. If the reason is a shadow page table fault or an external interrupt, the KVM performs the necessary operations before the guest is allowed to execute once again [9].

To virtualize memory, KVM can utilize either Intel EPT or AMD NPT, if such technologies are present in the CPU. If not, a solution based on shadow page tables is used. Synchronization between guest page tables and the shadow page tables is done by exploiting the fact that the Translation Lookaside Buffer (TLB) must be updated whenever the guest makes any changes to its page tables. In addition, some optimizations are in place to improve performance, most notably guest page tables are cached across context switches [9].

Since both the hypervisors we consider most viable for this project are based on Linux, a more detailed introduction of the OS is in order. In the next section, we will therefore look at parts of Linux that are relevant for our goal of a high performance IVMC mechanism based on shared memory.

2.3 Linux

Linux is today one of the most popular open-source OSes available. It has been widely adopted for enterprise use and, especially as of late, has seen an increasing popularity with end-users. We have chosen Linux as the development platform for this project. With it being open-source, we have access to the source code, which is important for understanding any OS mechanism that might be relevant to realize our project. and lets us perform modifications where necessary. Linux is also well documented and has an active developer community where advice can be sought on a variety of technical matters. In the following, we will introduce some general concepts and parts of Linux that are relevant to our project, starting with how memory

management is performed.

2.3.1 Memory management

Like most modern OSes, Linux utilize *virtual memory* or *virtual addressing* to perform memory management. In short, this involves using virtual addresses to access physical memory, rather than using the physical addresses directly. Each process in the OS is presented with their own virtual address space, where an arbitrary address may be mapped to a physical one. When a process accesses a memory location, the virtual memory address is translated into a physical address, if a mapping exists for the given virtual address in the process' address space. Mapping are thus performed on a per-process basis, where two identical virtual addresses may point to different places in physical memory, depending on the process. The mappings are set up by the OS and used by hardware upon memory accesses, which we will describe in more detail below.

Paging

Both the virtual address space of a process and physical memory are divided in units called *pages*. A page consists of a relatively small block of contiguous memory, typically 4 kilobyte (KB). The relationship between virtual and physical pages is that the former is backed by the latter when mapped to physical memory. For this reason, physical pages are often referred to *page frames*, in that they contain virtual pages, while virtual pages are simply referred to as pages. Dividing memory in such units facilitates fine-grained control over memory. Memory that appears contiguous to a process might in reality be backed by page frames scattered in physical memory, or it may currently not reside in physical memory at all. The latter involves a technique called *paging* or *swapping*[3]. Paging allows that the memory of a process, specifically one or more pages, can reside either in physical memory or on secondary storage such as a hard drive. The latter is commonly referred to as a *swap area* since pages are swapped to this area when evicted from a page frame. This lets the OS reuse physical memory that has not been utilized in a long time, and allows that the total amount of memory used by different processes can be larger than the amount of memory that is physically available. Note that the kernel itself is never swapped out, it is always resident in physical memory ([11], p. 813).

To manage page frames, Linux employs the `struct page` data structure. These structures contain information about one page frame, more specifically how it is currently being used. Included in the structure are fields used to determine if the page frame is in use and if so, which mapping is currently using the page frame, and how many references there is to the page. If a page frame's reference count is decreased to zero, it is put on the OS' list of free pages, and subsequently reused. In addition, there are members used when evicting the page that is currently held in the page frame. This latter is done on-demand through a *Least-Recently-Used* (LRU) based mechanism, which in short scans for page frames that has not been used recently and swaps the corresponding page to disk if memory pressure requires it.
Virtual memory areas on the other hand are represented through `struct vm_area_struct` data structures. We will not go into the details of this structure here, only mention that they contain information about a contiguous memory area in an arbitrary virtual address space.

[3]Not to be confused with the older OS technique where the whole address space of processes were swapped out from memory.

Page Tables and the MMU

Whether a page currently resides in memory, or in the swap space, is completely transparent to the process that owns it. Upon accessing a virtual address, the CPUs *Memory Management Unit* (MMU) is responsible for translating the given address into a physical address. The MMU is a piece of hardware, typically located in the CPU itself, that in short resolves such mappings by looking at *page tables*. Page tables are data structures set up by the OS that contain information necessary to translate processes virtual addresses into physical addresses.

Traditionally, page tables have been implemented as a two-level scheme on 32-bit architectures. Each process has a *page directory* that contains page tables, which in turn contain *page table entries* (PTEs). Both the page directory and one page table has 1024 32 bit entries, where one page directory entry points to a page table, and one PTE contain information about one page. As a result, each page directory and page table has a size of 4 KB, which means they can fit into a single page frame. When translating a virtual address, the MMU uses the upper 20 bits of the address to index into a page table entry. The 10 most significant bits identifies a page directory entry pointing to a page table, while the 10 next bits are used to locate the correct PTE for the virtual address in the resulting page table. This process of subsequent lookups is commonly referred to as *walking the page tables*. When a virtual address' page is resident in physical memory, the PTEs contain information about the physical address of the corresponding page frame. More specifically, the upper 20 bits of the PTE represents a page frame number (PFN), which when combined with the last part of the virtual address is used to locate the correct physical address. In the opposite case, the page contents resides on secondary storage, and the PTE contains an address to the swap area where the page can be found The additional 12 bits of PTEs are used for various purposes, such as indicating whether the page is present in physical memory, the page permissions, e.g., read and/or write, if the page has been accessed and whether it belongs to user or kernel space. How address translation is performed with a two-level scheme on 32-bit architectures, using 4 KB paging, is illustrated in figure 2.4.

The two level approach described above are commonly extended with another level of page directories to implement support for 64-bit architectures. This was the case with Linux up to the 2.6 version of the kernel, while today Linux employs a four-level scheme. The overall approach is the same, adding yet another middle layer of page directories, and using 512 entries per level instead of 1024.

If the MMU finds that the page is not backed by a page frame upon a virtual address lookup, e.g., it has been swapped out, a *page fault* occurs. The OS' *page fault handler* must then locate the page on the swap space, find a free page frame and copy the page contents into the frame. In addition, it must modify the page's PTE so that it points to the new page frame number and reflects that the page is now present in physical memory. After these operations the translation is performed again by the MMU using the new mapping. While the page directory and page tables are located in physical memory during translation, MMUs often employ a *Translation Lookaside Buffer* (TLB) to speed up the translation process. A TLB is a small, fast cache that contains the most recently accessed PTEs, and during translations this cache is initially checked to see if a valid mapping for the virtual address can be found. If one is found, the translations is performed using the cached PTE, while in the opposite case a *TLB miss* has occurred and the page tables must be walked as normal.

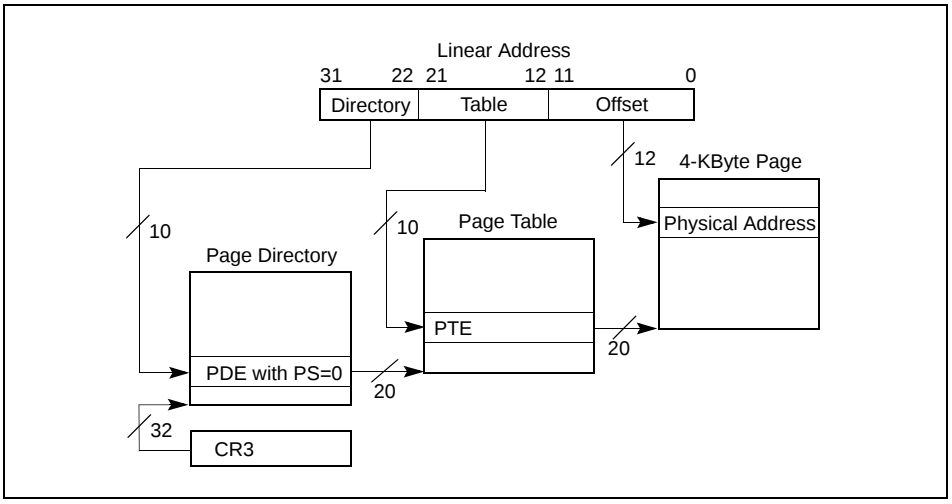

Figure 2.4: Translating a virtual address on 32-bit architectures using 4KB paging [2]

2.3.2 Device drivers in Linux

Device drivers are software that OSes use to communicate with devices. Through drivers, the OS can utilize the capabilities of the corresponding devices, while applications can use them to issue I/O requests. In Linux, device drivers are implemented as loadable kernel modules, which means that they have full access to hardware and can be loaded on-demand when required to handle a device. This makes adding devices to a Linux system easy and dynamic. Rather than requiring modifications to the kernel itself, it can be extended with functionality that handles any devices that are added to the system. From a general perspective, Linux differentiates between three type of devices; *character* devices, *block* devices and *network* devices. Character devices are typically devices that can be accessed to like a regular file, in that a byte stream can be written to or read from it, while block devices are accessed based on addressing certain parts of the device, e.g. a hard drive where specific blocks can be read upon request. Network devices, on the other hand, resolves around sending and receiving data packets. As a result of this classification, device drivers are divided into the same three categories, each with its own standard interface that a driver of the specific type must implement.

With the exception of network devices, all devices are represented as *special files* or *device files* in the Linux file system. Through these files, which are usually located in the `/dev/` directory, applications can communicate with the device driver, and ultimately with the device itself. This is done through the interface of the device driver type as described above, for instance character device drivers implements the interface defined in the `struct file_operations` data structure seen in figure 2.1. As the figure shows, standard operations for character devices include `open`, `read` and `write`, as possible on any regular file. Note that all parts of the interface need not be implemented. As a result, character device drivers are generally used for range of different devices that it may not be applicable to perform a `read` or `write` operation on. Instead, the `ioctl` operation is commonly used by applications to communicate various requests to a driver. This is a generic operation that tells the driver that some type of work needs to be performed. The operations that can be requested varies with each device and driver, and typically, each invocation contains a *command* code which the driver can use to determine the type of request.

```
struct file_operations {
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t,
           loff_t *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned
           long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
};
```

Code example 2.1: A cut down version of the file operations data structure

Identification of devices is done by a pair of *major* and *minor* numbers associated with each device file. The major number is used to determine which driver is responsible for handling the device, while the minor number differentiates between multiple devices handled by the same driver. As an example we can use a device driver handling hard drives. The identical major number of two different hard drives indicates that they are handled by the same driver. Upon receiving a device request the driver can determine which hard drive to operate on by looking at the minor number of the device file that the request originated from.

2.3.3 Communication with I/O devices

Various mechanisms can be utilized to communicate to communicate with I/O devices, The most widely used mechanisms are Port Mapped I/O (PMIO), Memory Mapped I/O (MMIO) and DMA. PMIO is an older mechanism where device resources are mapped into a separate I/O address space, and operations are be performed on them with special instruction such as `inb` and `outb`. As it is of little relevance to this project, we will not discuss it further.

MMIO, on the other hand, involves mapping device resources into the same physical address space that is used to access memory. This allows the CPU to use regular memory instructions such as load and store to operate on the resource, commonly referred to as Programmed Input/Output (PIO). As a result, communication with I/O devices is simplified and CPUs can be made less complex. Typically, a portion of the physical address space is reserved for MMIO to avoid collisions with physical memory. On 32-bit systems this reserved space is somewhat limited, as the CPU can only address up to 4 gigabytes (GB). The introduction of 64-bit architectures, however, have all but solved this problem. As a consequence, the drawbacks of MMIO are becoming less significant. Both registers, and device memory that acts as regular RAM, can be mapped as MMIO. Devices generally contain information about the type of resources they have to allow BIOSes and OSes to set aside the required amount of address space. PCI devices for instance, has a bit for each resource signaling how it should be accessed (see section 2.4).

In Linux there are a number of functions that can be invoked to remap MMIO areas for PIO access. The core function is `ioremap`, which remaps the area into kernel space. This

function is typically employed by device drivers to gain access to the registers and memory it uses for device communication. While direct access to MMIO resources often is limited to the kernel, it is also possible to map them to user space. This can be done through the Linux function `remap_pfn_range`, which remaps a range of physical addresses on a per page basis to a process' virtual address space[4]. Typically, drivers offer this possibility through the `mmap` operation of their interface, which applications can invoke to gain direct access to devices. When remapping resources both into kernel and user space, it is important to take the resource characteristics into consideration. Accesses to device registers generally have side effects, while other resources behave as regular RAM and can be accessed with optimized functions such as `memcpy`. We will discuss some issues related to this in section 2.3.4 below.

MMIO can be utilized in a number of ways, often specific to each device. One commonly employed technique is to write to a device register to indicate that a request must be handled, while putting any data relevant to the request in device memory.

As touched upon in section 2.1.3, DMA is an alternative to PIO where the CPU is not involved in data transfers. While various DMA techniques exist, they all involve a *DMA controller*. DMA controllers are hardware that may be programmed to initiate and handle data transfers between devices and physical memory, and in some cases, between two memory locations. The controllers may either be system-wide, i.e., be a part of the chipset, or reside on the devices themselves. Compared to PIO, the advantage of DMA is that the CPU can perform other tasks while waiting for data transfers to finish. A driver may initiate a DMA request to copy data from the device to memory, and in the meantime, handle other requests related to the device. Linux has its own DMA Application Programming Interface (API) that can be employed by drivers to set up and perform DMA. As the API is quite large and complex, involving a number of different mechanisms, we will not present it here, but rather introduce parts if and when necessary.

Another way to communicate with devices, or rather a way that devices can communicate with its drivers, is through *interrupts*. Interrupts are a means to let devices notify others about certain events, for instance, the completion of a write operation performed by a hard drive. As the name implies, interrupts represent an asynchronous notification mechanism; instead of a driver waiting for a device operation to finish, it can continue doing other work and rely on receiving an interrupt upon completion.

Traditionally, *pin-based interrupts* have been utilized to implement this mechanism. In this scheme, devices can assert a physical pin on the CPU, often through an interrupt controller that groups device interrupts together. The latter allows multiple Interrupt Request (IRQ) lines to be implemented and allocated to different devices, while only using one pin on the CPU. Commonly, 16 lines are available, which if required, must be shared between different devices. Upon receiving an interrupt, the CPU jumps to code responsible for handling the interrupt, commonly called an *interrupt handler* or an *Interrupt Service Routine* (ISR). These exist on a per device or per driver basis, and must retrieve information from the device to determine what kind of event happened and how to handle it. In the case of shared interrupt lines, it must also check if an interrupt actually happened on its device. In Linux, a driver can request an interrupt line and register an interrupt handler for a device by invoking `request_irq`.

[4]This function can also be employed to remap physical memory allocated in the kernel.

An alternative to pin-based interrupts for PCI(e) devices is *Message Signaled Interrupts*. In short, this mechanism realizes interrupts by having devices write to a special I/O address, which correspondingly is used to interrupt the CPU. MSI offers several advantages over regular interrupts, such as an increased number of interrupt line equivalents, called *vectors*, and prevention of certain race conditions that may happen with regular interrupts. While MSI allows a total of 32 vectors, the extension MSI-X allows 2048 to avoid that interrupts are shared between devices.

2.3.4 Caching

When accessing physical memory, all modern CPUs employ various *caching* techniques. The reason for this is that accessing physical memory is relatively slow compared to the speed that CPUs can operate on. In short, caching involves keeping recently accessed data and instructions that the CPU believes will be used soon, in smaller hardware caches located closer to the CPU. There are typically several levels of cache, each consisting of a faster type of memory and being closer to the CPU than the previous level. Modern CPUs typically have three levels of cache; a small Level 1 (L1) cache close to the CPU, a somewhat larger L2 cache and a significantly larger L3 cache. Normally, each level of cache is an order-of-magnitude faster than the previous, with L1 cache having similar performance as CPU registers. It is also common for each cache to be a subset of the next level, i.e., data resident in L1 cache is also present in the L2 and L3 caches.

In general, caching works as follows. Upon a memory access, it is checked whether the address is currently cached, potentially looking at all levels of cache present. If this is the case, a *cache hit* has occurred and the operation continues, either by reading from or writing to the cache. The opposite case represents a *cache miss*, where the CPU must copy the data into cache before continuing the operation. Caches operate on a *cache line* basis, which means that a certain amount of data is moved to and from cache regardless of the access size. In modern CPUs, the cache line size is typically 64 bytes, resulting in that 64 bytes are copied to cache regardless of the where the relevant data resides in the cache line. Data and instructions can also be *prefetched*. This involves caching the contents of memory that the CPU believes will be accessed soon in an attempt to minimize cache misses.

More specifically, there are a number of different ways to perform caching. Common modes includes *write-back* caching, *write-through* caching and *write combining*. *Write-back* caching is technique where cache lines are not written to memory before they are evicted from cache. To keep track of the operations performed on a line, a bit indicating whether the cache line is *dirty* or not, is used. If the dirty bit is not set, the cache line has not been modified, and a write to memory is not required during eviction. In contrast, *write-through* caching involves updating both the cache line and physical memory whenever the line is modified. This allows evicting cache lines unconditionally, but can result in an increased overhead. *Write combining* allows several write operations to be cached in a buffer before being issued as a single operation. Read operations are consequently performed without any form of caching. This technique is often employed when writing to I/O devices with memory that behaves as regular RAM, i.e., where accesses have no side effects.

In Linux, caching modes can be set on a per page basis through the *Page Attribute Table* (PAT) extension in modern CPUs. This is an extension to the regular page tables where certain bits in the PTEs indicate how the CPU should access the physical memory backing the page. The result is fine-grained control over what type of caching should be employed for each virtual

memory area. For pages backed by regular RAM, it is common to use write-back caching, while device resources such as registers, are not cached. An exception is when devices support write-combining, as described above. With regards to drivers remapping MMIO areas to user space, it is important that the caching scheme employed is correct for the type of resource. In addition, the scheme should be consistent for all mappings, regardless of whether device resources or physical memory allocated by the driver is being remapped. When remapping MMIO into kernel space in Linux, variations of the `ioremap` function allow setting different caching schemes, for instance `ioremap_wc` for write-combining. Correspondingly, the `vm_page_prot` member of the `vm_area_struct` can be tuned to achieve similar results when remapping physical memory or MMIO areas to user space with `remap_pfn_range`.

2.4 PCI Express

Peripheral Component Interconnect (PCI), and its successor PCI Express (PCIe), are bus standards that define how peripheral devices should be connected to a system. Historically, PCI has been the most commonly employed interface for extending a system with new devices, a role that PCIe now has inherited. Both are used to add a number of different devices, including Graphics Prosessing Units (GPU), network adapters and secondary storage devices.

PCIe retains several of the original concepts contained in PCI for backwards compatibility. Similar to PCI, PCIe devices are identified by their *bus*, *device* and *function* number, and they employ a *PCI configuration space* to let the OS know what kind of capabilities and resources they have. On a fundamental level, PCIe is quite different from PCI, however. While PCI uses a shared bus that all devices communicate through, PCIe uses a point-to-point topology where each device has its own path to the *root complex* of the system. The root complex connects the PCIe subsystem to the CPU and memory, similar to how a PCI host bridge work, and between a PCIe device and the root complex there may be one more switches which routes traffic to and from devices. Furthermore, paths between two points are called links, which may consist up to 32 bi-directional lanes. These lanes might be employed by a single device or be divided between one or more devices depending on the topology. In addition, PCIe uses a packet-based protocol for data transfer, instead of the transaction-scheme employed in PCI. The results of these changes are significantly higher bandwidth and lower latency, with maximum theoretical bandwidth for PCIe version 3.0 being 15.7 gigabytes per second (GB/s), compared to the maximum of 266 megabyte per second (MB/s) for regular PCI[5].

2.4.1 PCI configuration space

Host systems automatically perform configuration of PCI and PCIe devices by reading the device's *configuration space*. The configuration space comprises registers that contain various information about the device, for instance, its identity, capabilities and any resources it employs. While PCIe extended the PCI configuration space from 256 bytes to 4096 kilobytes, it is backwards compatible with PCI and identical in the parts that overlap. For this thesis, it is sufficient to present the regular PCI configuration space, which can be found in figure 2.5. The most notable fields in the configuration space is the *base address registers* (BARs). These

[5]The PCI eXtended (PCI-X) 2.0 standard did allow for a maximum bandwidth of 4.26 GB/s but was not widely adopted.

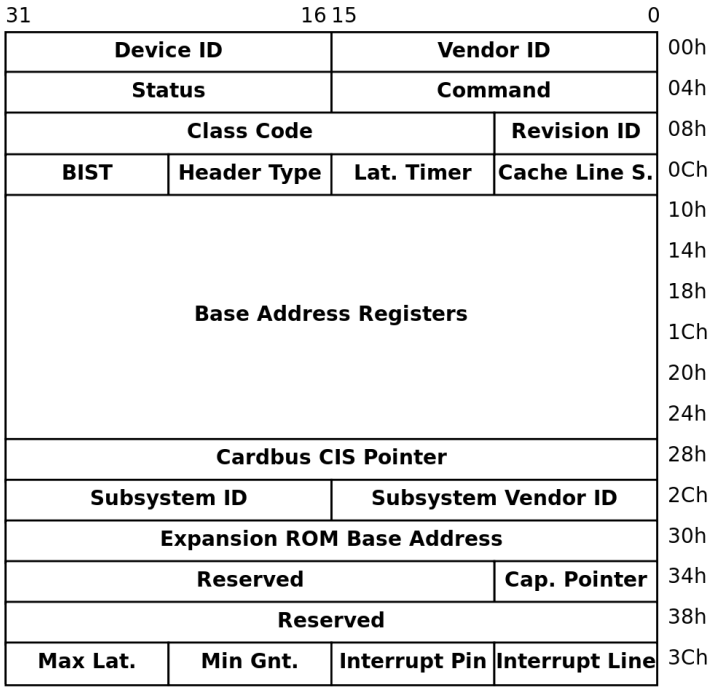

Figure 2.5: PCI configuration space [3]

contain information about what kind of resources the device has, and subsequently how much of the physical address space that must be set aside for them. There are two main resource types. The type of resource is indicated in the first bit of each BAR, which determines whether the resource should be mapped to the I/O space of the system or the regular memory space, in order to be accessed. In the latter case, the resource is a MMIO region, which a driver can remap to let the CPU access it as described in section 2.3.3. As shown in figure 2.5, there are six 32-bit BARs, alternatively three 64-bit BARs, all which may contain different resources, including registers and device memory. Typical for each type is that accessing registers have side effects, while device memory often behave as regular memory. This is indicated by a *prefetchable* bit in the BARs, which defines whether accesses to the resource must follow a strict pattern, or if operations can be reordered or merged.

