CREATING A PCI EXPRESS INTERCONNECT IN THE GEM5 SIMULATOR

BY

KRISHNA PARASURAM SRINIVASAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Associate Professor Nam Sung Kim

ABSTRACT

In this thesis, the objective was to implement a PCI (Peripheral Component Interconnect) Express interconnect in the gem5 architecture simulator. The interconnect was designed with the goal of aiding accurate modeling of PCI Express-based devices in gem5 in the future. The PCI Express interconnect that was created consisted of a root complex, PCI Express switch, as well as individual PCI Express links. Each of these created components can work independently, and can be easily integrated into the existing gem5 platforms for the ARM Instruction Set Architecture.

The created PCI Express interconnect was evaluated against a real PCI Express interconnect present on an Intel Xeon server platform. The bandwidth offered by both interconnects was compared by reading data from storage devices using the Linux utility "*dd*". The results indicate that the gem5 PCI Express interconnect can provide between 81% - 91.6% of the bandwidth of the real PCI Express interconnect. However, architectural differences between the gem5 and Intel Xeon platforms used, as well as unimplemented features of the PCI Express protocol in the gem5 PCI Express interconnect, necessitate more strenuous validation of the created PCI Express interconnect before reaching a definitive conclusion on its performance.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 PCI Express

PCI Express (Peripheral Component Interconnect Express) is a high-speed serial interconnect consisting of point-to-point links between components [1]. It is used both for peripherals integrated into the motherboard and for expansion card-based peripherals [2]. Video cards, sound cards as well as PCI Express-based solid state drives are all expansion cards that would be plugged into a particular PCI Express slot on the motherboard, depending on their bandwidth requirements [3]. On the contrary, certain Integrated Gigabit Ethernet Network Interface Cards (NIC), such as the Intel I-217, are a feature of the motherboard and cannot be removed [4].

In addition to being an onboard interconnect, PCI Express can also be used as a cluster interconnect [5]. Besides providing high bandwidth, a PCI Express interconnect fabric provides features such as non-transparency, DMA support as well as reliability. PCI Express switches (a type of PCI Express component that will be described in Section 2.2) can provide non-transparent switch ports that connect to different processor hierarchies [5]. These non-transparent switch ports provide address isolation between different PCI Express hierarchies by using address windows mapped into each hierarchy's address space, through which one hierarchy's host processor and devices can communicate with another hierarchy's processor and devices [5]. Each switch port can also contain a DMA engine, so a processor need not be involved in data movement between the fabric and a system's RAM [5]. The PCI Express protocol also enables reliable data transmission across a PCI Express link, using an acknowledgment-based protocol. This would save the protocols designed to move data across the PCI Express fabric from the burden of ensuring reliable transmission.

PCI Express devices can also support I/O virtualization (IOV) [6]. A PCI Express device residing on a single PCI Express slot can have multiple virtual “functions”. Each of these functions is treated as a separate device by the PCI Express protocol, which assigns a different function number to each virtual function. This support of functions within a device enables efficient sharing of a PCI Express-based peripheral device by different virtual machines (VM) [6]. A consequence of IOV is that devices for a particular virtual machine (VM) no longer need to be emulated by the monitor since each VM can directly access an assigned virtual function on the physical PCI express device [7]. Since each virtual function has its own memory space, isolated from that of other virtual functions on the same device, different VMs do not conflict with each other when accessing a device at the same time.

A PCI Express Generation 3 expansion card slot can offer up to 16,000 MB/s transfer speed, which is far faster than predecessor expansion slots based on the ISA (8 MB/s), PCI (133 MB/s) or AGP (2133 MB/s) buses [8]. Due to its high offered bandwidth, PCI Express-based devices include Gigabit NICs, solid state drives, GPUs, SATA controllers and many others [1]. PCI Express cabling is used for high-speed I/O devices such as flash arrays, scanning devices, etc. as well as for inter-processor communications as a substitute for Ethernet [9]. The PCI Express 4.0 standard, released in 2017, offers a bandwidth up to 32 GB/s in a single direction, while PCI Express 5.0, due for release in 2019, will offer up to 64 GB/s in a single direction [10].