2.4.2 Non-transparent bridge

In PCI so-called transparent bridges are employed to extend systems with additional PCI buses. The motivation for this was that one bus can only support up to five devices, which in many cases were insufficient. These PCI-to-PCI bridges were transparent, in that devices behind a bridge were visible and accessible as any other device residing on the main bus. While PCIe switches are modeled on such bridges, and actually appear as PCI-to-PCI bridges in software, PCIe also introduce a new concept called *non-transparent bridges* (NTBs). NTBs allows connecting PCIe root complexes together, facilitating point-to-point communication between systems. In contrast to transparent bridges, the devices and address space in one system is not visible in the other, instead the NTB can make memory windows available which one system can use to access devices or RAM on the other. These features makes it possible to employ PCIe as an interconnect between different computers, resulting in a very high bandwidth, low latency

networking mechanism. In addition to regular memory accesses, NTBs also allow remote devices to communicate using Remote Direct Memory Access (RDMA). This is possible due to PCIe' point-to-point topology and bypasses the need to involve the CPU even when devices are located on different systems.

As described in section 1.1, Dolphin Interconnect Solutions (Dolphin) takes advantage of the possibilities in NTBs to facilitate PCIe networking. In the next section, we will introduce the software and some of the hardware they provide to let remote applications share memory.

2.5 Software Infrastructure for Shared-Memory Cluster Interconnect (SISCI)

Dolphin provides add-on PCIe adapters with built-in NTBs and software to facilitate memory sharing between systems. Their adapters are connected by standard PCIe cables, either directly or through an external PCIe switch, with the latter allowing a number of systems to communicate. At the core of their shared memory solution is a notion of *memory segments*. These can be created on one system, referred to as a *node*, and made available to connected systems through the NTB located in their adapters. Subsequently, these segments can be accessed as regular memory by remote nodes to facilitate high-bandwidth communication with low latency.

On the software-level, Dolphin offers mainly two APIs to utilize the functionality in their hardware; Dolphin *SuperSockets* and the *Software Infrastructure for Shared-Memory Cluster Interconnect* (SISCI) shared memory API [16]. The former is a modified Berkeley Sockets API that lets existing socket-applications use PCIe based transport instead of e.g. ethernet, while the SISCI API lets applications take advantage of the shared memory capabilities of the PCIe adapters. In this thesis, we have focused on using the shared memory mechanisms in the SISCI API as a means to provide VMs with a high-performance IVMC solution. In the following, we will introduce the API, before shortly reviewing the hardware it can utilize.

2.5.1 API functionality

The SISCI API offers a number of mechanisms that allows applications to effectively communicate through memory shared over PCIe. In addition to creating memory segments that applications on other nodes can access, the API offers synchronization mechanisms through interrupts, error checking functionality, DMA support and more advanced features such as reflective memory (PCIe multicast). Before describing each feature more in detail, we should introduce one of the key data structures used in the API, namely the `struct sci_desc`. This structure represents a SISCI *descriptor*, which is used to manage API resources and communicate with the SISCI driver. Associated with each descriptor is a *device file* that must be opened to let the API pass subsequent requests to the driver. The latter is implemented as a character device driver and by default creates 256 device files. Initializing a decriptor is done through the `SCIOpen` API call, which in short finds an unused device file and opens it. Typically, a descriptor is an argument to each API call, with some exceptions where the descriptors are indirectly accessed through other API resources. These resources are always linked with one descriptor, and as a general rule descriptors can only contain one API resource of each type, for instance one local segment and one remote segment (see below). To close a

descriptor `SCIClose` is invoked, which deallocates any API resources and closes the device file associated with it.

The API can either be linked statically during compilation of applications that make use of it, or be utilized as a *shared library*. The latter involves dynamically linking it during runtime, which among other things makes it easier for applications to transition to new versions of the API.

Memory segments

As mentioned, the SISCI API allows applications to share memory through the creation of *memory segments*. The API distinguishes between *local segments* and *remote segments*, with the former being segments that a SISCI application creates and the latter representing segments that may be connected to. To create local segments, applications can invoke the `SCICreateSegment` API call:

```
void SCICreateSegment(sci_desc_t sd, sci_local_segment_t *segment,
    unsigned int segmentId, size_t size, sci_cb_local_segment_t callback,
    void *callbackArg, unsigned int flags, sci_error_t *error)
```

As seen, the function requires an open SISCI descriptor `sd` and `segment` pointer of type `sci_local_segment_t`, which is set during the call to later identify the segment. It creates a segment with a given `size`, `segmentId` and properties defined in `flags`, and returns status of the invokation in `error`. A callback can also be specified for the segment, which is invoked whenever an event related to the segment happens. Memory backing the segment stems from a pool of physical memory allocated on the local node by the SCI driver stack, and to make this memory available to applications on other nodes `SCIPrepareSegment` and `SCISetSegmentAvailable` must be invoked. The first function allows the segment to be accessed from Dolphins PCIe adapters, while the second makes the segment connectable from remote applications by utilizing NTB functionality. To connect to remote segments, applications invoke the `SCIConnectSegment` API call:

```
void SCIConnectSegment(sci_desc_t sd, sci_remote_segment_t *segment,
    unsigned int nodeId, unsigned int segmentId, unsigned int
    localAdapterNo, sci_cb_remote_segment_t callback, void *callbackArg,
    unsigned int timeout, unsigned int flags, sci_error_t *error)
```

The function requires that one specify the id of node that the segment resides on, the segment id and in case of multiple adapters, which adapter to perform the connection through. By setting a `timeout`, the application can define how long it wants to wait before giving up on connecting. Note that if the remote segment resides on the same node as the application connecting to it, a memory *loopback* mechanism will be employed when mapping and accessing the segment. In short, this involves remapping the physical memory backing the segment to the connecting application's address space. Consequently it is accessed as any other physical memory local to the system.

Accessing segment memory can be done by using PIO or DMA transfers. To access segments using PIO, the segment memory must first be mapped into the SISCI application. For remote segments this is done through the `SCIMapRemoteSegment` API call:

```
volatile void* SCIMapRemoteSegment(sci_remote_segment_t segment,
    sci_map_t *map, size_t offset, size_t size, void *addr, unsigned int
    flags, sci_error_t *error)
```

As indicated, it allows mapping `size` bytes of a remote segment from a given `offset`. The `map` parameter is used for managing the mapping, while by specifying an `addr`, the API will try to start mapping at the given virtual address. Local segments are correspondingly mapped through the `SCIMapLocalSegment` function, which has identical parameters except that the segment is of the `sci_local_segment_t` type. After mapping either a local or remote segment, they can be accessed through the returned virtual address using standard PIO operations such as load and store, or `memcpy` functions. The API also offers its own, slightly optimized function called `SCIMemCpy`.

The SISCI API allows DMA to be used for data transfers between local segments, from local to remote segments (RDMA) and to devices such as NVIDIA GPUDirect RDMA capable GPUs. This is made possible by various API calls, which we will briefly introduce here. To initialize a DMA queue, which is used to process DMA requests, `SCICreateDMAQueue` is invoked. Subsequently, one can start DMA transfers by calling `SCIStartDMATransfer`:

```
void SCIStartDmaTransfer(sci_dma_queue_t dq, sci_local_segment_t
    localSegment, sci_remote_segment_t remoteSegment, size_t localOffset,
    size_t size, size_t remoteOffset, sci_cb_dma_t callback, void
    *callbackArg, unsigned int flags, sci_error_t *error)
```

Most parameters will be familiar to readers, with an optional `callback` function which is triggered when the transfer is complete. To allow DMA transfers to third-party devices such as GPUs, `SCIAttachPhysicalMemory` must first be invoked to associate device memory with a SISCI segment, after which a DMA transfer can be started by specifying the resulting segment as either the source or the destination.

Interrupts

In order to let applications synchronize their actions during runtime the SISCI API offers *interrupt* mechanisms. These can be used to signal applications about relevant events, for instance the completion of a data transfer to an application's local segment. Similar to the segment mechanism, interrupts are created by one application and can be connected to by others. Subsequently, they can wait for an interrupt and trigger an interrupt, respectively. To create an interrupt the `SCICreateInterrupt` function is invoked:

```
void SCICreateInterrupt(sci_desc_t sd, sci_local_interrupt_t *interrupt,
    unsigned int localAdapterNo, unsigned int *interruptNo,
    sci_cb_interrupt_t callback, void *callbackArg, unsigned int flags,
    sci_error_t *error)
```

Here, `interruptNo` is interpreted as a hint by the API in regards to which number the interrupt should be assigned on the node. If not available, the API will assign an arbitrary interrupt number. Furthermore, `callback` points to an optional callback function, which in

this case is invoked upon receiving an interrupt. Other applications, whether residing on the same node or a remote one, can connect to the interrupt through `SCIConnectInterrupt`:

```
void SCIConnectInterrupt(sci_desc_t sd, sci_remote_interrupt_t
    *interrupt, unsigned int nodeId, unsigned int localAdapterNo, unsigned
    int interruptNo, unsigned int timeout, unsigned int flags, sci_error_t
    *error)
```

The caller must specify a `nodeID` and `interruptNo` in order to identify the correct node and interrupt, and a `timeout` can be set to give up on trying to connect after a given time interval. To trigger interrupts, applications simply call `SCITriggerInterrupt` with the returned `sci_remote_interrupt_t` data structure as the parameter. This causes one of two actions on the node that created the interrupt. If a callback function was specified when the interrupt was created, it is invoked. In the opposite case, any applications that are waiting on that specific interrupt are woken up. Waiting for an interrupt is done by calling `SCIWaitForInterrupt`:

```
SCIWaitForInterrupt(sci_local_interrupt_t interrupt, unsigned int
    timeout, unsigned int flags, sci_error_t *error)
```

The parameter of interest here is `timeout`, which specifies how long the application wants to wait for an interrupt to happen. This can either be a given time interval in ms, or the predefined macro `SCI_INFINITE_TIMEOUT`, which causes the application to block until an interrupt is received.

In addition to regular interrupts, the API also supports *data interrupts*. In short, this is a mechanism that allows sending a small amount of data, up to 100 byte, with each interrupt. Among its use cases are the implementation of transfer protocol, and as the mechanism is otherwise identical to the interrupts described above, we do not believe a more thorough introduction is required.

Error checking

While errors in data transfers over PCIe is scarce, especially when using short PCIe cables, the SISCI API offers functionality to perform error checking. This is done through a *sequence* mechanism, which makes it possible to check for errors when performing a sequence of memory accesses. A sequence is created through the `SCICreateMapSequence` API call, and started by invoking `SCIStartSequence`. After writing a number of bytes to a remote segment, for instance, `SCICheckSequence` can be called to flush any write-combining buffers and check if an error has occurred. Errors can include a *retriable* fault, where one can try to perform the write or read operation again, or a *fatal* fault. In the latter case, `SCIStartSequence` must be invoked until it returns no errors, to further act on the segment.

2.5.2 Hardware

As mentioned the SISCI API utilizes capabilities in Dolphins PCIe adapters to share memory between applications. Currently, the main line-up of adapters include one PCIe Gen2 adapter and two Gen3 adapters. The Gen2 adapter is called IXH610 and can utilize 8

PCIe lanes, allowing a remote bandwidth up to 3000 MB/s. Gen3 adapters include the IXH810, which supports x8 mode, and the 16 lane IXH830 adapter. They offer a maximum bandwidth/throughput of around 6000 MB/s and 10000 MB/s, respectively.

Note that the SISCI driver is not responsible for communicating with the hardware directly. Instead, it sends request to other lower level drivers in the SCI driver stack, which mediate these requests on behalf of it. We will not go into the details of the driver stack now, but introduce different parts of it where appropriate.

2.6 Inter-VM communication

Inter-VM communication (IVMC) is in short any communication that takes place between VMs. Traditionally, VMs has been limited to using standard networking mechanisms such as TCP/IP to exchange data, even when the VMs are located on the same physical computer. Lately, and especially with the introduction of hardware support for virtualization, it has become more viable to employ VMs in scenarios where high performance is required. Often referred to as High Performance Computing (HPC), such use cases may require high bandwidth and low latency This has led to various research efforts where the goal is to increase IVMC performance, in an attempt to avoid it becoming a bottleneck in HPC clusters utilizing otherwise high performance virtualization environments.

In this section we will introduce some of the aforementioned research. Many efforts are based on sharing memory between VMs, and while some focuses strictly on performance, most look into extending or replacing certain parts of mechanisms such as TCP/IP to retain binary compatibility with existing applications.

Developed by Zhang et. al., *XenSocket* [6] is an IVMC solution for co-located Xen domains. It was designed and implemented as an alternative to the regular TCP/IP communication offered in Xen, which performance was found to be lacking for a high-performance use case the authors were involved in. The solution is based on sharing memory between Xen domains and providing a new socket interface based on UNIX sockets, to the domains. Through the socket interface a *receiver* application can create a socket, which in turn allocates a circular buffer of shared memory. Correspondingly, a *sender* application can connect to the socket and write to it to transfer data into the shared memory. To exchange information about the shared memory, Xen's grant table mechanism is used during the connect phase.

As implied above, the solution only allows unidirectional data transfers, where the role of each application is set based on the direction of the first transfer. In addition, the circular buffer is limited to 128 KB, and only blocking *read* and *write* operations are supported. Furthermore, a new socket interface results in that binary-compatibility with existing socket-based applications is not achieved. The performance improvement, however, is significant, with up to 72 times the throughput of TCP/IP for certain data transfer sizes and a maximum of 9295 megabit per second (Mb/s).

XWAY [17] is another IVMC solution for Xen which is somewhat similar to the aforementioned XenSocket mechanism. In contrast to the latter, however, XWAY provides full binary compatibility with the standard socket interface. In short, this is achieved through so-

called *XWAY sockets*, which contain a XWAY channel and a standard TCP channel. The XWAY channel uses shared memory as its transport mechanism when it is detected that the receiver application resides on a co-located domain, otherwise the standard TCP channel is employed. In the former case, the TCP/IP stack is bypassed completely to increase performance. The result is that the peak throughput between co-located domains are nearly three times higher than for regular TCP/IP sockets, with a maximum of around 4000 Mb/s. This is still lower than the peak memory bandwidth available in the domains however, which is indicated to be around 8000 Mb/s.

In their paper from 2008 named *Efficient shared memory message passing for inter-VM communications* [18], Diakhate et. al. present a shared memory based solution for Qemu/KVM, used for Message Passing Interface (MPI) communication between VMs. The mechanism is based on using shared memory for smaller message size, and a pseudo DMA mechanism for larger sizes. The shared memory is exposed to the guests through the BARs of a virtual PCI device, while Virtio is used to forward DMA requests which are handled by the Qemu backend. A limitation with the mechanism is that Qemu instances are forked to allow guests to share memory. This reduces its usability, as the number of VMs communicating must be defined statically at startup. The mechanism is also only applicable for communication between co-located VMs. It is found that the solution offers better performance than standard MPI mechanisms employed between VMs, with around 5000 MB/s as the peak throughput.

Nahanni is another IVMC solution for Qemu/KVM, implemented as a part of Cameron MacDonnells PHD thesis from 2011 [19]. The mechanism, which also goes under the name of *Inter-VM Shared Memory (ivshmem)*, allows co-located VMs to communicate through POSIX shared memory allocated on the host. As [18], the shared memory is made available through the BARs of a Qemu virtual PCI device. More specifically, the Qemu backend opens a new or an existing POSIX shared memory object and maps the memory into a VM through one of the device BARs. In the guest, the memory appears as regular device memory and can be mapped and subsequently accessed by guest applications. In order to facilitate sharing of the memory between different VMs, the file descriptor associated with the object is exchanged through a socket-based server which runs on the host. In addition, the mechanism includes synchronization primitives for the applications that communicate. Contrary to [18], the use of POSIX shared memory allows Qemu instances to be initiated independently. However, any VMs that should share memory must define the server to use and the size of the shared memory object, upon startup.

The performance of the mechanism was found to be significantly higher than existing IVMC solutions at the time, especially for regular data transfer, where the shared memory allows for high bandwidth and low latency. It is noted, however, that the mechanism acts as a more basic solution for moving data between VMs, compared to mechanisms stressing binary compatibility.

In [21], Sreenivasamurthy et. al. addresses some issues with the *Nahanni* mechanism. More specifically, they target some security concerns regarding the static allocation of shared memory, and management related issues. Their modifications leave the performance of data transfers unchanged, and as before, communication is limited to co-located guests and their host.

Developed for Qemu/KVM by Zhang et. al., *MemPipe* [20] employs shared memory to optimize TCP and UDP communication between co-located VMs. Similar to Nahanni, this is achieved by exposing POSIX shared memory to guests through a virtual device. A significant difference, however, is that the mechanism allows *dynamic* management of the memory that is shared[6]. As a result, it is not required to statically define the memory that should be shared when setting up the VMs. The mechanism also supports falling back to regular TCP/IP when remote VMs communicate. Evaluation is done by running TCP and UDP workloads on top of the shared memory. It is found that MemPipe has better performance both when compared to regular machine to machine scenarios, and scenarios where two co-located VMs use regular TCP and UDP for their communication.

In [22], Pinto et. al. presents a zero-copy shared memory framework for bi-directional host to guest communication. In contrast to [19], memory sharing is achieved by having host applications remap memory allocated in the guest. This avoids the need to define shared memory properties during initialization of a guest. Performance is found to be close to native, but as indicated, no support for remote communication is present.

2.7 Summary

In this chapter, we have introduced various topics that are relevant for our project. We have described virtualization technology, some parts of the Linux OS and how the SISCI API facilitates memory sharing between both local and remote applications. In addition, we have discussed some previous research that has been done on the IVMC area. As we have seen, several shared memory IVMC mechanisms targeted at co-located VMs exist. However, in these solutions, support for communication between VMs that are located on different physical computers is nonexistent or limited to standard mechanisms such as TCP/IP. In the next chapter, we will propose our design for a new IVMC solution that utilizes the SISCI API on top of a PCIe interconnect to provide VMs with both local and remote high-performance communication capabilities.

[6] As will be seen in the next chapters, our design and implementation share this basic idea with the MemPipe mechanism. The latter was, however, discovered at the final stages of writing this thesis and has not been a source of inspiration

Chapter 3

Design

In this chapter, we describe our design for a shared memory IVMC mechanism based on the SISCI API. Our main goal is to allow high performance communication to take place between VMs, regardless of whether they are co-located or located on different physical computers. To facilitate this, we intend to use local shared memory in the former scenario, and memory exposed through a PCIe interconnect in the latter.

Our approach to both the design and implementation of this mechanism has been an iterative process. We have explored different ideas, tested them, and scrapped the ones not deemed appropriate or sufficient for our goals. At the same time, some key points were apparent when we started the process, aspects that have been persistent throughout both the design and implementation phase.

In the next section, major design goals and considerations will be introduced. Later, we present our choice of hypervisor for the project, some of the approaches we explored and the architecture of our initial design.

3.1 Design goals

Since we aim to implement a high-performance IVMC solution, a major goal is related to performance. Combined with the PCIe interconnect from Dolphin, the SISCI API offers very high bandwidth and low latency communication mechanisms. Optimally, our mechanism should capture most of the underlying performance potential to be viable, and as a consequence, we set out with clear goal of achieving memory bandwidth and latency comparable to those in a native environment. A proof of concept of a working virtualized solution with a significant overhead would still be relevant, as it could act as a base on which future optimizations could be applied. However, its immediate relevance would undoubtedly be smaller, considering there are numerous standard communication mechanisms with acceptable performance that can be used between VMs. Since Dolphin's solutions target systems requiring high performance, our solution should have high performance as well. To achieve this, the hypervisor or host involvement should be kept to a minimum. For instance, when a guest requires data transfer to another guest using SISCI's shared memory mechanism, the actual memory operations should optimally be performed in the guest, without involvement from the hypervisor or other components in the host. In short, most of the performance-sensitive functionality must be isolated to the guest, and only when necessary is the host included.

Another consideration related to extending the SISCI API to a virtualized environment,

is binary compatibility. Ideally, existing SISCI applications should work in VMs running virtualized SISCI, without any modifications. If modifications are necessary, they should be minor and kept to a minimum. We therefore aim to implement a solution that requires minimal changes to the SISCI API, if any, and that leaves interfaces and the core functionality of the API intact. This makes the solution more available to existing users, it allows us to reuse code, and since the API includes a number of benchmarks, it may help us evaluate our implementation.

A third aspect, that stems from the virtualization process, is security. The new communication capabilities given to the VM must not introduce new ways of compromising the host. Boundaries already in place between the host and VMs should be upheld, and isolation between co-located VMs should be kept intact as far as possible.

We also define a set of functional requirements that the IVMC mechanism should adhere to. A key requirement is support of communication between VMs located on different physical computers, or in SISCI terminology, *nodes*, as well as between co-located VMs. Both communication schemes should be available during runtime, i.e., communication between both remote VMs and local VMs should be possible without defining the type(s) of communication required when setting up the VM. More specifically, the communication mechanism should offer high-speed data transfer capabilities through shared memory and a means to perform synchronization. We aim to provide this by implementing a subset of the SISCI API in a virtualized environment. This includes core functionality that allow access to local and remote segments, interrupt mechanisms and possibly DMA support.

3.2 Choice of hypervisor

Choosing a hypervisor for this project is an important decision that influences both the design and implementation of our mechanism. As described in section 2.2, the two hypervisors we have considered viable are Xen and KVM with Qemu, as they are both open source and have active developer communities. They do, however, take quite different approaches to virtualize guests. In the following, we will compare the two hypervisor alternatives to determine which is most suited to achieving our goals. A number of considerations are relevant in this comparison. Since our main goal is to provide a high bandwidth, low latency IVMC solution that introduces minimal overhead, performance is an important factor. Other aspects include what environment the hypervisors run in and the features they offer.

3.2.1 Performance comparison

Although a reasonable amount of performance comparisons have been made, some have been performed in the early days of the respective hypervisors and are outdated. Some are also performed without employing paravirtualized I/O for KVM, which often is required to achieve maximum I/O performance. While these benchmarks may not necessarily represent a totally accurate picture of how the hypervisors compare to each other performance wise, they can still be of some value in assessing certain performance aspects.

In [23], Deshane et al. compare the overall performance, performance isolation and the scalability of Xen and KVM. They find that Xen outperforms KVM in all aspects except disk I/O, while performance isolation, to what extent the performance of one VM affects the performance of other VMs, is similar. The scalability of Xen is found to be excellent, while

KVM struggles when the four or more VMs are run. However, being from 2008, the comparison is dated, and KVM does not use paravirtualized I/O.

A newer set of benchmarks are the ones performed by Steve G. Langer and Todd French in 2011 [24]. Here, the memory, disk and network I/O, as well as the CPU performance of both hypervisors are compared. Xen and KVM are found to have similar performance, with the exception of memory write operations, where Xen is twice as fast as KVM, and outbound network traffic, where Xen lags behind KVM by the same margin.

Graziano evaluates several performance aspects of both hypervisors in his graduate thesis from 2011 [25]. In single user scenarios, he finds that Xen generally outperforms KVM with regards to memory operations and disk I/O. In contrast, KVM is slightly faster when running CPU intensive tasks and has a clear advantage when it comes to performance isolation.

In [26] from 2014, the performance of using GPU-passthrough with nVidia Fermi and Kepler GPUs are compared using a range of benchmarks. They conclude that KVM consistently delivers near native performance, while Xen's performance is consistently average.

While it is difficult to make an absolute conclusion about Xen and KVM's performance based on these benchmarks, it initially seems that Xen may have a slight advantage. On the other hand, indications about KVM having better performance isolation must be taken into account. Because of this, and the fact some benchmarks are outdated, our conclusion is that the performance difference between the hypervisors is small.

3.2.2 Hypervisor environment

Another aspect is the environment of the hypervisors. KVM is a kernel module in Linux that adds virtualization features to an otherwise standard Linux environment, while using *Qemu* to emulate hardware. In contrast, Xen comprises the hypervisor and dom0, and divides typical system tasks between the two components. While the implications of this structure is not fully understood at the time of choosing a hypervisor, it may add some complexity to the integration process. The SISCI driver and the SCI driver stack are implemented for a regular Linux environment and will possibly require modifications to work with the structure of Xen. KVM on the other hand, is, as mentioned, said to work with any hardware supported by Linux. It may also be advantageous that KVM uses Qemu to emulate hardware. Qemu supports emulating a range of different devices and device mechanisms and has APIs that make it relatively easy to add new virtual devices that KVM may utilize. In addition, KVM's support of the *Virtio* framework for paravirtualized I/O may be beneficial in the development process, as it is widely adopted by the Linux community.

3.2.3 Features comparison

The features supported by either hypervisor is also an important consideration. Many important virtualization features are present in both Xen and KVM. This includes live migration of VMs from one host machine to another without noticeable delay, and memory overcommitment. The latter involves being able to allocate more virtual memory to the set of VMs currently running than are physically available, which increases the number of potential VMs on a host. Device passthrough in general is also supported in both hypervisors, but while Xen allows passthrough of GPUs, this is not officially supported in KVM. It is possible, however, as proved in [26].

Xen additionally offers a mechanism for different VMs to share memory between one another, by the use of *grant tables*. Through grant tables, a Xen domain may allow other domains to access its memory, which then can be used for communication. A similar feature does not inherently exist in KVM, but at the time of choosing a hypervisor, it is not clear whether this mechanism is actually suited to our project.

In conclusion, the performance and feature set of either hypervisor is similar. KVM does, however, offer an environment that is proven to be compatible with the SISCI driver and the SCI driver stack. It also and employs Qemu to emulate hardware, which provides a number of ways to implement I/O for the guests. For these reasons, KVM is chosen as the hypervisor for our project.

3.3 Exploring different designs

After choosing a hypervisor, we initially looked at three different approaches to allow the SISCI API to operate in a virtualized environment; performing passthrough of the Dolphin IXH610 PCIe adapter to VMs, using a Virtio-based virtual device utilizing virtqueues as transport, and creating a custom Qemu virtual device with SISCI segment memory acting as device memory.

3.3.1 Passthrough

Passthrough of the IX610 adapter would represent a clean way of exposing its capabilities to a VM. Device passthrough is supported by both hypervisors and allows standard device drivers to be utilized in the VM, on top of which we could run an unmodified version of the SISCI API. As described in section 2.1.3, however, an important limitation with regular passthrough is that a PCI/PCIe device only can be assigned to one VM at a time. This means that one IX610 adapter would be needed per VM on a single host, which is neither a flexible nor cost-effective approach, and represents little in terms of innovation in the IVMC field.

Instead, one could imagine a solution based on SR-IOV, which lets a single PCIe device be shared among multiple VMs. SR-IOV is supported by most hypervisors, including KVM, and ultimately, an SR-IOV solution could also allow running an unmodified SISCI API in VMs. The IXH610 adapters are capable of supporting SR-IOV, but at the time of writing this thesis it is not implemented. Implementing it would require in-depth knowledge of the adapter and its drivers, a task that undoubtedly would be large and complex. The implementation would also be very device-specific, limiting the value of the work presented in this thesis from a research standpoint. In addition, for communication between co-located VMs, SR-IOV would probably not be optimal, as stated in section 1.1.

For these reasons, we abandoned the thought of using regular passthrough or SR-IOV to implement our IVMC mechanism

3.3.2 Virtio

A different approach is using paravirtualization to bring SISCI capabilities to guests. Paravirtualization is a commonly used technique for I/O virtualization because of the performance advantage it often has over a standard I/O emulation. One option would be to

employ the *Virtio* framework (see section 2.1.3) for this purpose. Virtio has become increasingly popular and is supported by several hypervisors and OSes, leading to excellent portability. Its performance is also often better than regular paravirtualization approaches, meaning that it could represent a viable starting point for our solution. Through the framework, we could forward API calls to the host, which then performs the request on behalf of the guest. However, a caveat with Virtio is that the host by default also is involved when transferring data over the VM border. Virtio uses ringbuffers, where requests are placed for the host to handle, and where results are posted to the guest. Any data that is to be transferred from the guest must initially be written to a ringbuffer and, in our case, subsequently be copied from these buffers into a SISCI segment to perform the actual transfer. This means that our goal of minimizing host involvement is not accomplished.

Despite this, we investigated the performance of Virtio by running some simple tests with the `virtio-serial` device. The performance metric we were most interested in was what kind of bandwidth Virtio could deliver, and to review this metric, we devised a simple benchmark where data were moved from a guest to its host. Running the benchmark yielded a bandwidth of around 700 megabytes per second (MB/s). After a bit of research, we came to the conlusion that Virtio could provide a somewhat higher bandwidth, around 1200 MB/s, but this is nowhere near the maximum memory bandwidth available in the host. It is also quite far off the theoretical maximum bandwidth of around 3000 MB/s that the IXH610 adapters offer when accessing remote memory. We therefore looked for an alternative approach.

3.3.3 Custom virtual device

The next approach we explored was a paravirtualization-based solution where a custom virtual device is employed to virtualize the capabilities provided by the SISCI API. As described in section 2.5, the API and the underlying driver stack is based on using segments to allow local and remote applications to share memory. After our Virtio tests, it became clear that if we were to keep the high-bandwidth, low latency characteristics originally offered in the API, we had to find a way to remap segment memory into a guest. Our initial thought on how this could be done,

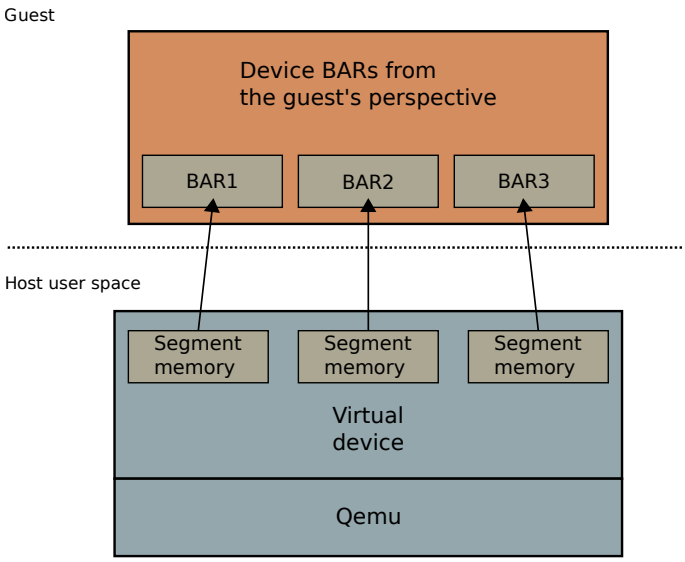

Figure 3.1: Device BARs backed by SISCI segment memory

was to expose the memory as part of a virtual device presented to the guest. As shown in figure 3.1, with a PCI device the basic idea is to have segment memory backing the device's BARs. Ultimately, this could allow SISCI applications in the guest to operate directly on segment memory, similar to how native applications access it, without the involvement of the host. In effect, it would represent a *zero-copy* mechanism, which in a virtualization context refers to that no intermediate copy operation is needed when transferring data. As a result, bandwidths close to that of the native case could be obtainable, with hypervisor memory performance being the main limiting factor. In this approach, management related operations such as creating segments and mapping them, and control operations such as triggering interrupts, can be forwarded to the host using standard paravirtualization techniques. As transport, we could either employ a Virtio-based communication channel or a mechanism tied directly to the custom virtual device.

Overall, we regarded this approach as very promising, with the main drawback being decreased portability, at least when compared to a full Virtio solution. Its performance potential led us to investigate it further, however, and our first goal was to successfully remap segment memory into a guest.

When investigating how host memory can be mapped into guests with KVM and Qemu, we happened upon the aforementioned IVMC mechanism *Nahanni/ivshmem* (see section 2.6), that is included in Qemu. Nahanni maps shared host memory into guests in the same way that we want to map memory that stems from a SISCI segment into a guest. While there are differences between POSIX shared memory and the memory associated with SISCI segments, the overall approach of sharing memory is similar. We therefore set out to experiment with mapping SISCI memory into a guest using the *ivshmem* virtual PCI device as a starting point[1].

Initially, we simply tried to use memory stemming from a local SISCI segment, instead of POSIX shared memory, with the existing mechanisms in the ivshmem device. We then attempted to remap the device memory in the guest and access it. This did not prove a success. The guest crashed whenever we tried to map the memory into a user space application, and sometimes this would also make the host crash in form of a kernel panic, which made debugging difficult. In spite of that, we started debugging extensively to extract whatever information we could gather about the causes of these crashes. This included debugging a number of mechanisms in both Qemu and KVM. After spending a significant time doing this, we concluded that the KVM simply did not support remapping the type of memory backing local SISCI segments. Local segment memory in SISCI stems from a memory pool of physically contiguous pages allocated with the kernel function `__get_free_pages`. For later Linux kernel versions, these are remapped to user space with the `remap_pfn_range` function[2]. Mappings set up by `remap_pfn_range` are special, in that there are no `struct page` associated with them. They are what is referred to as *raw pfn mappings*, meaning that they are mainly set up by modifing the page tables of the user space process. This is indicated in `remap_pfn_range` by it setting the `VM_PFNMAP` and `VM_IO` flags in the corresponding `vma` structs of the mappings. The problem with this, was that KVM did not support handling such physically remapped pages; upon accessing the memory from the guest, an EPT violation and a page fault that KVM could not handle, would occur, and the guest would crash. More specifically, KVM did not support resolving mappings that had no `struct page` associated

[1]The specific techniques that we employed in an attempt to achieve this will not presented here, rather it will be covered sufficiently in section 4.1.1. Note that while our implementation uses the same basic techniques, it also differs significantly from the early tests we performed at this stage.

[2]This function is also used to remap remote segment memory to user space

with them, as `get_user_pages` would fail on the addresses. To confirm that KVM was the problem, we ran our VM using Qemu without KVM acceleration, i.e., in standard emulation mode. Accesses to segment memory from the guest were then successful, albeit very slow.

At this point. we considered two options for our further course of action. One was attempting to modify KVM to support resolving page faults stemming from accessing such memory. However, our knowledge of the intricate memory mechanisms in Linux and KVM was lacking at the time, which made us reluctant to try this. A second option was to continue working on the approach using Qemu only. While the bandwidth achieved without KVM was very low, it would allow us to design and implement a proof-of-concept that could become viable in the future. We decided to do the latter and began looking at how we otherwise could bring SISCI capabilities to a guest using a custom virtual device. Early in this process, however, a small patch was introduced to KVM which made us optimistic. It specifically targeted resolving page faults stemming from `VM_IO` and `VM_PFNMAP` mappings manually. Our initial tests with local SISCI segments was not successful, but mapping remote segment memory into a guest now worked, with preliminary tests showing that remote memory bandwidth in the guest was comparable to native.

After this, we continued debugging why local segments were not being mapped correctly, and in the end found that it was not actually related to our mapping procedure. Rather, the way that the SCI driver stack allocates the physical memory pool that local segment memory stems from, was incompatible with KVM. As mentioned, this memory pool is allocated by calling `__get_free_pages`, which if many page frames are requested, only increases the reference count of the first page frame. When KVM handles a page fault from physically remapped memory, it temporarily increases the reference count, which upon decrease, leads to that the reference count of subsequent pages reach zero. As a consequence, the pages backing local segments are put on Linux' list of free pages, allowing them to be resued as it sees fit. Upon accessing these pages from a guest, a variety of errors would then occur, as the pages no longer belonged to a SISCI local segment. To resolve this, we changed how the SCI driver stack allocates pages for the memory pool. Instead of using `__get_free_pages`, we modified it to use `alloc_pages_exact`, which increases the reference count not only for the first page, but for all pages allocated. As a result, we could now access local segment memory from the guest without errors, also with a near native bandwidth.

Having managed to map both local and remote segment memory into a guest through a virtual device, and confirmed the performance this approach offers, we chose it as the basis for our further design and implementation. In the next section, we will propose an architecture for our design.

3.4 Architecture

As indicated in the previous section, our design is based on employing a custom Qemu virtual device to present VMs with SISCI capabilities. Here we discuss and propose an architecture for this approach, which served as a basis for our implementation.

As shown in figure 3.2, our architecture consists of the following main components; a Qemu virtual PCI device, a communication channel between the guest and the host and a guest kernel driver for the virtual device. In addition, Qemu/KVM must be modified to work with the SISCI API, and the API itself modified to work in a virtualized environment. Note that the figure is

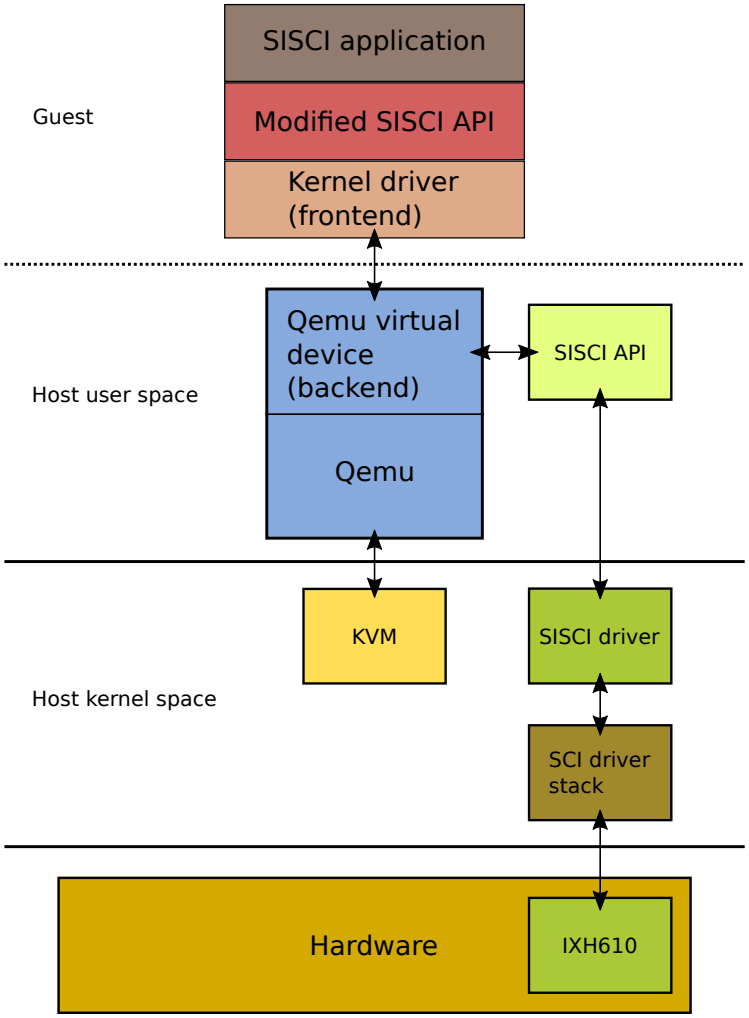

Figure 3.2: Architecture of our design proposal

not entirely precise with regard to the role of KVM. In reality, it sits between the guest and host user space, forwarding I/O requests to Qemu. Its position in the figure has been changed to better illustrate that the guest driver communicates with Qemu.

In the following, we will shortly introduce each component and describe their main responsibilities. These small sections reflect our thought process in the design phase, which, as previously stated, overlapped somewhat with the implementation phase. As a result, our architecture is not exhaustive, but rather an overview of each component. After presenting the final architecture, we will discuss some modifications to the approach that we ended up disregarding.

3.4.1 Qemu virtual device

The Qemu virtual device interfaces against the standard SISCI API on the host, to which it forwards translated API requests received from the guest driver. Qemu itself is modified so that

it can make use of the SISCI API, by linking SISCI as a shared library when configuring and compiling Qemu. To make the request translation operation as simple as possible, the forwarded requests should have a format that is similar to their SISCI API counterparts. Since most of the SISCI API calls revolve around *descriptors*, the virtual device includes data structures and mechanisms to keep track of the resources associated with each descriptor and to identify which descriptor a given guest request is related to. More specifically, the state of a descriptor in the host reflects the state of the corresponding descriptor in the guest, resulting in a 1-to-1 relationship between host and guest descriptors (see section 3.4.3).

The virtual device is also responsible for making segment memory available in the guest, by mapping it as part of the device's memory. To realize this, exclusive device BARs backed by segment memory are created, which is possible through existing functionality found in the PCI and memory APIs of Qemu. Optimally, it should be possible to dynamically add and remove segments from the device on demand during runtime. The opposite approach, to require that all segments which are to be used are defined upon guest startup, will severely limit the flexibility of our mechanism. This would require exhaustive knowledge of all possible SISCI use cases in a particular guest during initialization.

Furthermore, the device is tasked with injecting interrupts into the guest. This is important because VM exits effectively halt the guest, meaning a mechanism where the host waits for events on behalf of the guest, is not viable. The solution is asynchronous notifications from the host to a guest, in form of interrupts. In practice, this results in a mechanism where any SISCI interrupts received in the host is propagated to the guest through the device, independently of guest execution.

3.4.2 Communication channel

The communication channel is responsible for transferring API requests from the guest driver to the Qemu backend residing in the host. It must support bi-directional data transfer in order to allow the host to communicate request results and other relevant data, back to the guest. In addition, multiple concurrent requests should be allowed. By the latter, we mean that it is possible for the guest driver to queue up several requests from the SISCI API at a time, without any request information being overwritten or race conditions occurring[3]. Furthermore, we aim to implement a simple and generic communication protocol, to minimize the cost of request handling and to increase the portability of our solution.

3.4.3 Guest driver

The main task of the guest driver is to interface with the modified SISCI API in the guest, forward requests from the API to the host, and to map memory made available through device BARs into SISCI applications. To minimize modifications to the guest API, the driver must employ a similar interface to the standard SISCI driver. We intend to implement the driver as a character device driver to realize this. The driver should handle requests on a per SISCI descriptor basis, and be able to keep track of the resources associated with each descriptor. This is necessary to correctly handle subsequent requests on a descriptor or its resources, and to keep the 1-to-1 relationship between descriptors in the host and in the guest. Furthermore, our guest

[3]As a VM exit is normally performed when Qemu/KVM needs to handle an I/O requests, it is not actually possible to handle requests concurrently in the Qemu backend.

driver must handle the existing SISCI data structures as parameters to requests from the API. These requests are translated to the format used in our guest to host communication channel and forwarded to the Qemu backend. Note that we aim to forward requests only when strictly necessary, as VM exits are expensive.

As mentioned, the driver is also responsible for mapping segment memory made available through the device BARs, into the guest. More specifically, it must offer a mechanism to remap this memory into SISCI applications upon request, allowing them to access it directly. This can be realized through the `mmap` operation in its interface, which also provides basic protection from unauthorized memory accesses.

In addition, the driver must implement mechanisms for handling interrupts from the Qemu backend. This mechanism notifies any SISCI applications currently awaiting an interrupt, preferably in a manner that is coherent with the SISCI API's interrupt mechanisms.

3.4.4 Modified SISIC API

In short, the SISCI API employed in the guest is modified to interface against our new guest driver instead of the standard SISCI driver. We aim to keep modifications to a minimum and adhere to the existing structure and style of the API. In any case, the API interfaces must be kept intact, to allow compatibility with existing SISCI application.

3.4.5 Discussion

An alternative approach to architecture described above, is to have a more direct path between the guest driver and the SISCI driver or the SCI driver stack. More specifically, the SISCI API in the host would not be involved when receiving API calls from a guest, which could possibly reduce the overhead of request forwarding. We envision this may be achieved in two different ways.

One option would be to pass requests from the API in guests directly to the Qemu backend without performing any translation. Structures and other parameters from the API would be forwarded without modifications, and in turn the Qemu backend could pass these to the SISCI driver without involving the API. This might have resulted in a somewhat lower overhead when forwarding requests but would undoubtedly add complexity to our solution. In many cases, the SISCI API performs a number of tasks to complete an application request. This can include several `ioctls` to the SISCI driver, sanity and error checks and in general operations that are dependant on the current environment. While sanity and error checking to a degree could be left out, as the SCI drivers also perform such tasks, other operations would have to be carried out by the Qemu backend. In effect, this would represent moving functionality from the SISCI API to Qemu, which by itself is of little purpose. In addition, it is unlikely that the parameters set in guests could directly be employed in the Qemu backend, seeing how a guest and its host represent two different environments.

A second approach is to have KVM pass SISCI related I/O requests directly to the SCI drivers instead of forwarding them to Qemu. This could represent an even faster method for handling requests than the one described above, but it is also equally more complex. Since KVM by itself provides no mechanisms for device emulation, such functionality would have to be added, and it is not clear to us how or if this could be done. Possibly it would involve

a solution without a virtual device, where memory is made available through other more traditional memory mechanisms.

Common for both these approaches is that they are more complex than our chosen architecture. While they potentially may offer better performance, the benefits are doubtful in the first case, and the second might prove difficult to implement. For these reasons, we have regarded the architecture underlined in the introductory sections as the most viable for our project, leaving the alternative approaches as possible optimizations for the future.

3.5 Summary

In this chapter, we have described our design for a new shared memory IVMC mechanism based on the SISCI API. We have introduced some general design goals, chosen a hypervisor for the project, and explained how we reviewed different approaches for realizing the mechanism. An approach based on a custom virtual PCI device has been determined to be the optimal solution based on a goal of high performance. By exposing SISCI segment memory through the device's BARs, guests can access both local and remote shared memory directly, which means that host involvement is not necessary during data transfer. Finally, we have outlined an architecture for this approach, including the required components on the host and in guests. In the next chapter, we will go into detail of how we implemented our mechanism based on this design.

Chapter 4

Implementation

Our final implementation follows the design outlined in the previous chapter pretty closely. We employ a Qemu virtual PCI device that sets up structures and mechanisms required for mapping segment memory into guests through its BARs, and for receiving SISCI requests from the guest driver through a communication channel. This channel has been realized through the virtual device itself, rather than as a standalone component. In the Qemu backend, requests are translated into calls to the regular SISCI API and various management related operations. The guest driver receives requests through ioctls from a modified SISCI API which it forwards to the Qemu backend. It also manages and keeps track of open resources in the guest, which reflects the actual SISCI resources allocated on the host.

Our implementation is inspired by the previously mentioned *Nahanni* mechanism (see section 2.6), and we have used a stripped down version of the *ivshmem* device and its guest kernel driver as a starting point for the implementation. Note that the final implementation differs significantly from Nahanni beyond the basic idea of mapping host memory into guests to allow them direct access to it.

In the following, we will describe each component of our mechanism in detail, starting out with the Qemu virtual device.

4.1 Qemu virtual device

Before going into the details of how we have implemented the different mechanisms of our device, a general introduction of Qemu and virtual devices is in order. Qemu has a rich API for adding new devices, based around the Qemu Object Model (QOM). The QOM is an object-oriented framework implemented in C that has a notion of types, classes, inheritance and objects. Through the framework, it is possible to create new devices by defining a *type*, using the `TypeInfo` data structure. A simplified version of the structure can be found in code example 4.1.

The `parent` field of the data structure defines which type the device should inherit from. This is required for all new types, with the base case being inheritance from the `Object` type. The remaining fields are optional; if not set explicitely in the new device's `TypeInfo`, the parent type's values and functions are used. This allows a new virtual device to inherit any desired functionality from its parent, while implementing the specifics of the device where needed. It also means that Qemu can treat the new device as a subtype of the parent type. Futhermore, all types have an associated class, which in turn new objects are initiated from.

```
struct TypeInfo
{
        const char *name;
        const char *parent;

        size_t instance_size;
        void (*instance_init)(Object *obj);
        void (*instance_finalize)(Object *obj);

        size_t class_size;

        void (*class_init)(ObjectClass *klass, void *data);
        void (*class_finalize)(ObjectClass *klass, void *data);
};
```

Code example 4.1: The TypeInfo data structure

If desired, a new class can be created, where device-specific fields or functions are added, or alternatively, one can simply inherit the contents from the parent class. This inheritance is done by copying the parent class into the child class, before finalizing the child class and initializing a new object from it.

How this works is best explained using our own virtual device as an example. The parent type of our device is `PCI_DEVICE`, which in turn inherits from the `DEVICE` type. Initialization and removal of PCI devices happens through the PCI class functions `realize` and `exit`, respectively. In order to override these functions with our own, we simply point to them in our `class_init` function, as shown in code example 4.2. We then set our `TypeInfo` struct

```
static void siscivm_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    PCIDeviceClass *k = PCI_DEVICE_CLASS(klass);

    k->realize = pci_siscivm_realize;
    k->exit = pci_siscivm_exit;
    k->vendor_id = PCI_VENDOR_ID_SISCIVM;
    k->device_id = PCI_DEVICE_ID_SISCIVM;
    k->class_id = PCI_CLASS_MEMORY_RAM;

    dc->reset = siscivm_reset;
    dc->props = siscivm_properties;
```

Code example 4.2: Overriding the default Qemu PCI class functions

to point to our `class_init` function, resulting in it being invoked instead of the standard function inherited from the PCI class.

During initialization we perform a number of operations to set up the structures and functionality required for our device. The details of these will be introduced later in their appropriate sections. As depicted in code example 4.2, we also set our own reset function as well as other fields, including the `properties` for our device. In short, properties are device specific options. As shown in code example 4.3, these can be set as parameters to a device when starting Qemu from the command line. The most important properties for our device are `adapterNo` and `size`, which sets the adapter number for the Dolphin IXH610 adapter and the BAR sizes for our device (see section 4.1.2). Note that the `PCIDeviceClass` and

```
qemu-system-x86-64 [...] -device sisci_vm,size=128MB,adapterNo=0
```

Code example 4.3: Setting device properties when starting Qemu

`DeviceClass` contain several other fields and functions; the ones we set in our `class_init` functions are simply those we want to override.

Some words about device objects are also in order. Qemu must generally be able to dynamically cast objects to its parent type during runtime to access functions inherited from the parent. In our case, it must be able to view our device as a PCI device in order to add and remove it from the PCI hierarchy. To support this, a device object needs to have a parent object as the first member of its structure[1].

We have now introduced the Qemu Object Model and how we create our device. In the following, we will go into the details of its functionality.

4.1.1 Mapping SISCI memory into device BARs

Being able to map memory from SISCI segments into VMs is paramount to our goal of high-performance. This ensures that guests can access segment memory directly, whether the segments are located on the local host or on a remote node. As described in section 3.3, we realize this by exposing segment memory as device memory through our virtual device's BARs.

In Qemu, device BARs can be created through the PCI API. The main function is called `pci_register_bar` and has the following signature:

```
void pci_register_bar(PCIDevice *pci_dev, int region_num, uint8_t
   type, MemoryRegion *memory)
```

As parameters it takes a data structure representing the PCI device that the BAR should be registered to, the desired BAR number, what kind of attributes the BAR should have and a `MemoryRegion` data structure containing the memory that should back it. The two first parameters are self-explanatory, while examples of attributes are whether it is a 64-bit BAR, and where the BAR region should reside in the VM's address space. As mentioned in section 2.4, PCI device resources can either be mapped to the guest's I/O address space or to memory space as an MMIO region. The most important parameter, however is the `MemoryRegion` data structure. MemoryRegions are a core structure in Qemu used to represent various memory that can be made available to the guest.[2] They are created and managed through the Qemu memory API, which offers a rich set of functionality for creating different types of guest memory, including regular RAM, I/O regions and Read-Only-Memory (ROM). In addition, it is possible to create a number of types used for more management related purposes.

To map SISCI segment memory into a device BAR, we need a memory region that represents the memory and associate it with the BAR. For this purpose, we can use the `memory_region_init_ram_ptr` function in the Qemu memory API. This function

[1] C guarantess that the first member of a struct always reside at on offset of zero.

[2] While the actual memory resides in Ram_Block_t data structures, these are mostly used internally in Qemu. For now we therefore limit our attention to MemoryRegions.

allows initializing a memory region from an existing user space pointer, ultimately allowing already allocated host memory to be accessed by the guest. To achieve our goal of mapping segment memory into guests, we can create a new SISCI segment, map its memory into Qemu user space and initialize a memory region with the resulting pointer. The region can then be passed to `pci_register_bar` to create a device BAR backed by segment memory. In our implementation, we utilize the possibilities outlined above, albeit in a somewhat different manner than creating one BAR backed by a single SISCI segment. This will be explained below.

4.1.2 BAR structure

There are a number of ways to set up the virtual device's BARs to facilitate mapping SISCI segments into a guest. The basic approach is, as mentioned, to associate one BAR with a memory region backed by some host memory. This would not be a very flexible solution, however, as a device can only have a limited number of BARs, and BAR properties can not be modified during the devices lifetime. Since we want to map segment memory into guests on-demand, but device BARs are static, we need a way to leave BAR properties unchanged while being able to make underlying changes to what the BARs contain with respect to host memory. We have achieved this by using the *container* memory region type available in Qemu. Like the name implies, containers are memory regions that can hold other memory regions, but which by themselves contain no memory. Containers are initialized with the `memory_region_init` function, which in short takes a `size` parameter. This represents an upper boundary on the combined size of the memory regions that can be held in the container. One can add such regions, referred to as *subregions*, to a container using the `memory_region_add_subregion` function, which is included in the Qemu memory API:

```
void memory_region_add_subregion(MemoryRegion *mr,
   target_phys_addr_t offset, MemoryRegion *subregion)
```

Here, `mr` is the container to add a subregion to, `offset` is where the region should reside in the container, and `subregion` the memory region that should be added. Our approach is to create containers with sizes equal to the maximum amount of segment memory that we want to be able to map into a guest, and register the containers as device BARs. This is done during initialization of the device. When a guest SISCI application later requests to map a local or remote SISCI segment, we map the segment with SCIMapLocalSegment or SCIMapRemoteSegment in the Qemu backend. We then initialize a new memory region backed by segment memory and add it as a subregion to the appropriate container at a runtime determined offset. Upon returning to guest execution, the driver remaps the memory by offsetting into the BAR responsible for that type of memory (see section 4.3). The result is that host memory is dynamically mapped to the device BARs without modifying the actual BARs. How we set up BARs and add subregions is illustrated in code example 4.4 and 4.5, respectively, using the BAR for local segments as an example.

As indicated above and shown in figure 4.1, we employ two BARs for the purpose of mapping segment memory into a guest; one for local segments (BAR2) and one for remote segments (BAR4). The reason for using two BARs is to a degree historical, and partly because

```
memory_region_init(&s->bar2, OBJECT(s), "sisci-vm-bar2-container",
   s->sisci_vm_bar_size);
...
pci_register_bar(dev, 2, attr, &s->bar2);
```

Code example 4.4: Initializing and registering the container for BAR2

```
desc->local_ptr = SCIMapLocalSegment(desc->localSegment,
   &(desc->localMap), vm_packet->offset, vm_packet->size, NULL,
   vm_packet->flags, &vm_packet->error);
...
sprintf(segment_name, "%s%u", "localsegment", desc_id);

memory_region_init_ram_ptr(&(s->localSegmentRegions[desc_id]),
   OBJECT(s), segment_name, vm_packet->size,
   (void*)desc->local_ptr);

memory_region_add_subregion(&s->bar2, vm_packet->bar_offset,
   &(s->localSegmentRegions[desc_id]));
```

Code example 4.5: Adding a subregion to the BAR2 container

it makes managing resources in the guest somewhat easier[3]. Both BARs are set up as MMIO resources to allow regular PIO access from the guest, with additional attributes indicating that they are 64-bit prefetchable BARs. They each have a default size of 128 MB, which, as described earlier, can be overridden when starting Qemu, as long as the size is a power of two.

In addition to the two BARs used for segment memory, we also use one BAR for device registers, and one BAR to pass request parameters and results when forwarding API calls from the guest to the host. How we initialize and employ these BARs will be described in section 4.2.

4.1.3 Unmapping SISCI memory from device BARs

Unmapping memory from a BAR is initially similar to the mapping procedure. Note that the scenario here is that the memory already has been unmapped from the SISCI application in the guest. When the guest driver requests to unmap segment memory from a BAR, `memory_region_del_subregion` is invoked on the host, removing the corresponding memory region from the appropriate BAR container. However, care must be taken before actually unmapping the memory backing the region, from Qemu. In the Qemu memory API [27], both adding and removing memory regions at runtime is in general advised against. With regards to removal, the main reason is that memory regions obviously should not be destroyed while being in use by VM. This could lead to a number of unwanted consequences. As a result,

[3] If desired, one BAR could be employed for both local and remote segment memory, but for now we have stuck with this structure.

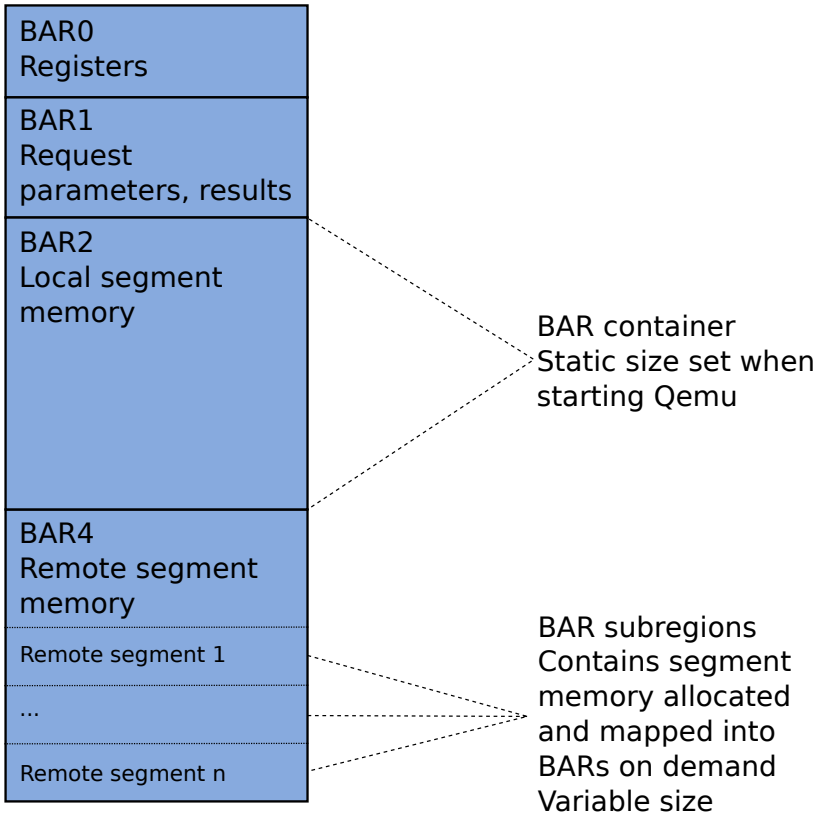

Figure 4.1: Overview of our device's BARs

Qemu's way of handling removals is based on its object model; a memory region is never destroyed while its object is referenced in the object hierarchy. When the object no longer has any references, it is not in use and can be safely removed.

We regard the possibility of dynamically adding and removing memory regions during runtime as a vital part of our solution. Without it, we would have to resort to defining and statically mapping every segment a guest wants to utilize, upon starting Qemu[4]. This would severely limit the usability of our solution. Reading the Qemu Memory API documentation more closely, however, it seems that dynamically adding memory regions is considered safe, as long as the implications of removing them are handled correctly. It can be argued that unmapping segments during runtime is not critical to review the viability of our solution at this point, seeing how we currently do not reuse BAR sections in the guest (see section 4.3.3). On the other hand, leaving segments mapped after they no longer are in use, means that the memory can not be reused by other host or guest applications. In general, this does not represent a solution we are satisfied with.

In the end, we have decided to stick with removing memory regions at runtime, and instead make some minor modifications to Qemu in order to unmap segment memory at the correct time. As with any Qemu object, MemoryRegions has a finalize function that deallocates resources used by the region. It also has a destructor function that is invoked during finalization,

[4]A way to possibly avoid this would be if we somehow could reserve a portion of the PCIe NTBs address space and the local memory pool that SISCI uses, for a guest. By forcing new guest initiated segments to be located in these regions, we could circumvent the need to statically define segments during startup. However, this would definitely require modifications to the SISCI driver and possibly to other parts of the Dolphin stack.

which is responsible for removing the region's memory from the list of available resources in the guest. Memory backing regions, however, is normally unmapped through the use of a Read-copy-update (RCU) mechanism. In short, this mechanism ensures that any references to the memory are resolved, before invoking the function that unmaps the it. In this case, that function is `reclaim_ramblock`, which unmaps memory allocated by Qemu but leaves preallocated host memory untouched, as the source of this memory is unknown.

To unmap SISCI segment memory correctly, one option would then be to modify `reclaim_ramblock` to include a call to `SCIUnmapSegment`. However, this would require modifying several central Qemu interfaces in order to pass required parameters to the SISCI API call. It would also involve introducing the SISCI API to a core mechanism in Qemu. Another approach would be to modify `reclaim_ramblock` to allow a custom unmap operation to be passed as a parameter. This would represent a more generic approach, which could be useful for unmapping other types of preallocated memory as well. In a SISCI context, both these approaches have drawbacks. In a typical SISCI scenario, an application unmaps segment memory, removes the segment and closes the SISCI descriptor before exiting. As defined by the API, removing a segment is not allowed if there exists a mapping to it, while closing a descriptor forcibly unmaps memory stemming from any associated segments. As a result, removing a segment or closing the associated descriptor before segment memory is actually unmapped, should not be allowed. In our view, a mechanism that handles these dependencies would be complex and potentially untidy.

Instead, we have chosen a simpler approach that circumvents these problems, at the cost of a more expensive unmapping procedure. In `reclaim_ramblock`, we set a variable to signal that it is now safe to unmap the memory backing the corresponding memory region. When the Qemu backend of our device receives a request to unmap segment memory, it checks this variable to see if the request can be fulfilled. If it can, `SCIUnmapSegment` is invoked and a status signalling that the request was completed, is sent to the guest driver. In the opposite case, an error status is returned to the guest driver, which waits a while before sending the same request again. This is repeated until the segment memory is successfully unmapped from the Qemu backend. Performance wise, we consider this solution acceptable, as segments are typically unmapped at a non-critical time of a SISCI application's execution[5].

4.1.4 Interrupts

We have implemented an interrupt mechanism in our virtual device to let the Qemu backend asynchronously notify guests of relevant SISCI API events. As touched upon in section 3.4.1, the main motivation behind such a mechanism is that VM exits effectively halt the guest. For instance, if `SCIWaitForInterrupt` API calls in the guest were simply forwarded to the Qemu backend and invoked there, guest execution would be suspended until receiving an interrupt or a timeout occured. Currently, the mechanism is only used to forward SISCI interrupts from the Qemu backend to the guest. However, it could be used for other purposes, such as notifying guests about the completion of a DMA transfer, or events related to remote nodes becoming unavailable. Note that requests from guest applications to *trigger* SISCI interrupts, through calls to `SCITriggerInterrupt`, are handled simply by forwarding the API calls to the Qemu backend.

[5]Preliminary testing has also shown that little time is spent on waiting for memory to be safely unmapped.

In our implementation of this mechanism, we use functionality offered by Qemu to emulate regular pin-based interrupts. The techniques employed are based on *Nahanni's* way of triggering and handling interrupts in guests, which has been extended and modified to our needs. On the host side, we initially configure our device's interrupt pin by calling the `pci_config_set_interrupt_pin` function that is included in Qemu's PCI API. In short, this function writes the chosen pin number to the device's configuration space. Interrupts from our device can then be triggered in the guest by calling `pci_set_irq`, with a pointer to our device object, and the desired *level* the pin should be set to as the parameters. The level is set high to trigger interrupts and low when clearing them, but updates to the actual pin level in the guest only happens when there is a change in the device level, to avoid conflicts when the interrupt line is shared.

To trigger guest interrupts when a SISCI interrupt is received in the Qemu backend, we employ the interrupt callback mechanism in the SISCI API. This allows a callback function and a callback argument to be registered when creating a local interrupt. In turn, the callback function is invoked when an interrupt is received. How we create local interrupts with callbacks when handling a `SCICreateInterrupt` request in the host, is depicted in code example 4.6. As shown, we register our callback function and set the arguments to include our device object and a descriptor id. The latter corresponds to the minor number of the guest descriptor that `SCICreateInterrupt` were invoked on in the guest (see section 4.2).

```
...
case IOCTL_ALLOC_INTFLAG:
    ...
    if ((desc->intcb =
       g_malloc(sizeof(sci_interrupt_callback_args_t))) == NULL) {
        vm_packet->error = SCI_ERR_NOSPC;
        break;
    }

    desc->intcb->SISCIVMState = (void*)s;
    desc->intcb->desc_id = desc_id;

    cb_func = sisci_interrupt_callback;

    SCICreateInterrupt(desc->sd, &desc->local_interrupt,
       s->localAdapterNo, &vm_packet->interruptNo, cb_func,
       (void*)desc->intcb, SCI_FLAG_USE_CALLBACK |
       vm_packet->flags, &vm_packet->error);
    sisci_check(vm_packet->error);
    ...
break;
...
```

Code example 4.6: Creating a SISCI interrupt with a callback

The callback function we employ to trigger guest interrupts is depicted in code example 4.7. As shown, we initially perform some SISCI specific error checking, and if no faults have

```
static sci_callback_action_t siscivm_interrupt_callback(void *arg,
   sci_local_interrupt_t interrupt, sci_error_t status)
{
    sci_interrupt_callback_args_t *intcb_args =
       (sci_interrupt_callback_args_t*)arg;

    SISCIVMDevice *s = (SISCIVMDevice*)intcb_args->SISCIVMState;

    if (status == SCI_ERR_OK) {
        siscivm_IntrStatus_write(s, (intcb_args->desc_id << 16) |
           wait_event_irq);
        return SCI_CALLBACK_CONTINUE;
    }
    else {
        return SCI_CALLBACK_CANCEL;
    }
}
```

Code example 4.7: The host function invoked when a SISCI interrupt is triggered

happened, we call `siscivm_IntrStatus_write` with the provided descriptor id and an event code indicating that a SISCI interrupt has happened. In this function, the interrupt status our device, which simply is a variable in our device object, is set to reflect that an interrupt has occurred on the given guest descriptor. We then invoke `pci_set_irq` and set the level to high in order to trigger a guest interrupt. In turn, our interrupt handler in the guest is executed, which reads the `IntrStatus` register in BAR0 to handle the interrupt (see sections 4.2 and 4.3). On the host this results in an invocation of the `siscivm_intrStatus_read` function, which, in addition to returning the status to the guest, clears the status and calls `pci_set_irq` to set the device interrupt level to low.

Note that SISCI callback functions each run in their own thread, on a per SISCI interrupt basis. This allows for multiple concurrent interrupts to be handled at a time, but also opens the possibility of race conditions occurring. If a new interrupt is received before the guest reads the status of a previous interrupt, information about the latter will be lost. To prevent this, we require a lock, in the form of a semaphore, to be taken before updating the interrupt status in `siscivm_intrStatus_write.` This semaphore is subsequently released by the guest interrupt handler when reading the interrupt status.

4.1.5 Managing SISCI descriptors and resources in the host

As described in section 2.5, all SISCI API resources are related to descriptors, which are represented by the `sci_desc_t` data structure. For the most part, each resource type is limited to one per descriptor, meaning a descriptor can only have one local segment and one remote segment. Since we allocate SISCI resources in the host on behalf of guest applications through forwarded API requests, it is important the state of resources is correctly kept track of across the host and guest. This includes which SISCI descriptors are in use and the resources currently associated with each descriptor. To realize this, a 1-to-1 relationship between SISCI

descriptors in the guest and SISCI descriptors in the host has been implemented. For the purpose of keeping track of descriptors and resources in the host, we have created a `sisci_desc_t` data structure, depicted in code example 4.8. This data structure holds information about one

```
typedef struct sisci_desc_t {
    sci_desc_t sd;
    void *ioctl_args;

    int has_local_segment;
    uint32_t localNodeId;
    uint32_t localSegmentId;
    sci_local_segment_t localSegment;
    uint32_t localSegmentSize;
    sci_map_t localMap;
    volatile void *local_ptr;
    sci_local_interrupt_t local_interrupt;
    sci_interrupt_callback_args_t *intcb;

    int has_remote_segment;
    uint32_t remoteNodeId;
    uint32_t remoteSegmentId;
    sci_remote_segment_t remoteSegment;
    uint32_t remoteSegmentSize;
    sci_map_t remoteMap;
    volatile void *remote_ptr;
    sci_remote_interrupt_t remote_interrupt;
    sci_sequence_t remote_sequence;
} sisci_desc_t;
```

Code example 4.8: Host structure for handling descriptors and associated resources

open SISCI descriptor in the guest, and as shown, it contains a regular `sci_desc_t` descriptor, a dedicated portion of BAR1 for communication purposes (see section 4.2), and other variables used to keep track of any SISCI resources that is associated with a particular guest descriptor. To identify the `sisci_desc_t` data structure of a given guest descriptor, we use the minor number of the device file that the latter is associated with in the guest (see section 4.3), to offset into a table of these structures.

4.2 Guest to host communication channel

The communication channel is in short used to forward SISCI API requests from the guest to the host. Initially we considered implementing it through a standalone Virtio device, but in the end we instead decided to integrate it into our custom device. The main reasons for this choice is that Qemu offer numerous suitable device mechanisms for realizing our communication needs, and that complexity is reduced compared to using a standalone device.

The components used to implement the communication mechanisms are a Qemu I/O memory region and a regular RAM memory region, both exposed to guests as device BARs. These BARs acts as device registers and device memory, respectively. Guests use the registers to request I/O handling in the Qemu backend, while the device memory is used by both to

exchange request parameters and results. An overview of how these components interact can be found in figure 4.2. In the following, we go into the details of these components, on the host

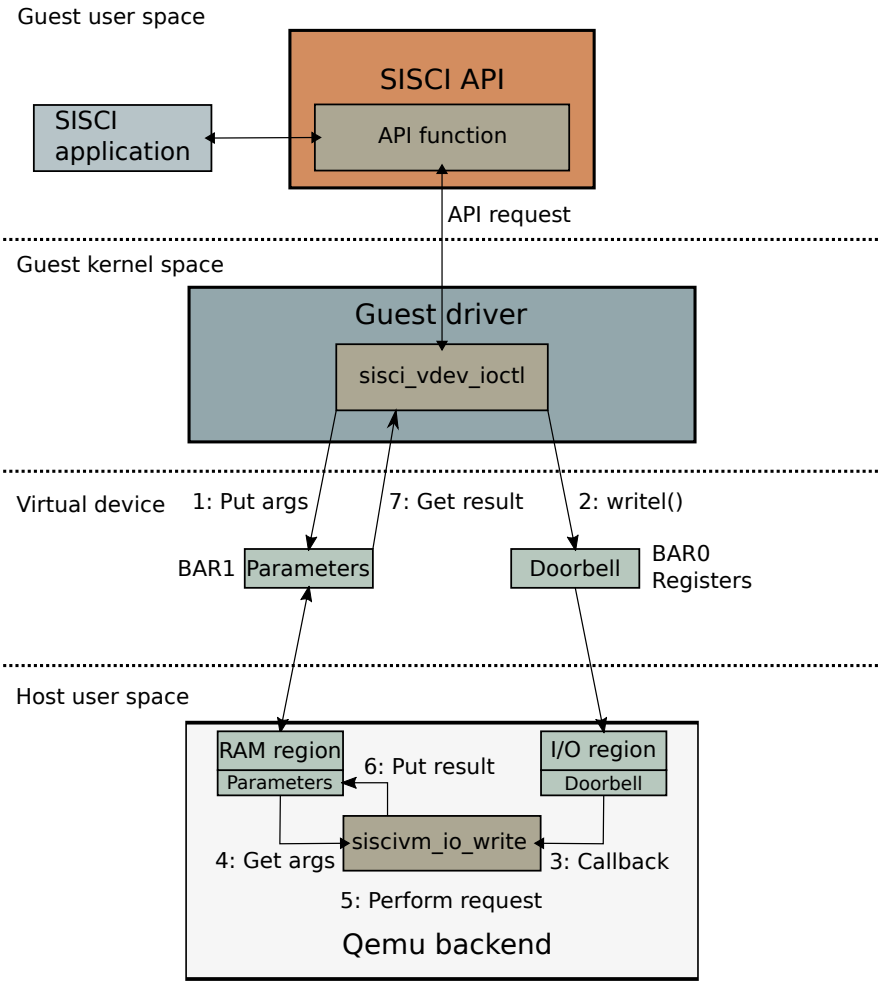

Figure 4.2: Overview of guest to host communication

side (Qemu virtual device backend) and guest side (kernel driver), respectively.

4.2.1 Host side of communication

On the host side, both the I/O and the RAM memory region are set up during initialization of our device. As before, they are registered as device BARs by invoking `pci_register_bar`.

The RAM memory region is exposed to guests as BAR1 and is used to hold request parameters and results. Request parameters are written to the BAR by the guest driver before requesting I/O handling, and subsequently extracted from the memory region by the Qemu backend. Upon finishing an I/O request on behalf of a guest, the Qemu backend correspondingly writes the request results to the RAM region backing the BAR. The BAR is set up in a similar fashion to how we set up the segment BARs. The main difference is that we initialize the region with `memory_region_init_ram`, with the result being that it is backed by memory allocated by Qemu. In addition, we do not use any containers or subregions; it is directly

registered as BAR1. By default the BAR is set up with a size of 256 kilobytes (KB), with each `sisci_desc_t` data structure (see section 4.1.5) having its own dedicated portion of the memory region. This avoids request information being overwritten when handling multiple concurrent requests.

As mentioned, we employ an I/O region to implement device register. In Qemu the I/O memory region type is special, in that accesses into the region causes callbacks to be triggered in the host. I/O regions can be initialized through the Qemu memory API function `memory_region_init_io`, which has the following signature:

```
void memory_region_init_io(MemoryRegion *mr, const MemoryRegionOps *ops,
   void *opaque, const char *name, uint64_t size);
```

The parameter of interest here is the `MemoryRegionOps` data structure, which defines a set of I/O operations:

```
typedef struct MemoryRegionOps {
        /* Read from the memory region. @addr is relative to @mr; @size is
         * in bytes. */
        uint64_t (*read)(void *opaque,
                         hwaddr addr,
                         unsigned size);

        /* Write to the memory region. @addr is relative to @mr; @size is
        * in bytes. */
        void (*write)(void *opaque,
                         hwaddr addr,
                         uint64_t data,
                         unsigned size);
        ...

} MemoryRegionOps;
```

When guests perform a read or write operation in a BAR backed by an I/O region, a VM exit to the Qemu backend will happen, and the corresponding callback defined in the `MemoryRegionOps` data structure will be triggered. We use this mechanism to implement a notion of two device registers, exposed to guests through BAR0. This is done by creating a new I/O memory region with our own set of `read` and `write` functions and registrering it as a BAR. The guest driver then remaps this BAR as MMIO, and upon reading or writing to the registers, I/O handling is triggered in the backend of our virtual device. Note that in contrast to the other device BARs we set up, the BAR is marked non-prefetchable, as accesses to the BAR does have side effects.

The registers are implemented as predefined offsets in the BAR. When handling I/O requests from the guest, we differentiate between the two registers by looking at the `addr` parameter to our read function. The first register is `IntrStatus`, residing at offset `0x04`, which is accessed by the guest when it wants to read the interrupt status of our device. When read, the corresponding callback in the backend of our device invokes `siscivm_IntrStatus_read`, as described in section 4.1.4. As the interrupt status is currently cleared as a part of the read operation, the guest driver does not write to the `IntrStatus` register.

The second register is the `Doorbell` register, residing at an offset of `0x08`. The guest driver writes to this location when it forwards SISCI requestw from the guest API to the host. Code example 4.9 shows an excerpt of how these requests are handled in the Qemu backend, illustrated by a request to create a new local segment. As shown, in the write

```
static void siscivm_io_write(void *opaque, hwaddr addr,
                uint64_t data, unsigned size)
{
    SISCIVMDevice *s = opaque;

    addr &= 0xfc;
    switch (addr)
    {
        ...
        case DOORBELL:
        {
            uint16_t desc_id, request_id;
            vm_ioctl_packet_t *vm_packet;
            sisci_desc_t desc;

            unpack_ioctl_data((unsigned int)data, &desc_id, &request_id);

            switch (request_id) {
                ...
                case IOCTL_MK_LOC_SHM_BUF:
                    // Create a local SISCI segment
                    desc = s->desc_table[desc_id];
                    vm_packet = (vm_ioctl_packet_t*)desc->ioctl_args;
                    desc->localSegmentId = vm_packet->segmentId;

                    SCICreateSegment(desc->sd, &(desc->localSegment),
                        desc->localSegmentId, vm_packet->size,
                        SCI_NO_CALLBACK, NULL, vm_packet->flags,
                        &vm_packet->error);
                    sisci_check(vm_packet->error);

                    if (vm_packet->error != SCI_ERR_OK)
                        break;

                    desc->localSegmentSize = vm_packet->size;
                    desc->has_local_segment = TRUE;

                    break;
                ...
        ...
    }
}
```

Code example 4.9: Handling a request from the guest

callback we initially unpack the `data` parameter (see section 4.2.2) to determine which guest minor device initiated the request, and the type of request. The former is done to identify the `sisci_desc_t` data structure that the request relates to. Information therein is used for error

checking and to locate the SISCI descriptor and resources that are relevant to the request. If the request type dictates any additional parameters, these are then retrieved from the appropriate portion of BAR1. Finally, the operations required to realize the request are executed, which may comprise both calls to the regular SISCI API and Qemu related invocations.

4.2.2 Guest side of communication

In the guest driver we map the two BARs into kernel space during initialization, in order to access them. We initially invoked `pci_resource_start` and `pci_resource_len` to retrieve the I/O address and the size of the BARs, respectively. Since both our BARs represent MMIO resources, we can use the `ioremap` family of functions to remap them. As accesses into BAR0 has side effects, we map it without caching by invoking `ioremap_nocache`[6]. The address returned is later used as a base when offsetting into our two registers. BAR1, on the other hand, is mapped with `ioremap_cache`. We use the address returned from this function when a SISCI descriptor is opened in the guest, more specifically when initializing the corresponding minor device's `Minor_t` struct (see section 4.3.5). At that time, a pointer to the descriptor's portion of BAR1 is then set up, based on this address and the minor number. As a consequence, we write directly to the device BAR when passing request parameters to the Qemu backend. This way of accessing I/O memory is generally advised against, but since we know that the BAR contains regular memory, and that no side effects occur upon reading or writing to it, we can do this safely[7].

When the driver receives a request from the modified SISCI API through an ioctl, it checks which minor device initiated the ioctl and what kind of request it is. Depending on the type, parameters are subsequently written into the minor device's portion of BAR1. For this purpose we have defined a simple `vm_ioctl_packet_t` data stucture, which the guest driver uses to set parameters and the Qemu backend on the host uses to extract them (see section 4.2.3). The guest driver then invokes `sisci_vdev_forward_request` to forward the API call to the Qemu backend. This function packs the minor device number and the request identication number into an integer, before using `writel` to write the resulting value to the `Doorbell` register in BAR0. This notifies the host about a pending request, which the host handles as described in the section 4.2.1. After the request has been handled by the host and control is returned to the VM, we read the request status from BAR1 and depending on the result, either complete the request or return the error status set by the host, to the guest API.

The second register, `IntrStatus`, is only read by the guest during interrupt handling, which we have touched upon in section 4.1.4 and will describe further in section 4.3.4.

4.2.3 Protocol

We have chosen to implement a general data structure for passing parameters when forwarding requests, regardless of the request type. The reason for this is that we wanted to keep requests simple and avoid requiring to handle different structures for each request. As a result, not all members of the structure are used on every request. However, since we only update BAR1

[6] While regular ioremap also uses no caching as its standard mode, we do this in case this changes.

[7] We could also have used the memremap kernel function, which specifically has been purposed to remap resources from I/O space when it is known that the resource is regular memory.

according to the data structure, we do not actually send the complete structure from the guest to the host, there is no overhead in any unused members. The structure is defined as follows:

```
typedef struct vm_ioctl_packet_t {
    sci_error_t error;
    uint32_t bar_offset;

    uint32_t offset;
    uint32_t size;
    uint32_t flags;

    uint32_t result;

    uint32_t localAdapterNo;
    uint32_t nodeId;
    uint32_t segmentId;

    uint32_t timeout;
    uint32_t interruptNo;
} __attribute__((packed)) vm_ioctl_packet_t;
```

Most of the members are self-explanatory when viewed in a SISCI context, with `size` being used both when creating SISCI segments and mapping them, and `result` being used on every request to indicate the resulting status of request. As indicated, the request type has been left out of the structure. Instead, the value we write to the `Doorbell` register comprises both the minor number of the device file through which the request was initiated, and the type of request.

4.3 Guest driver

The guest driver is the main component on the VM side of our implementation. To reiterate, it is mainly responsible for forwarding requests from our modified version of SISCI in the guest OS, to the backend of our virtual device in Qemu. In addition, it must set up the mappings that allow guest applications to access shared memory exposed through the BARs of our virtual device.

Since the virtual device is presented to the guest OS as a regular PCI device, we utilize standard Linux procedures for registering a driver and initializing the device. We have implemented the driver as a Linux kernel module representing a character device driver, which means that access to the driver is provided through *device files*. This is identical to how the SISCI API and driver communicate, where one SISCI descriptor is associated with one device file. As the original SISCI driver, we create 256 device files that the API can use for communication. After opening a device file, various operations can be performed on the file descriptor to communicate with the driver and ultimately the Qemu backend. As mentioned in section 2.3.2, the operations that are valid for a given device driver is defined in a `file_operations` structure, and we define our set in the following manner:

```
static const struct file_operations sisci_vdev_ops = {
        .owner   = THIS_MODULE,
        .open    = sisci_vdev_open,
        .mmap    = sisci_vdev_mmap,
        .unlocked_ioctl    = sisci_vdev_ioctl,
        .release = sisci_vdev_release,
};
```

These functions are invoked when their Linux system call counterparts are called on a device file that our driver is responsible for, e.g., `open(2)` is called on a path name corresponding to one of our device files. During the latter operation we in short check if the minor device is unused, in which case we initialize data structures used to handle information about it (see section 4.3.5 and mark the device as used. When a device file is closed, `sisci_vdev_release` function is invoked. It correspondingly frees up any remaining resources associated with the minor device, both on the guest and the host, and marks it as unused. We will go into the details of the remaining functions later, for now it should be enough to mention that `sisci_vdev_mmap` is utilized to remap local and remote segment memory made available to guests, and that `sisci_vdev_ioctl` is invoked by the SISCI API in the guest to issue requests to the driver.

The file operations structure is set during initialization of our kernel module, when registering our driver as a character device with the `register_chrdev` function. During module initialization we also register our driver as a PCI driver with a call to `pci_register_driver`. The parameter to this function is a `struct pci_driver`, which contain the driver name, pointers to our device identification table and our `probe` and `remove` functions. The device identification table is used to identify which PCI device our driver is responsible for handling, while the probe and remove functions are called during device initialization and device removal, respectively.

4.3.1 Device initialization

During device initialization we perform a number of management related tasks. We initially enable the device by calling `pci_enable_device` and reserve device resources for our driver by invoking `pci_request_regions`. The two BARs used for guest to host communication are then mapped as described in section 4.2.2, and we retrieve the I/O addresses and sizes of the BARs used to contain segment memory. In addition, we set up the mechanisms for handling interrupts from the Qemu backend, and the data structures used to manage SISCI descriptors and resources in the guest. The latter topics will be reviewed later under their appropriate sections. Furthermore, we send an *init request* to the Qemu backend, which retrieves the adapter number of the Dolphin PCIe adapter, and the node identification number for this guest.

4.3.2 Handling SISCI API requests

General API requests are communicated to the driver via the `ioctl` command, which invokes our implementation of the function. It has the following signature:

```
static long sisci_vdev_ioctl( struct file *filp, unsigned int cmd,
    unsigned long arg)
```

Here, `filp` is a pointer to the device file the operation was invoked on, `cmd` contains the request identification code and `arg` can be used to provide data that is relevant to the request. We use a number of request codes to differentiate between the operations that is possible to perform on our device through the SISCI API. Some of these are reused from the original API and some are added to handle the implications of a virtualized environment. Depending on the type of request, `arg` contains a user space address to a SISCI data structure where parameters to the request are found. To limit the modifications to the guest API, our driver accepts the existing data structures exchanged between the original API and the driver. An illustration of how we handle requests from the SISCI API, in this case stemming from the API call `SCICreateSegment`, is depicted in code example 4.10. As shown in the example, we first determine which minor device initiated the request, and perform some error checking. In this specific scenario, we check whether this minor device, corresponding to a SISCI descriptor in the host, already has a local segment associated with it. Furthermore, we use `copy_from_user` to copy parameters from user space into kernel space, and set the request parameters in the `vm_ioctl_packet_t` structure. This function performs some sanity checks on the user space buffer, for instance whether the pointer actually contains a user space address and whether the buffer size is sufficiently large. Note that `vm_ioctl_packet_t` variable directly points to the minor device's portion of BAR1, which the host will read parameters from when handling the request. To forward the request to the host we invoke the `sisci_vdev_forward_request` function as described in section 4.2.2. When control returns to the guest, we check wheter the request was successful, and set relevant information in the minor device's `Minor_t` data structure. This information is used upon further requests on the same minor number, for instance when the SISCI application wants to map the segment that was created.

4.3.3 Mapping device memory into a SISCI application

As described in section 4.1.2, segment memory is mapped on demand during runtime. In the guest, SISCI applications invoke the regular `SCIMapLocalSegment` and `SCIMapRemote-Segment` API calls, which in turn communicate with our driver to perform the requests. As seen in figure 4.3, which illustrates mapping a local segment, this comprises two operations on the driver side; one `ioctl` command to map the segment into the guest as device memory, and one `mmap` operation to map the device memory into the requesting application's address space. The `ioctl` part of the mapping procedure initially performs some sanity checks, such as determining if the minor device in fact has an associated segment. It is also checked whether there is room for a new segment of the requested type in the corresponding BAR. All segment sizes are page aligned to allow remapping them correctly later, and if there is sufficient space we set a page aligned BAR offset for the new segment. The map request is then forwarded to the host as described in the previous section. The host subsequently adds a new subregion to the appropriate BAR at the offset determined by the guest driver. Note that we map only a part of a segment during this phase, if requested by the SISCI application. More specifically, we page align the *size* and *offset* parameters given by the API, and map the corresponding amount

```
static long sisci_vdev_ioctl( struct file * filp, unsigned int cmd,
    unsigned long arg)
{
    Minor_t *minor;
    vm_ioctl_packet_t *ioctl_packet;
    ...
    sci_error_t error = SCI_ERR_OK;

    unsigned int minor_no = MINOR(file_inode(filp)->i_rdev);
    ...
    switch (cmd) {
        ...
        case IOCTL_MK_LOC_SHM_BUF:
            minor = &minors[minor_no];

            if (minor->local_offset != UNINITIALIZED) {
                error = SCI_ERR_ILLEGAL_PARAMETER;
                break;
            }

            copy_from_user((void*)&shmPacket, (void*)arg,
                sizeof(SHMPacket));
            ioctl_packet = (vm_ioctl_packet_t*)minor->bar1_ptr;

            ioctl_packet->segmentId = shmPacket.segId;
            ioctl_packet->size = PAGE_ALIGN(shmPacket.segSize);
            ioctl_packet->flags = shmPacket.flags;

            error = sisci_vdev_forward_request(minor_no, cmd,
                ioctl_packet);
            if (error == SCI_ERR_OK)  {
                minor->local_offset = NOT_MAPPED;
                minor->local_size = shmPacket.segSize;
            }
            break;
        ...
    }
    return error;
}
```

Code example 4.10: Handling an ioctl from the API requesting creation of a new segment.

of bytes from this position into the guest as device memory.

Mapping segment memory into the address space of SISCI applications is handled by our driver's `mmap` function, which is invoked if the ioctl operation succeeds. Our implementation has the following standard signature:

```
static int sisci_vdev_mmap(struct file *filp, struct vm_area_struct * vma)
```

The operation is implemented in a relatively straightforward manner, with modifications in place to take our device's BAR structure into account. As with the `ioctl` function, we

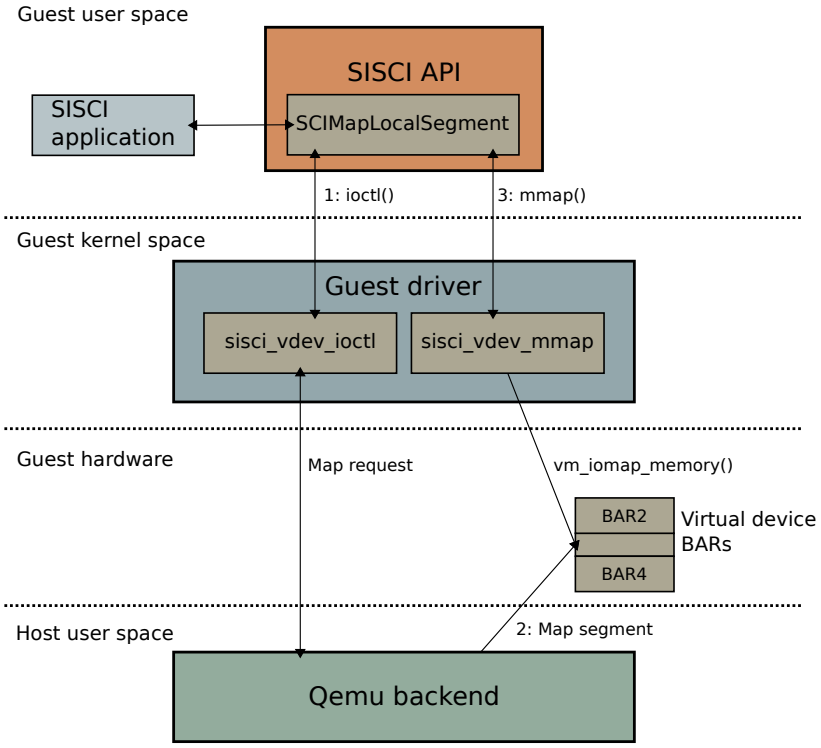

Figure 4.3: Overview of how segment memory is mapped into SISCI applications

determine which minor device initiated the operation by looking at the `*filp` argument. This is done to retrieve the status of the minor device with regards to associated segments. We identify the type of segment, local or remote, by three bits in the `pg_offset` variable contained in the `vm_area_struct`. This technique is taken from the SISCI API and driver, and allows for a simple method to determine which BAR we should map memory from, while requiring minimal changes to the API. Subsequently, we perform checks such as determining if the minor device has a segment of the relevant type ready to be mapped, and if the map request otherwise is valid.

The actual mapping procedure is performed with `vm_iomap_memory`. This is a function that wraps around `remap_pfn_range` and performs a number of error checks that otherwise would have to be done manually. The function has the following signature:

```
int vm_iomap_memory(struct vm_area_struct *vma, phys_addr_t start,
    unsigned long len)
```

As parameters we use the `vma` received through the `mmap` operation, the I/O address of the segment and the requested length of the mapping. Depending on the type of segment that we are remapping, we set the desired cache mode for the mapping before invoking the function. This is done by tuning the `pg_prot` member of the `vma` that represents the virtual memory area. Local segments are backed by physical memory in the host, which means we can use regular write-back caching. Since this is the default caching mode employed in Linux, we leave `pg_prot` untouched in these scenarios. Remote segments can either reside on remote nodes, or on the host local to the guest. We differentiate between these cases by looking at a *loopback*

variable set in the SISCI API when `SCIConnectSegment` is invoked. If they are located on the same node as the guest, they are backed by physical memory on the host and we use write-back caching as described above. In the opposiste case, we employ write-combining by modifying the `pg_prot` member. This is identical to how actual remote segments are mapped in a native environment.

With regard to segment mapping, our current implementation contains some limitations compared to the regular SISCI API and driver. First of all, it is not allowed to map a segment multiple times, e.g., map different portions of a segment. To be able to access multiple sections of a segment, a SISCI application must map the whole segment and offset into the desired sections. In addition, we do not reuse sections of the BARs when segment memory is unmapped from them. This means that currently the BAR sizes represent an upper limit of how much segment memory guests can utilize. While these limitations would be relatively easy to remove, they do not effect our ability to review the viability of our proof-of-concept. As a consequence, we regard them as acceptable for now.

4.3.4 Interrupt handling in the guest

In the following, we will describe how the guest driver handles interrupts from the Qemu backend, and how the implemented interrupt mechanism conforms to the mechanism offered by the SISCI API and driver. As mentioned in section 4.1.4, guest API calls to *trigger* interrupts are simply forwarded from the guest driver and will not be the topic of discussion here.

In order to receive interrupts from our virtual device, the guest driver needs to request an interrupt line and register an interrupt handler. We do this by invoking the `request_irq` function during driver initialization, which has the following signature:

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long
    flags, const char *name, void *dev);
```

Here, `irq` represents the Interrupt Request (IRQ) number we want to request, and `handler` should point to the function that acts as an interrupt handler. The former has been negotiated by the PCI subsystem and is provided to us through the `struct pci_dev` data structure during device probing. Before describing how our interrupt handler is implemented, we should introduce how the guest driver handles requests stemming from the `SCIWaitForInterrupt` API call. There are two main scenarios in these cases. The first scenario is that the SISCI application has specified an infinite timeout using the `SCI_INFINITE_TIMEOUT` macro, meaning it should wait indefinitely until an interrupt occurs. Alternatively, it has specified a given time interval, which if exceeded, means that execution should continue whether an interrupt has happened or not. In both situations, we employ a `wait queue` to allow the application to sleep while waiting for an interrupt. In short, wait queues are lists that can be used to keep track of processes that are sleeping while waiting for certain events. Subsequently, the queues can be used to wake the processes when a relevant event occurs. In our driver we utilize the `wait_event_interruptible` and `wait_event_interruptible_timeout` functions to make processes sleep, with the former having the following signature:

```
wait_event_interruptible(queue, condition);
```

The parameters corresponds to the wait queue which the process should be added to, and the condition that the process waits for to become true. The corresponding function for waking processes is `wake_up_interruptible`, which takes a wait queue as its only parameter. We utilize the aforementioned wait functions to allow SISCI applications to wait for an interrupt indefinitely and for a given time interval, repectively. In both cases, the condition they wait for is that their `interrupt_event` variable becomes non-zero. This variable exists on a per `Minor_t` data structure basis, and is used to only wake up the process that is waiting for an interrupt on the corresponding minor device.

To process interrupts stemming from our virtual device and wake up any waiting SISCI applications, we have implemented a relatively straightforward interrupt handler. When an interrupt occurs, we initially read the `IntrStatus` register to retrieve the interrupt status of our virtual device. This is done both to determine if the interrupt stems from our device, since regular pin interrupts may be shared, and to retrieve and clear the status. In the case of no interrupt, we simply return `IRQ_NONE`. If an interrupt has happened, we unpack the status variable, which, as described in section 4.1.4, contains the guest minor number that the interrupt occurred on and the actual interrupt status[8]. We then set the `interrupt_event` variable of the `Minor_t` data structure that belongs to the minor number and call `wake_up_interruptible` on our wait queue. If a SISCI application is currently waiting for an interrupt on the device file corresponding to that minor number, it will be woken up and continue execution.

4.3.5 Managing SISCI descriptors and resources in the guest

As described in section 4.1.5, we employ a 1-to-1 relationship between SISCI descriptors in the guest and SISCI descriptors in the host. To manage descriptors and resources in the guest, we take advantage of the fact that device files are used for communication between the SISCI API and the driver. Each device file can belong to one open SISCI descriptor and has a device minor number. In the driver, we use these minor numbers to manage and identify the resources of the corresponding descriptors. This is done by associating each minor number with a data structure called `Minor_t`, which is depicted in code example 4.11. The structure contains various information about the state of a minor device, most notably which resources the corresponding descriptor has allocated through it. As most of the members are self-explanatory, we will not explain them further here. To keep the 1-to-1 relationship between descriptors and resources in the host and the guest, we include the minor number of the descriptor that a SISCI application initiated a request on, when forwarding API calls to the host. This lets the Qemu backend easily identify the corresponding host descriptor it should perform the request on, and allows the state of resources to be synchronized across the host and the guest.

4.4 Modified SISCI API

The last component in the guest is a modified SISCI API that runs on top of the our driver. As previously described, we have sought to realize our implementation in a manner that keeps

[8] As of now this is redundant, as we only use interrupts for forwarding SISCI interrupts. In the case of DMA support, this could be used to differentiate between a SISCI interrupt event and a DMA event, for instance a DMA transfer completion.

```
typedef struct Minor {
    unsigned int open_count;
    struct semaphore sema;

    void *bar1_ptr;

    s32 local_offset;
    unsigned64 local_size;

    s32 remote_offset;
    unsigned64 remote_size;
    int remote_loopback;

    int has_local_interrupt;
    int has_remote_interrupt;

    int interrupt_event;
    int dma_event;

    int has_sequence;

    MAPPacket mapPacket;
    SHMPacket shmPacket;
    INTPacket intPacket;
} Minor_t;
```

Code example 4.11: Guest driver structure for handling minor devices and associated resources

modifications to the API to a minimum. While this is mostly relates to the interface our guest driver presents to the API, we have also tried to retain the semantics of each API call throughout our stack, from the guest API to the Qemu backend in the host. The result is that one API call in the guest equals one call to the regular API in the host, for the most part. Furthermore, all the standard API interfaces are intact, with modifications being limited to the body of specific functions. This, together with the fact that SISCI can be linked as a dynamic library, means that in principle, binary compatibility with existing SISCI applications is achieved. We have, however, found it necessary to leave out some of the API functionality in the guest, compared to the regular API. The reason is mainly the time set aside for the project, which has led us to focus on core functionality. As a consequence, we currently support a subset of the original API features. These limitations will be presented shortly.

The most important factor in reducing modifications to the SISCI API is that the guest driver has been implemented to conform to how the original SISCI API and driver communicate. As a result, no data structures have been added to the API, and the structures are predominantly set and used in the same manner as before. Instead of ioctl operations being directed to the SISCI driver, we send them to our new driver, while retaining the format and data structures of the requests. We have, however, added some new ioctl command codes in addition to the ones that the API already employ. The reason is that we sometimes need to perform some additional steps, especially when mapping segment memory into a guest. As described in section 4.3.3, we invoke `mmap` to remap the memory into SISCI applications. This is the same approach used in the original API, and while parameters to the mmap operation are slightly modified,

the techniques employed are otherwise identical. A testament to this is that we use the original SISCI internal function `unix_unmap` to unmap memory from applications.

The API modifications we have performed are all encapsulated in the C preprocessor statements `#ifdef` and `#else`. This allows us to define what functionality we want on compile time, according to the current environment. An illustration of how we have used these statements to modify the API, can be found in code example 4.12. As depicted, the statements revolve around a new C preprocessor flag that we have introduced, the `SISCI_VM` flag. The flag signals that the API will be used in a virtualized environment, and is set by adding `CPPFLAGS='-DSISCI_VM'` to the command line when configuring the SISCI package. When defined, the API includes code that makes it communicate with our driver instead of the regular SISCI driver, and leaves out code that is not applicable to a virtualized environment. In the opposite case, the API is built as normal.

The flag is also used to make API functionality that is not supported by our implementation, unavailable. In these cases the `SCI_ERR_NOT_IMPLEMENTED` error value is returned to the SISCI application, signaling that the functionality of the API call is not implemented. Currently, the most important functionality not supported is DMA. A simple DMA mechanism could be implemented by forwarding the relevant API calls to the host. This would in practice be a DMA transfer initiated by the host, with the only difference being that the VM has set the contents of source segment. As a result, the bandwidth would be equal to native plus the extra overhead incurred from the forwarding procedure. An alternative would be to implement support for directly initiating transfers from the guest, i.e., without host involvement. This could prove a more complex task, as we would have to program the DMA controller in the Dolphin adapter from the guest. It would also require an IOMMU, for which support is not yet fully implemented in the SISCI API and the SCI driver stack. For these reasons, and the time set aside for the project, we have decided to leave DMA capabilities out of our implementation for the time being. There are also additional functionality not currently implemented, including data interrupts and a number of management related operations, which fall beyond the scope of this thesis.

It should also be noted that while we do support error checking through the `Sequence` mechanisms offered by the API, our implementation is somewhat limited from a performance perspective. Currently, we forward the `SCICheckSequence` API calls from the guest to the host, and have the host return the status of the data transfer to the guest. The obvious drawback of this approach is that a VM exit is performed for each error check. As a consequence, frequent error checking leads to a significant performance overhead. We did, however, mostly implemented this feature as an additional method of verifying that our data transfers complete successfully. In practice, errors are scarce when using a PCIe based interconnect over shorter distances, and vigorous error checking is not really necessary to accomplish our goal of implementing a proof of concept. If we were to implement a more performance focused mechanism, we could exploit the fact that the SISCI API maps error register associated with each mapped segment, and checks these to determine if any errors have occured during data transfers. Similarely to how we map segments, we could possibly map these registers into guests to perform error checking without involving the host.

```
volatile void *SCIMapRemoteSegment(sci_remote_segment_t segment,
                                   sci_map_t            *map,
                                   size_t               offset,
                                   size_t               size,
                                   void                 *addr,
                                   unsigned int         flags,
                                   sci_error_t          *error)
{
    ...
    (*map)->rseg = segment;
    (*map)->lseg = NULL;
    (*map)->map_flags = 0;
#ifndef SISCI_VM
    memcpy(&(*map)->mapPacket.sp, &segment->segmentPacket,
       SIZEOF(segment->segmentPacket));
#endif
    (*map)->mapPacket.offset = (unsigned64)offset;
    (*map)->mapPacket.size = (unsigned64)size;(*map)->mapPacket.flags =
       flags;
    (*map)->mapPacket.mp_addr = (void *)addr;
    ...
#ifndef SISCI_VM
    *error=kcMmapShm(segment->sd->handle, &(*map)->mapPacket);
#else
    *error = ioctl(segment->sd->handle, IOCTL_MAP_REM_SHM_BUF,
       &(*map)->mapPacket);
#endif /* SISCI_VM */

    if (SCIIsError(*error)) {
        free(*map);
        return NULL;
    }

    /* The IOCTL_MMAP went ok */
    ...
}
```

Code example 4.12: Illustration of how the SISCI API has been modified in the guest

4.5 Summary

In this chapter, we have described how we implemented an IVMC mechanism based on the SISCI API that allows both co-located and remote guests to communicate through shared memory. We have presented the details of a new Qemu virtual device that acts as the backend of our solution, more specifically, how it maps SISCI segment memory into guests through device BARs and receives forwarded API calls from a guest driver. In addition, we have described how a modified SISCI API in the guest communicates with the this driver. In the next chapter, we will evaluate our implementation by looking at the performance it offers, potential shortcomings, and what could have been done differently.

Chapter 5

Evaluation and Discussion

During the design and implementation process, we have continually evaluated our choices and made changes when necessary. Our primary concern has been to validate that our implementation performs as desired with regard to correctness and performance, that it is sensible in a virtualization context, and that it integrates well with the SISCI API both on the host and the guest side. Since our goal has been to implement a high-performance IVMC solution, a significant amount of time has been spent on getting our segment mapping procedures to perform as desired, so that guests can access the memory directly. To that end, we have debugged Qemu and KVM extensively, made changes to our implementation and evaluated the performance at every turn.

This chapter is structured as follows. In the next section, we briefly introduce our evaluation environment, before describing how we have validated our solution with respect to correctness in section 5.2. The benchmarks we have performed and the corresponding results are presented in section 5.3, and in section 5.5, we discuss these results and some aspects of our solution that had impact on them.

5.1 Evaluation environment

As shown in figure 5.1, our benchmarks have been run on and between two machines acting as SISCI nodes, each running a Qemu/KVM guest. We have used Ubuntu 16.04 LTS as the Linux distribution on both host and guests, with the hosts using Linux kernel version 4.8-RC2 to include the patch that enables KVM to handle page faults stemming from physically remapped memory (see section 3.3)[1]. Our main test machine has an Intel Core i5-4590 processor with four cores running at 3.3 GHz. The CPU has 6 MB shared L3 cache and supports virtualization extensions, including Intel EPT (see section 2.1.2). In addition, the machine has 16 GB of DDR3 1600 MHz RAM in a dual channel setup, and a Dolphin IXH610 card running in x8 mode, i.e., using 8 PCIe lanes. The other machine has the same specifications and setup, except 8 GB of RAM. Furthermore, the adapters have been set up with a direct node-to-node topology, i.e., no PCIe switch is utilized between our nodes. The guests have each been given one CPU core and 4 GB of RAM, and a virtual device with BARs that can contain 128 MB of local and remote segment memory, respectively.

[1] Kernel version 4.8, which includes the patch, has since been released.

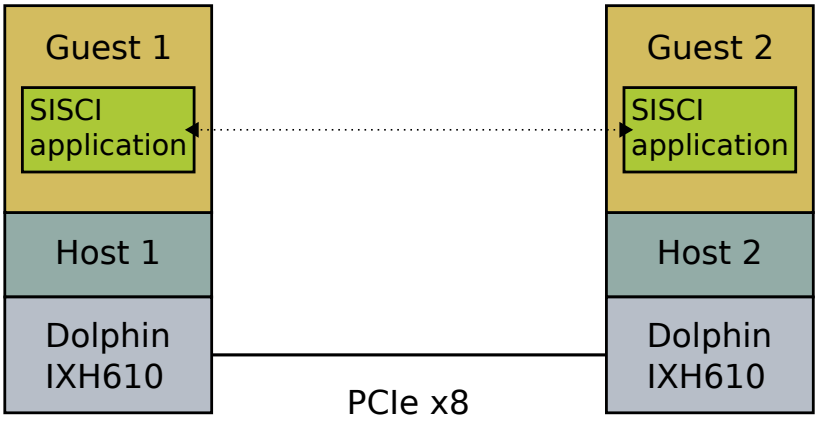

Figure 5.1: Overview of our test setup

5.2 Correctness

Throughout this project, we have continually conducted small tests to determine if our solution works as desired with regard to correctness. In the context of this project, we primarily define correctness as successfully transferring data between SISCI segments, without any errors or data loss. In addition, it is important that other mechanism such as interrupts, mapping procedures and management operations work as intended. With regards to the latter category, we have performed extensive error checking in a number of different scenarios to determine if our solution works correctly. While we do no claim that it is perfect, we have concluded that it has reached a sufficient level of robustness for a proof-of-concept.

To check that data transfers complete successfully, we have made a small test application that uses the SISCI API to transfer files between both local and remote guests. Subsequently, the GNU utility *sha1sum* has been employed to verify that the hash of the source file matches the hash of the resulting file on the destination guest. This has proved to be the case in all our tests, both for transfers between local guests and remote guests[2]. As a result, it has been redundant to use the `Sequence` mechanism in the SISCI API to check for errors, and we have only occassionaly utilized our implementation of it for this purpose.

5.3 Performance

The most important performance metrics for a shared memory based IVMC mechanism are memory bandwidth and memory latency. Memory bandwidth can be defined as the maximum rate that data move between two memory locations, and is often measured in MB/s. We define memory latency as the amount of time it takes for a write operation to become visible either in cache or memory. Applied to this project, these metrics indicate how fast data can be moved from one guest to another. Optimally they should be in range of the native performance that the host offers, and in our case, the overall goal has been to achieve similar bandwidth and latency as SISCI applications running directly on the host. Note that the SISCI API offers significantly

[2]On a hardware-level this can be explained by the very short PCIe cables that we have used between the adapters in our test machines. In such situations errors are very scarce in PCIe.

higher bandwidth for write operations than for read operations[3]. As such it employs, and is based on using, write as the standard direction for data transfer. All our benchmarks have therefore been performed by writing data into segments, rather than reading from them.

In addition, these two metrics we will review the latency of SISCI interrupts, as they are commonly used to perform synchronization between applications. An otherwise fast IVMC mechanism would be of less value if synchronization added a significant overhead. We define interrupt latency as the time elapsed between triggering an interrupt in one SISCI application and the other application receiving it.

The performance of management related operations has not been a focus during our evaluation. By such operations, we mean initializing memory segments and interrupts, mapping segments, teardown, and other operations that are typically performed once during startup or upon application exit. As they are not, or should not be, frequently used during runtime, they have little impact on the overall performance and are of lesser importance when determining the viability of our solution.

We have mostly used standard benchmarks included in the SISCI API to assess memory bandwidth and latencies, both in a non-virtualized environment and in virtualized environments. This allows for more comparable results than if we were to utilize a different set of benchmarks in guests. The main results presented are the averages from running each benchmark five times, with all benchmarks having inner loops that perform operations repeatedly to increase the accuracy of measurements. We also include some additional data that describes the results in more detail, such as overhead percentage and the standard deviation (σ) for the set of results from each benchmark. Note that the tests have been performed with a minimal load on both hosts and guests to capture the full performance potential of the mechanism.

With regards to terminology, we use the term *native* when talking about performance in a non-virtualized environment, i.e., a benchmark that has been performed in the host. Running the same benchmark in guests yields the corresponding *virtualized* performance.

5.3.1 Memory bandwidth

We compare the memory bandwidth of native and virtualized SISCI applications by employing the `scibench2` benchmark included in the SISCI API. This benchmark measures PIO bandwidth by writing (or reading) different amounts of bytes into a remote segment and recording the elapsed time for each transfer size. It outputs the average bandwidth in MB/s and the average time it took to perform each transfer. To increase the accuracy of measurements, each operation is performed several times in a loop, by default 40000. The segment can either reside on a remote node or a local node, and in the latter case, *loopback* (see section 2.5.1) is performed internally on the machine. Furthermore, a number of operations can be used to move the data, including regular `memcpy` and the `SCIMemcpy` function included in the SISCI API. Since we do not support the latter, our measurements stem from using `memcpy`[4]. The benchmark is run by starting it on a server node and a client node, with the former being responsible for creating the segment that the client node transfers data to. Below is an illustration of how to start the benchmark and the most important parameters.

[3]This is related to how PCIe handles read and write operations. In short, reading requires waiting for a transaction confirmation, while writing does not.

[4]Native tests have shown that bandwidth is similar regardless of the function employed.

```
scibench2 -rn remoteNodeId (-client | -server) [-size segment_size]
    [-loops n] [-copymode (m|s|d)]
```

There are two required parameters, *remoteNodeId* and either *-client* or *-server*, while the default values of the remaining parameters can be overridden if desired. We run our benchmarks with a *size* of 8 MB and with `copymode -m`. The former results in transfer sizes corresponding to all values that are a power of two in the interval between 4 byte and 8 MB, while the latter indicates that `memcpy` should be used to move data.

While `scibench2` should be able to capture the maximum available bandwidth when running in a native environment on a single host, i.e., with loopback, we have created a similar benchmark that is stripped of all SISCI functionality, for reference. The benchmark, called `memcpy_bench`, simply copies a given amount of data between two memory locations on the host using `memcpy`, with some small optimizations identical to ones used in `scibench2`. As the latter, we perform the copy operation several times in a loop for each transfer size and measure the resulting bandwidth. This gives us a baseline to compare our results to, whether the SISCI applications are run natively or virtualized. The results from this benchmark are found in figure 5.2 and table 5.1, where we look at the bandwidth for transfer sizes ranging from 1KB to 8 MB. From the results we can see that the reported bandwidth for smaller transfer sizes is

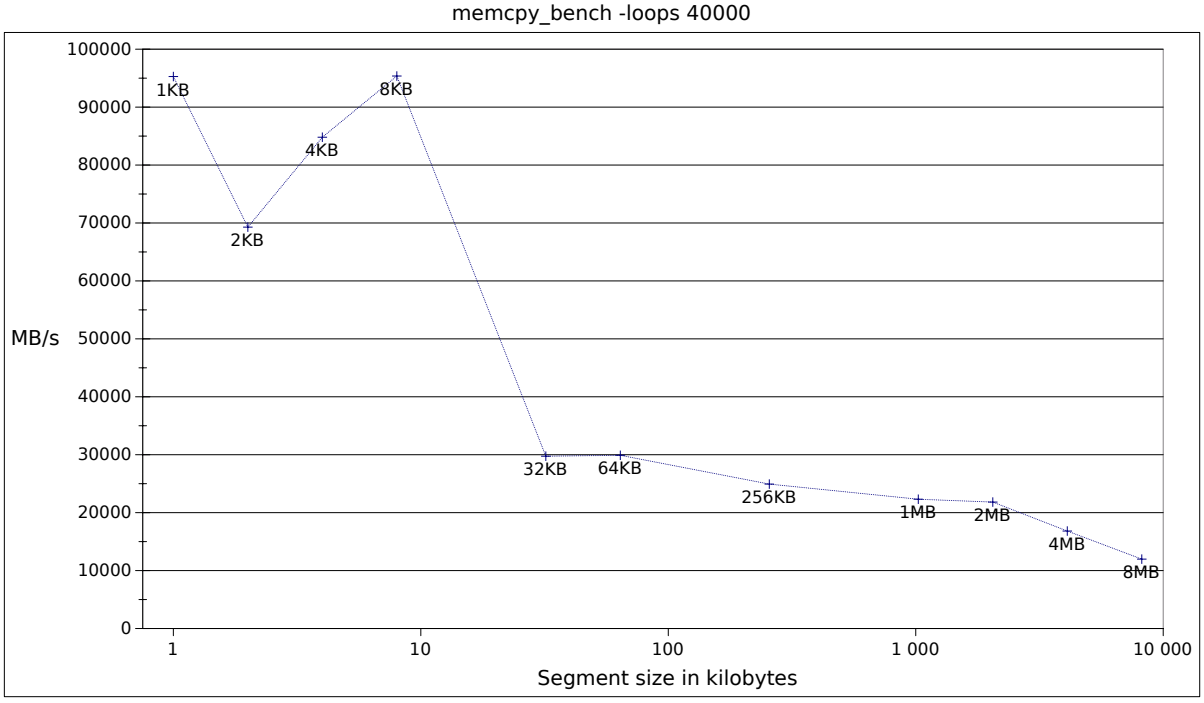

Figure 5.2: Native memory bandwidth in our main test machine

1KB	2KB	4KB	8KB	32KB	64KB	256KB	1MB	2MB	4MB	8MB
95279	69285	84804	95368	29743	29901	24930	22318	21826	16825	13227

Table 5.1: Native memory bandwidth results (MB/s)

significantly higher than the theoretical maximum of our RAM setup. This stems from caching effects, as the number of bytes transferred fits in the various caches of the CPU. In addition to having 6 MB shared L3 cache between cores, our CPU has 32 KB L1 cache and 256 KB L2 cache, each per core. Presumably, transfers up to 8 KB are being performed continually on L1 cache. There is a drop in bandwidth from 1 KB transfers to 2 KB, before increasing and peaking at 8 KB transfers. We assume this pattern stems from compiler optimizations, as we, similarly to `scibench2`, use the `-O2` flag. Without any optimization flags, the graph is similar, except that bandwidth for 1 KB transfers are somewhat lower than the one for 2 KB transfers. For the next transfer sizes, L2 cache might be sufficiently large to hold data for transfers up to 64 KB. Furthermore, we assume L3 cache is employed for the subsequent sizes up to 4 MB. Although the L3 cache is initially big enough to hold all the data for the 4 MB transfer, there is a noticeable bandwidth drop from 2 MB to 4 MB. Possibly, data for this transfer size is only partially resident in L3 cache during the benchmark. For the largest transfer, the bandwidth is significantly lower than the theoretical maximum of our RAM, which may be attributed to the data set being larger than all the caches. As a consequence, it will continually force a number of cache operations, which impacts the bandwidth negatively [28]. We will not discuss this further, as the point was not to speculate how the CPU performs caching during our benchmark, but rather having a performance baseline to compare our results to.

The corresponding results from running `scibench2` in a guest and on its host, e.g., transferring data from the guest to a remote segment on the host using loopback, is depicted in figure 5.3 and table 5.2.

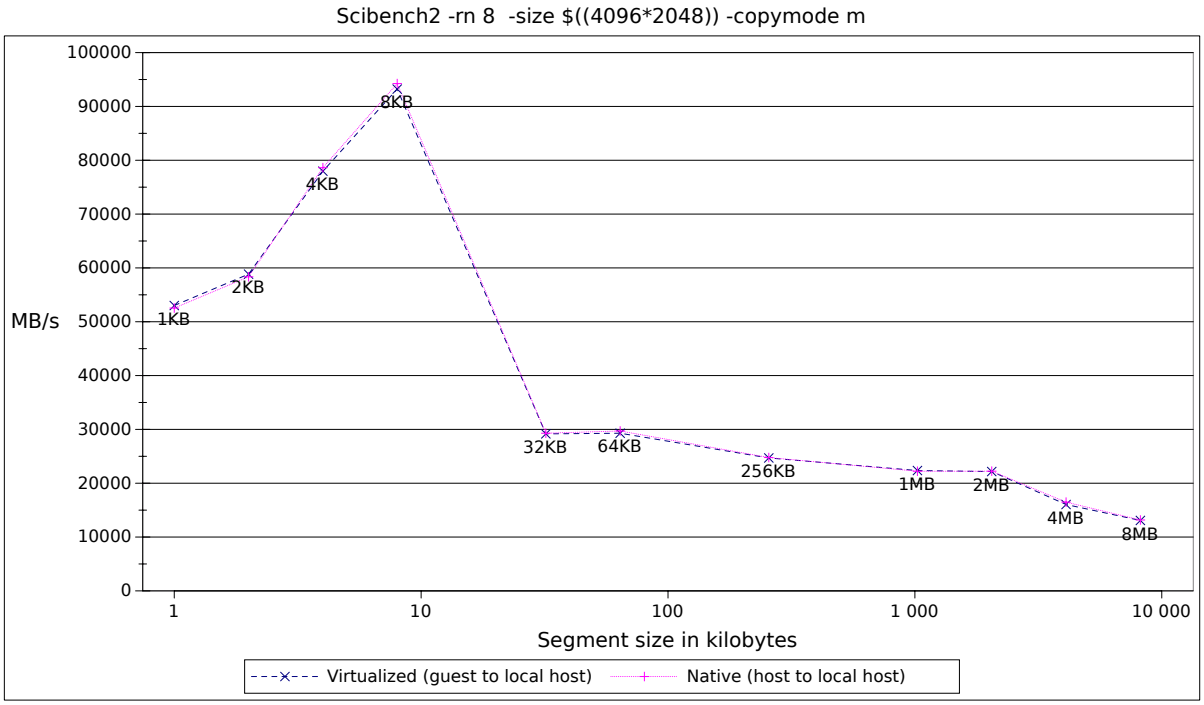

Figure 5.3: Guest to local host bandwidth

As the results show, running `scibench2` from a guest to its host, yields a bandwidth that is very close to the results from running it between native applications. The overhead ranges from -0.89 to 2.68 percent, and while some variance must be expected, it indicates that guest

	1KB	2KB	4KB	8KB	32KB	64KB	256KB	1MB	2MB	4MB	8MB
Native	52541	58351	78625	94248	29288	29707	24772	22229	22239	16486	13171
Virtualized	53010	58841	77971	93222	29171	29286	24694	22355	22170	16045	13085
Overhead %	-0.89	-0.84	0.83	1.09	0.40	1.42	0.32	-0.57	0.31	2.68	0.65
Virtualized σ	1142	653	983	1217	65	423	367	91	173	210	283

Table 5.2: Guest to local host bandwidth results (MB/s)

applications can utilize the shared memory with near native performance[5]. The reason for these numbers is that once the first round of page faults has been handled by KVM, memory accesses to SISCI segments are done through the EPT mechanism, adding very little overhead compared to native memory accesses. While the guest and host bandwidth reported by `scibench2` is similar, for some of the smaller segments it is signicantly lower than the one achieved in the `memcpy_bench`. We investigated the cause of this, and found the overhead to primarily stem from the `SCIFlush` API call invoked in the `scibench2` benchmark. For smaller transfer sizes, the cost of this call greatly impacts the bandwidth. In short, the function flushes write combining buffers in the CPU, which in remote scenarios is necessary to be absolutely certain that all the data has left the CPU. In local use cases, this should normally not be required, at least on the x86 architecture. However, we decided to leave the benchmark unmodified to simpliy switching between the two scenarios during our tests.

The results from running the same benchmark between two co-located guests are shown in figure 5.4 and table 5.3. In practice, this benchmark is very similar to the previous, as the

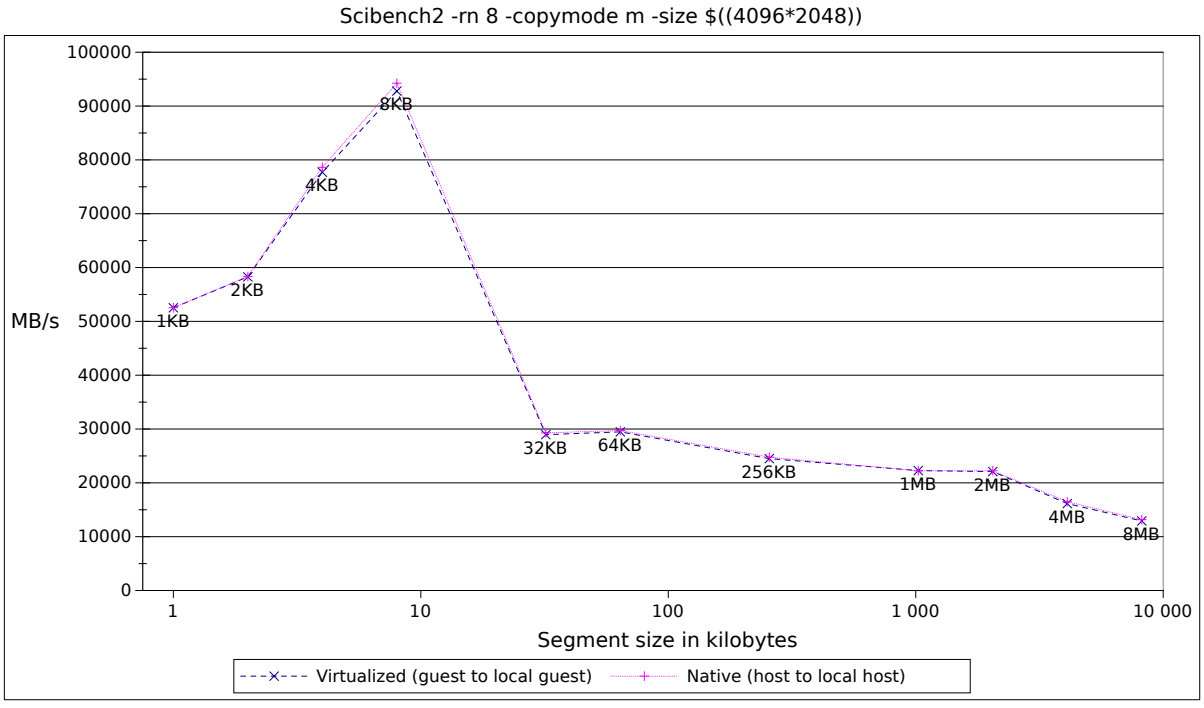

Figure 5.4: Guest to co-located guest bandwidth

[5]Note that the bandwidth numbers are rounded for presentation purposes, while the percentages are based on the raw data.

	1KB	2KB	4KB	8KB	32KB	64KB	256KB	1MB	2MB	4MB	8MB
Native	52541	58351	78625	94248	29288	29707	24772	22229	22239	16486	13171
Virtualized	52525	58270	77710	92789	28953	29474	24501	22281	22091	16150	12920
Overhead %	0.03	0.14	1.16	1.55	1.14	0.78	1.10	-0.23	0.66	2.04	1.91
Virtualized σ	835	529	1202	1403	309	356	417	105	331	147	408

Table 5.3: Guest to co-located guest bandwidth results (MB/s)

guest writes into a SISCI segment located on the host. The only difference is that the other local guest was responsible for initializing the segment, and that the benchmark applications both run in VMs instead of one running on the host. As a consequence, the results are comparable to those achieved above, with similar bandwidth and overhead. Running the benchmark in two co-located VMs does introduce the performance isolation aspect, but it is difficult to say anything conclusive based on these numbers. For the most part, the bandwidth achieved is close to those in the guest-to-host scenario. This may change if more VMs are deployed on the host, but as we use a standard version of KVM, the performance isolation should be similar to the one achieved in other scenarios involving intensive memory operations in KVM guests.

From figure 5.5 and table 5.4 we see that the results from running the benchmark between two remote guests show a similar pattern.

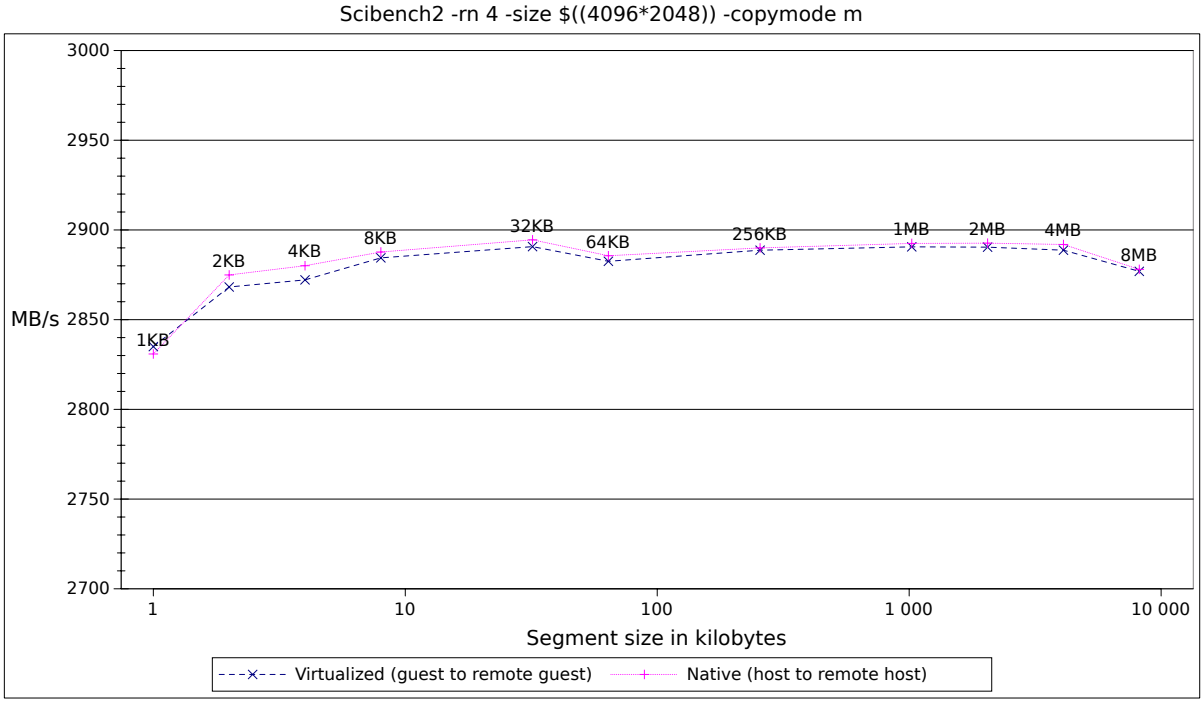

Figure 5.5: Guest to remote guest memory bandwidth

As shown, the native bandwidth is somewhat lower than the IXH610's theoretical maximum of around 3000 MB/s. We assume this is the maximum bandwidth achievable between our test machines when using the IXH610 adapter, as native DMA benchmarks yields similar results. More importantly, we achieve a remote bandwidth in the guest that closely follows the native

	1KB	2KB	4KB	8KB	32KB	64KB	256KB	1MB	2MB	4MB	8MB
Native	2831	2875	2880	2888	2894	2886	2890	2892	2893	2892	2878
Virtualized	2835	2868	2872	2884	2891	2883	2889	2891	2890	2889	2877
Overhead %	-0.15	0.23	0.27	0.11	0.13	0.11	0.04	0.07	0.08	0.11	0.04
Virtualized σ	3.5	10.7	12.9	2.6	3.3	3.5	2.2	1.3	1.3	1.9	8.7

Table 5.4: Guest to remote guest memory bandwidth results (MB/s)

remote bandwidth for all transfer sizes. The overhead ranges from -0.15 to 0.27 percent, which is more or less negligible. As before, this is attributed to the fact that guests directly operate on the segments, and the memory access performance that KVM delivers when EPT is enabled.

In addition to this, we also ran some preliminary remote benchmarks using the PXH830 PCIe adapter, for which the results can be found in figure 5.6 and table 5.5. This adapter

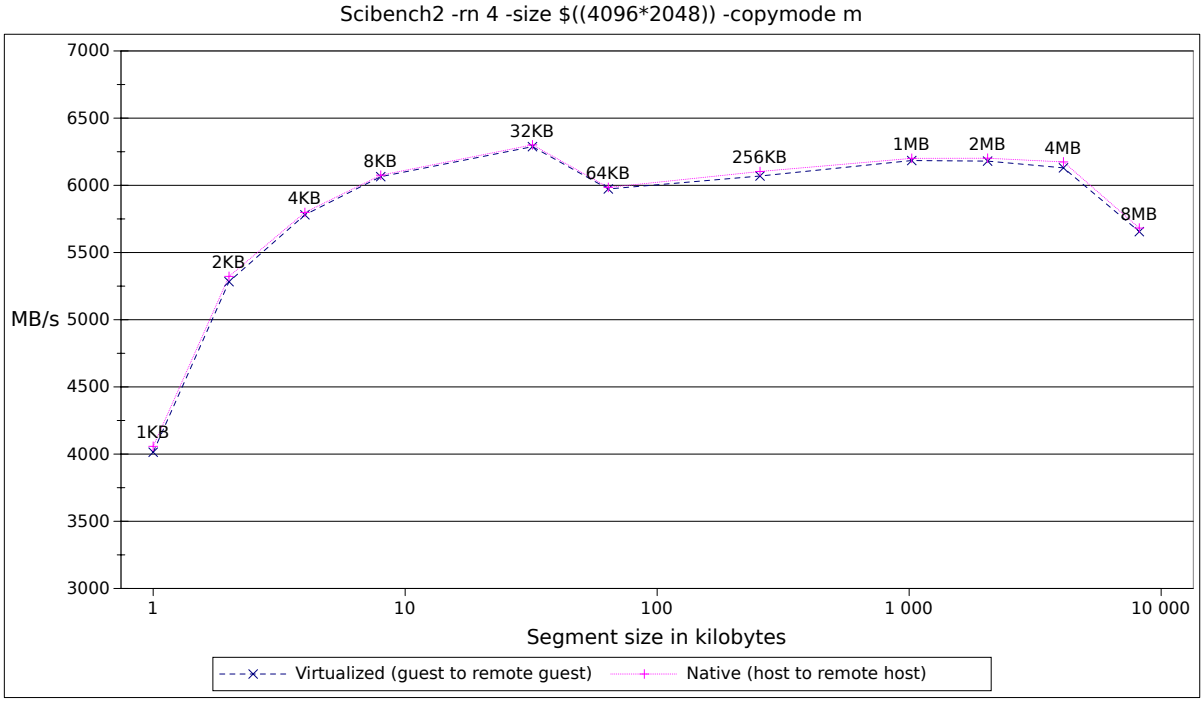

Figure 5.6: PXH830 guest to remote guest bandwidth

	1KB	2KB	4KB	8KB	32KB	64KB	256KB	1MB	2MB	4MB	8MB
Native	4058	5322	5798	6075	6301	5985	6103	6201	6202	6174	5682
Virtualized	4014	5283	5781	6065	6288	5973	6069	6184	6180	6130	5655
Overhead %	1.08	0.74	0.30	0.18	0.21	0.19	0.55	0.26	0.35	0.71	0.47
Virtualized σ	51.4	35.9	21.6	17.7	7.8	28.9	23.5	10.7	12.7	27.2	79.4

Table 5.5: PXH830 guest to remote guest bandwidth (MB/s)

uses PCIe Gen 3 and 16 PCIe lanes, allowing a maximum remote bandwidth of around 11000 MB/s when using DMA. For PIO transfers, the bandwidth is depends on the CPU employed. Note that these tests were run on a different set of physical machines, but with identical CPUs

and similar memory configurations. As figure 5.6 and table 5.5 show, employing the PXH830 adapter allows for a maximum remote bandwidth of around 6300 MB/s both between guest applications and native applications. Moreover, the trend from previous benchmarks, that guest bandwidth closely follows the native, is present also when utilizing an adapter that offers higher performance.

Overall we see that our procedure of mapping segment memory into guests allows for near native bandwidth both in local and remote scenarios. Next, we will review memory latency.

5.3.2 Memory latency

Memory latency has been evaluated through the `scipp` benchmark included in the SISCI API. This application calculates memory latency for transfers of varying sizes by utilizing a so-called *ping-pong* mechanism. The client and the server side of the benchmark take turns in copying *size* bytes to one another, with the last 4 bytes of each transfer signalling the other side that the operation is complete. This memory location is polled for a predefined value by the other, and when received, it correspondingly repeats the transfer the other way. To get an average time for each exchange, the elapsed time for exchanging messages of a given size is recorded on the client side, and divided by the number of *loops*. Results are thus comparable to the round-trip time of the transfer, which is divided by two to get an approximation of the memory latency. The benchmark is invoked in the following manner, with some advanced options omitted for simplicity:

```
scipp -rn remoteNodeId (-client | -server) [-loops n] [-copymode (d|s)]
    [-errcheck]
```

We run the benchmark with the standard parameters, meaning data are copied in regular for-loops and without checking for errors. Furthermore, the number of exchanges per size are left to the default value of 10000, and we only compare the memory latency between two remote guests and two remote hosts, respectively, with measurements in microseconds (µs). Results are found in figure 5.7 and table 5.6.

Size (bytes)	4	8	16	32	64	128	256	512	1024	2048	4096	8192
Native	0.710	0.712	0.715	0.719	0.757	0.780	0.827	0.914	1.075	1.429	2.138	3.572
Virtualized	0.709	0.711	0.714	0.719	0.758	0.781	0.828	0.916	1.076	1.433	2.140	3.574
Overhead %	-0.113	-0.056	-0.084	0.056	0.158	0.103	0.194	0.153	0.168	0.252	0.094	0.073
Virtualized σ	0.003	0.003	0.005	0.006	0.003	0.000	0.001	0.001	0.001	0.006	0.001	0.001

Table 5.6: Remote memory latency results (µs)

From the graph, we see that the latency from guest to guest is very close to the native case, with overhead percentages ranging from -0.113 to 0.252. It is difficult to deduce any patterns from these small deviations, but we did perform some additional benchmarks with a *histogram* option enabled. Output is then more detailed, with best and worst case latencies for each transfer size, among other things. Note that running the benchmark with this option does add some overhead, and the following must be read with this in mind. The results show that the worst case latencies in the guests *at times* are significantly higher that those in a native environment. Typically, the native worst cases lie around 50 µs for some transfer sizes, while the guest worst cases

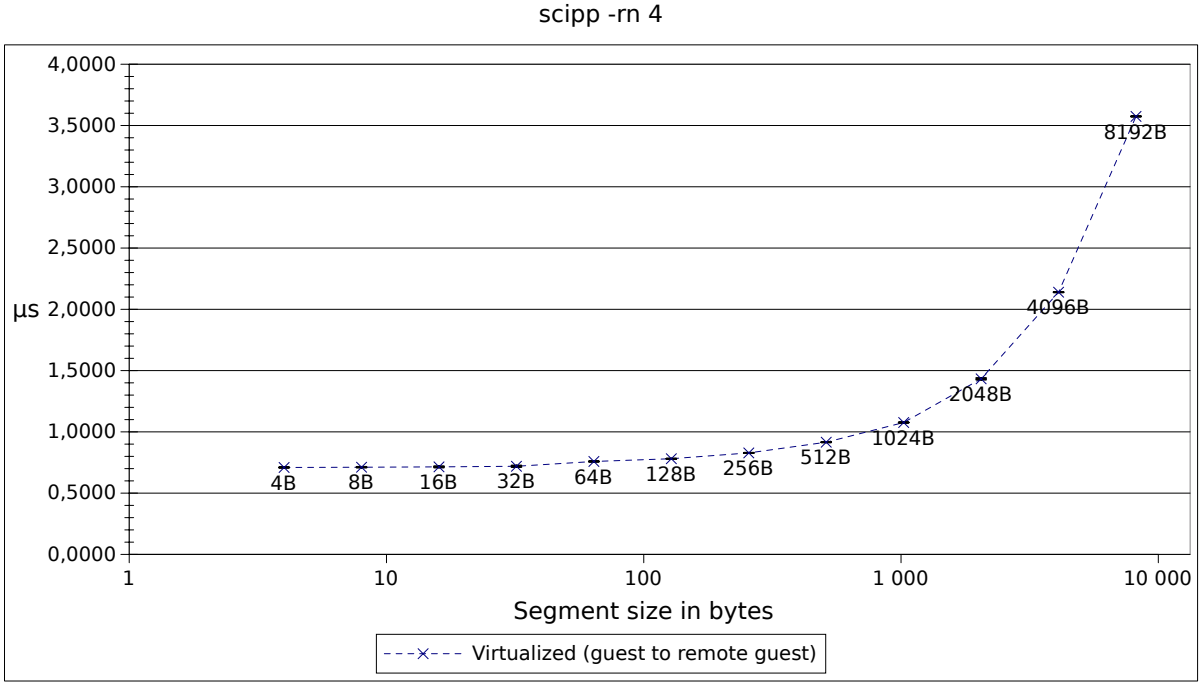

Figure 5.7: Remote memory latency

sometimes are an order-of-magnitude higher. This varies from one benchmark to another; they are equally often more or less identical to the native ones. The source of these significantly increased worst cases is hard to pin-point. The EPT mechanism do introduce an additional page-walk, which means the cost of any TLB misses are greater than in a native environment. However, it is difficult to imagine any TLB misses when we are acting on a maximum of two pages, especially since the TLB entries for guests are not necessarily flushed upon a VM exit when using Intel CPUs [29] [30]. A more likely explanation is that it stems from other factors, such as scheduling on the host or other, general virtualization implications that it is not immediately easy to identify. As the worst cases also differ significantly between benchmarks, the issues discussed above should not be taken as a definitive explanation or conclusion.

While the worst case memory latency occasionally is higher for guest applications than for native applications, it is nearly identical for the most part. In the next section we will review interrupt latency.

5.3.3 Interrupt latency

To review interrupt latency we employ another benchmark included in the SISCI API, namely `intr_bench`. As in `scipp`, latency is measured using a ping-pong mechanism, where the average round-trip time is calculated by triggering a given number of interrupts between to SISCI applications. The average latency is approximated by dividing the round-trip time by two. The benchmark is started using the following command:

```
intr_bench -rn remoteNodeId (-client | -server) [-loops n] [-inner]
```

Most of the parameters will be familiar to readers, except the `inner` option. This option enables a more detailed analysis that uses a histogram to report the frequency of latencies in each 10 µs interval up to 300 µs, as well as best and worst cases. Figure 5.8 shows different scenarios where `intr_bench` is run with this option disabled.

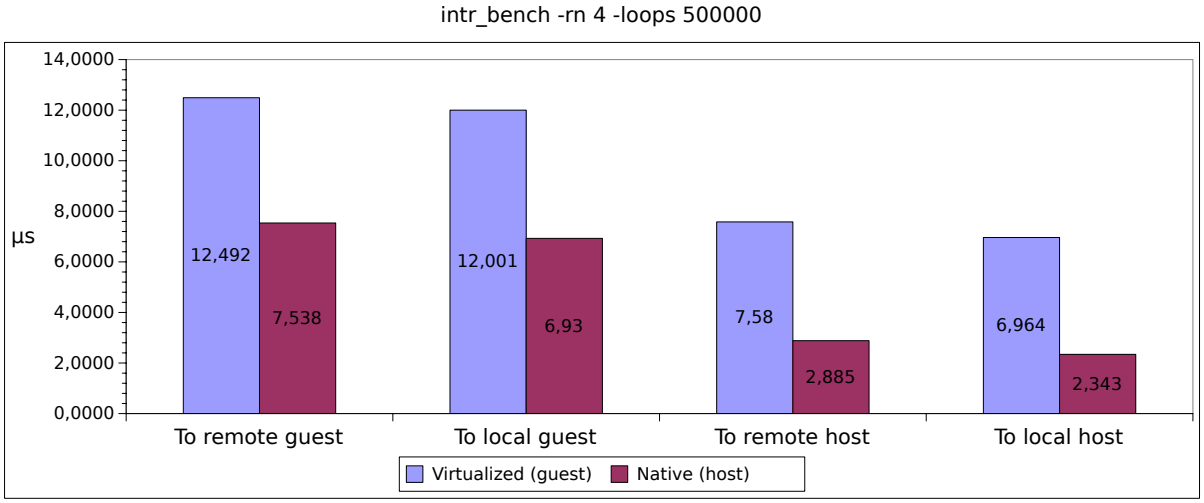

Figure 5.8: Interrupt latency overview

	Remote	Local
Native (host to host)	2.885	2.343
Virtualized (guest to guest)	12.492	12.001
Overhead %	332.95	412.15
Virtualized σ	0.067	0.059

Table 5.7: Interrupt latency results (µs)

From the results we see that interrupt latency is significantly higher in a virtualized environment than in the native case. Guest to remote guest interrupts are over four times slower than the the ones from host to remote host. Latency between co-located guests are somewhat lower, similar to how local and remote interrupt latency differ in a native environment, but with a larger relative overhead.

At the same time, the unit of measure is µs. While an increased overhead of 300 to 400 percent is initially massive from a relative viewpoint, around 9.6 µs of added latency is not completely unacceptable. Some overhead is arguably inevitable when virtualizing a high-performance communication mechanism. In our case, the extra overhead in guests may stem from a number of factors. One is undoubtedly that we add an additional layer of interrupt handling in software, on top of the SISCI mechanism. The callback that is triggered when the real SISCI interrupt is received in the host, performs several operations to trigger an interrupt in the guest. Both the callback mechanism in itself, and these operations, add overhead compared to the native case. The cost of handling interrupts in our guest driver is also not free. In the guest interrupt handler, we perform a VM exit to read the interrupt status before waking up any applications that are waiting for an interrupt. In addition, VM exit is required to trigger interrupts. VM exits may be costly depending on the host architecture, adding to the time

it takes before the SISCI applications are woken up in the guest, and remote nodes receive interrupts, respectively. In an attempt to evaluate the performance of our interrupt mechanisms more precisely, we decided to measure the cost of VM exits. We implemented a small test that triggers 10000 Qemu I/O callbacks in the host, where control is immediately returned to the guest driver. To measure the average time elapsed we employed the `ktime` mechanism in Linux. The average VM exit time from 10 runs proved to be 2.48 µs, which explains some of the overhead compared to native SISCI interrupts. It is clear, however, that the added layer of interrupt mechanisms still play a significant part of the total overhead.

We have also run `intr_bench` with the `inner` option enabled, giving us an overview of the best and worst case interrupt latencies. Similar to memory latency, the worst cases for guest interrupts are sometimes significantly higher than the native worst cases, which typically lie below 300 µs. This is not frequently occurring, with the results showing that, on average, 99.78 percent of the latencies are in the 10-20 µs interval, while only 0.0016 percent are over 310 µs, equaling 8 interrupts out of 500000. However, while memory operations are performed as normal after mappings have been set up, interrupts are delivered and received through our own interrupt mechanism. Therefore, we investigated whether the latter contained any errors that could be the source of these numbers. We did not manage to find any evidence that this was the case. As before, we consequently believe that the increased worst cases mainly stem from general virtualization performance factors, rather than being specific to our implementation.

In general, we see that the interrupt latency between guest applications is significantly higher than between native applications. To circumvent the increased latency, it is possible to use PIO and SISCI segments to synchronize applications. This can be done by employing the same polling mechanism that is used in `scipp` benchmark, resulting in a synchronzation latency that is similar to memory latency. For this reason, it is actually advised by Dolphin to use memory segment polling for synchronization instead of interrupts, in some scenarios. However, as polling represents a *busy-wait* mechanism, it does increase CPU usage, which means it is not viable in all situations.

5.4 Security

While the primary goal of this project has been to implement a high-performance IVMC mechanism, we have attempted to keep security concerns in mind during the implementation. We will not delve into a detailed security analysis of our mechanism here. However, we do not believe that our mechanism introduces significant security issues compared to similar solutions.

In general, the techniques we employ are mechanisms that already exist in Qemu and KVM, which other Qemu virtual devices also utilize. Since we map segments on demand into guests, there is no automatic mechanism that shares a given amount of memory between guests. The overall integrity of the guests are intact as the segment memory resides in the host; inherently there is no added way for a guest to access memory belonging to another guest. Furthermore, guests are only given access to memory granted through the regular SISCI API and the SCI driver stack on the host. Since segment memory is initially remapped to Qemu user space before being mapped into device BARs, KVM sets up the mappings for the guest based on what can be accessed from the Qemu. As such, there is not static range of physical memory that guests can access through our mechanism.

On the guest side, we use standard mechanisms available in Linux to remap the memory to

user space applications. More specifically applications can request to map memory through the `mmap` operation of our driver interface. In the driver this leads to an invocation of `remap_pfn_range`, which allows us to only remap the memory ranges that the applications should have access to. In addition, we have attempted to expose as little information from the driver as possible. SISCI applications use the API as originally to request functionality and gain access to SISCI resources, and it is transparent to both the API and applications how the driver communicates with the Qemu backend.

Analyzing security issues stemming from the Qemu mechanisms we employ and KVM itself falls beyond the scope of this thesis. Reviewing such matters, and a more detailed analysis of our mechanism with regards to security, is left to further work.

5.5 Discussion

In summary, we achieve near native memory bandwidth and latency in both local and remote guest-to-guest scenarios. This is possible through our approach of remapping segment memory from the host to guests, and the performance of memory address translations when using KVM with Intel EPT. However, there are indications that the memory performance varies more in a virtualized environment than in the native. We believe this stems from the virtualization process itself, rather than having a reason that is specific for our mechanism. The impact of these variations on our results are also limited, in that the overall performance is very similar to native. Subsequently, the variations do not define the performance that is achievable. In general, the memory performance indicates that an IVMC mechanism based on the SISCI API is a viable option for high performance use cases where a shared memory API may be employed.

On the other hand, interrupt latency between guests is noticeably higher than the native latency. In contrast to memory operations, triggering and receiving interrupts involves both the guest driver, and the SISCI API and drivers on the host. As a consequence, the interrupt path between guests are longer when compared to native, involving additional layers of software. While we believe this is acceptable in a proof-of-concept, optimally guest-to-guest interrupt latency should be closer the native latency. One way to achieve this could be to have a more direct path between guests and the SISCI and SCI drivers residing on the host. Currently, any API call in our solution comprises two calls to the SISCI API, one in the guest and one in the host. We also up data structures and arguments two times, of which the guest parameters are only temporarily used. If we could get the guest driver to communicate more directly with the SCI drivers on the host, interrupt latency could possibly be decreased. In general, this could also have positive effects with regards to concurrent SISCI applications in a guest. Forwarding API calls involves VM exits, which may be detrimental to the performance of other SISCI applications currently executing. In our implementation, we have tried to limit the amount of time spent in the host while handling I/O requests, but a more direct path could reduce this even further.

Another approach for lowering the interrupt latency, would be to allow guests to directly communicate with the Dolphin PCIe adapters. To trigger interrupts, the SCI driver stack write to certain registers in the adapters, and if we could map these registers into VMs, interrupts could be triggered without any host involvement. This would involve moving functionality from the SCI driver stack to a guest driver, which means some control over the adapters is decentralized. With the proper mechanisms in place, however, it could be a viable approach.

Despite the increased interrupt latency between guests, we do believe that an IVMC mechanism based on the SISCI API could be of value in high performance scenarios. The memory performance is very close to native, and as described in 5.3.3, PIO can be used for synchronization purposes to circumvent the higher interrupt latency. There may also be unrealized potential in how interrupts are delivered to and handled in guests.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we have investigated how VMs can be extended with shared memory mechanisms from the SISCI API to allow high performance communication in local and remote scenarios. To that end, we have reviewed different approaches to realize a unified IVMC solution based on the API, which utilizes local shared memory for co-located VMs, and memory exposed through PCIe NTBs for remote communication.

Based on using KVM and Qemu as a combined hypervisor, we found that a custom virtual device is optimal to realize our goal of a high performance mechanism. By remapping both local physical memory and remote memory to guests through the BARs of a Qemu virtual PCI device, guest applications can operate directly on the memory without host involvement. We implemented a dynamic mapping scheme, where SISCI segment memory are mapped on demand, rather than being defined statically on VM startup. This allows fine-grained control over what memory is shared between VMs and variably sized memory regions.

To utilize the our IVMC mechanism, guest applications employ a slightly modified SISCI API that interfaces with a new guest driver. The latter communicates with the Qemu backend to forward API calls invoked by guest applications, which includes requests related to shared memory and other, more general API functionality.

We have evaluated our proof of concept by using benchmarks included in the original SISCI API. This has allowed us to make a direct comparison between the performance achieved through our IVMC mechanism, and the performance that the SISCI API offers in a native environment. The results show that the performance is very close to native. Between co-located VMs, the achieved memory bandwidth is within 2 percent of the native bandwidth. In remote scenarios, this overhead is even smaller, at just over 1 percent. Memory latency between remote VMs is nearly identical with the native latency, although some more variance seem to be present with regards to worst cases. In contrast, synchronization between guest applications, in the form of interrupts, introduces a significant overhead relative to the native case. The absolute average overhead is less than 10 µs, however, which we believe is acceptable for a proof of concept.

In conclusion, we believe that our prototype shows that an IVMC solution based on the SISCI API is a viable option in scenarios requiring high bandwidth and low latency. A mechanism such as this, may increase the feasibility of employing VMs in high performance use cases that otherwise are suited to a virtualized environment, with high-availability servers and electronic trading as possible examples.

6.2 Main Contributions

We have shown that a unified shared memory IVMC solution, using a PCIe interconnect for remote memory, is capable of delivering near native memory bandwidth and latency. Through our proof of concept based on the SISCI API, both co-located and remote VMs can seamlessly utilize high performance communication mechanisms without the overhead associated with more traditional networking mechanisms, such as TCP/IP.

VMs on the same physical computer use local shared memory to communicate, which translates to a PIO bandwidth that is only around 2 percent lower than the native performance. In remote scenarios, the overhead reported by our tests reaches 1 percent, with a maximum bandwidth of around 2900 MB/s for the IXH610 PCIe adapter that we have primarily employed during testing. With the new PXH830 adapter, the achievable remote bandwidth between two guest applications increases to around 6300 MB/s. Furthermore, our mechanism offers a remote memory latency that is nearly identical to the native, with latencies as low as 0.7 µs.

To facilitate these communication capabilities, we expose shared host memory to guests, allowing them to directly operate on the memory without host involvement. More specifically, we employ a dynamic mapping procedure, where memory is mapped to the BARs of a Qemu virtual PCI device on demand. This circumvents the need to define a specific guest's requirements with regards to shared memory.

In addition, binary compatibility with existing SISCI applications is ensured, by utilizing a slightly modified SISCI API in guests. This guest version of the API has all the original API interfaces intact.

6.3 Future work

While our proof of concept provides mechanisms that allow guest applications to communicate through shared memory and synchronize their execution, we do not claim it to be complete with regards to functionality nor optimizations.

The former primarily relates to the mechanisms offered in the SISCI API. Our prototype currently supports a subset of the API functionality, with DMA mechanisms being the most important part missing. We have previously outlined how a simple forwarding technique could be employed to implement DMA transfers. An alternative to this would be to initiate DMA transfers without host involvement, which will be possible once IOMMU support for the SCI driver stack is released. Other functionality not currently implemented includes data interrupts and a number for management related operations, which can be forwarded from guests like other API calls.

In addition, exploring how VMs running SISCI applications can be successfully migrated is integral to increase the viability of the mechanism. Today, migration support is a feature that is more or less expected to be present in software that is tailored to virtualized environments. The security implications of our mechanism should also be reviewed in more detail, to assess its viability in scenarios requiring a high degree of isolation.

There are also several possibilities related to optimizing the current mechanism. Interrupt latency could possibly be lowered by either having a more direct path between the guest driver and the SCI driver stack, or by avoiding host involvement altogether. The latter might be achieved by mapping the PCIe adapters' registers responsible for handling interrupts, into guests. This would allow guests to trigger, and possibly receive, interrupts directly through

the PCIe device. A similar approach could also be employed to reduce the overhead related to error checking. By giving the guests access to the PCIe adapters' error registers, they can read the status of transfers directly, without involving the host.

Appendix A

Source code

The source code is available as a git repository at:

`https://bitbucket.org/mpg_code/sisci-vm`

To get access to the repository, send an e-mail to `hckiella@ifi.uio.no`.
The source code is structured as follows:

DIS/ Contains the DIS package where the SISCI API is included

guest/ Guest code, including our device driver

qemu/ A modified Qemu version where code for our virtual device is found

The source code for the parts of the API that we have modified, are found in the `DIS/src/SISCI/api` folder. Most of our modifications can be found by looking for statements that involve the `SISCI_VM` macro. Moreover, our guest driver reside in the `guest/driver` folder, as `sisci_vdev.c`. Finally, the code for our Qemu virtual device is located in `qemu/hw/misc/siscivm.c`.

Instructions on how to use our implementation can be found in the `README` file of the git repository.

Bibliography

[1] Understanding Full Virtualization, Paravirtualization, and Hardware Assist. White paper, VMware Inc., 2008.

[2] *Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide, Part 1*. Intel Corporation, 2011.

[3] Vijay Kumar. `https://en.wikipedia.org/wiki/PCI_configuration_space#/media/File:Pci-config-space.svg`. Accessed: 2017-02-04.

[4] Thomas J. Bittman, Mark A. Margevicius, and Philip Dawson. Magic Quadrant for x86 Server Virtualization Infrastructure. Technical report, Gartner Inc., 2014.

[5] Thomas J. Bittman, Philip Dawson, and Michael Warrilow. Magic Quadrant for x86 Server Virtualization Infrastructure. Technical report, Gartner Inc., 2016.

[6] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Linwood Griffin. XenSocket: A High-throughput Interdomain Transport for Virtual Machines. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, pages 184–203. Springer-Verlag New York, Inc., 2007.

[7] Yi Ren, Ling Liu, Qi Zhang, Qingbo Wu, Jianbo Guan, Jinzhu Kong, Huadong Dai, and Lisong Shao. Shared-memory optimizations for inter-virtual-machine communication. *ACM Comput. Surv.*, 48(4):49:1–49:42, February 2016.

[8] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, January 1989.

[9] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, June 2007.

[10] Dan Marinescu and Reinhold Kröger. State of the art in autonomic computing and virtualization. Technical report, Wiesbaden University of Applied Sciences, September 2007.

[11] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson Education Limited, third edition, 2014.

[12] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[14] Rusty Russell. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[15] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 41–46, 2005.

[16] Dolpin Interconnect Solutions. SISCI API specification. `http://ww.dolphinics.no/download/SISCI_DOC_V2/index.html`. Accessed: 2017-01-14.

[17] Kangho Kim, Cheiyol Kim, Sung-In Jung, Hyun-Sup Shin, and Jin-Soo Kim. Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 11–20. ACM, 2008.

[18] François Diakhaté, Marc Perache, Raymond Namyst, and Herve Jourdren. Efficient shared memory message passing for inter-vm communications. In *Euro-Par 2008 Workshops – Parallel Processing*, volume 5415 of *Lecture Notes in Computer Science*, pages 53–62. Springer Berlin Heidelberg, 2009.

[19] A. Cameron MacDonnell. *Shared-Memory Optimizations for Virtual Machines*. PhD thesis, University of Alberta, 2011.

[20] Qi Zhang and Ling Liu. Shared memory optimization in virtualized cloud. In *Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing*, CLOUD '15, pages 261–268. IEEE Computer Society, 2015.

[21] Shesha Sreenivasamurthy and Ethan L. Miller. Sivshm: Secure inter-vm shared memory. Technical Report UCSC-SSRC-16-01, University of California, Santa Cruz, May 2016.

[22] C. Pinto, B. Reynal, N. Nikolaev, and D. Raho. A Zero-Copy Shared Memory Framework for Host-Guest Data Sharing in KVM. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, pages 603–610, July 2016.

[23] Todd Deshane, Zachary Shepherd, Jeanna N. Matthews, Muli Ben-Yehuda, Amit Shah, and Balaji Rao. Quantitative Comparison of Xen and KVM. In *Xen Summit*, pages 1–3, June 2008.

[24] Steve G. Langer and Todd French. Virtual Machine Performance Benchmarking. *Journal of Digital Imaging*, 24(5):883–889, 2011.

[25] Charles David Graziano. A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project. Master's thesis, Iowa State University, 2011.

[26] J. P. Walters, A. J. Younge, D. I. Kang, K. T. Yao, M. Kang, S. P. Crago, and G. C. Fox. GPU Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 636–643, June 2014.

[27] Qemu memory API documentation. `http://git.qemu.org/?p=qemu.git;a=blob;f=docs/memory.txt`. Accessed: 2017-01-23.

[28] Ulrich Drepper. What Every Programmer Should Know About Memory. Technical report, Red Hat, Inc., November 2007.

[29] *Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part 2*. Intel Corporation, 2011.

[30] Sheng Yang. Extending KVM with new Intel® Virtualization technology. `http://www.linux-kvm.org/images/c/c7/KvmForum2008$kdf2008_11.pdf`. Accessed: 2017-02-02.