1.2 `gem5` Architecture Simulator

`gem5` is an event-driven [11] computer architecture simulator that is used to model different Instruction Set Architectures (ISAs), processor and memory models, cache coherence protocols,

interconnection networks and devices [12]. gem5 can operate in two modes - Syscall Emulation (SE) mode and Full System (FS) mode. In SE mode, gem5 emulates a system call made by the benchmarking program by passing it to the host OS. In FS mode, gem5 acts as a bare metal hypervisor and runs an OS [12]. FS mode provides support for interrupts, I/O devices, etc. and provides an accurate model of a benchmarking program's interaction with the OS.

gem5 can be used to model Alpha, ARM, MIPS, x86, SPARC and Power ISAs. In FS mode, a particular kernel source code needs to be compiled for the ISA used before it can be run on gem5. In addition to a kernel, gem5 needs to be provided with a disk image to load into a non-volatile storage device. Unlike in real systems, the kernel is not loaded into memory from the provided disk image, and is provided separately to gem5 [13]. gem5 also maintains a copy on write (COW) layer to prevent modifications to a disk image when running a full system simulation [13]. Writes to the IDE disk used by gem5 are stored in buffers, but the modified data is lost when the simulation is terminated.

gem5 provides simple CPU, in order CPU and out of order CPU models respectively. In the simple CPU model, instructions are ideally fetched, decoded, executed and committed in the same cycle [12]. The simple CPU model can be further divided into atomic simple CPU, where memory accesses return immediately and timing simple CPU, where memory access times are modeled, resulting in a delay for Instruction Fetch as well as for memory reads and writes [14]. In order and out of order CPUs are pipelined CPUs where instructions are executed in the "execute stage" of the pipeline instead of at the beginning or end of the pipeline [15].

Currently implemented device models in gem5 include NICs, IDE controller, UARTs, a frame buffer and DMA engines [12]. Both the NIC models and IDE controller are PCI-based

devices in gem5 at the logical level, although there is no physical PCI bus implemented in gem5.

gem5 is a C++ based simulator. All components including CPUs, peripheral devices, caches, buses and bridges are implemented as C++ classes. Inheritance is heavily utilized, with components that share similar characteristics inheriting from the same base class. For example, PCI-based devices inherit from the PciDevice class and since PCI-based devices are capable of DMA, the PciDevice class inherits from the DmaDevice class. In addition to inheritance, gem5 classes utilize polymorphism to account for the different behavior of different components that inherit from a base component. For example, *ports* used to connect objects together define standard functions to send or receive data. A base class for *ports* declares these standard functions as virtual or pure virtual, and all port classes that derive from this base class implement these standard functions based on their requirements. Ports used by a device and ports used by a bus have vastly different implementations of these standard functions.

In gem5, time is measured in *ticks*. A tick is defined as one picosecond. Since hardware components have delays in the order of nano or micro seconds, it is advantageous to be able to model timing at such a fine granularity. Events are scheduled for and executed at a certain tick value. Usually the *curTick()* function is used to get the current tick value, and an offset in ticks is used to indicate how long from the present an event should be scheduled for. The current tick is stored in an unsigned 64-bit variable, allowing a maximum simulation length of around 214 days.

1.3 Thesis Objective and Outline

The goal of this thesis is to create a PCI Express interconnect in gem5. This process involves a verification that PCI Express-based devices can indeed be modeled in gem5 in the future,

followed by the creation of the actual PCI Express interconnect. Currently, there are no PCI Express-based devices present in gem5, so evaluation of the interconnect is performed using the existing PCI bus-based IDE disk.

The organization of the thesis is as follows. First the basics of the PCI bus and PCI Express shall be described, providing a general overview and comparison of both interconnects. Then the gem5 platform used to implement PCI Express shall be described, along with the various devices present and the buses used. This shall be followed by a detailed description of the work performed, including the new features added to the simulator. Next, validation of the created PCI Express interconnect in gem5 using a PCI Express interconnect from a real system shall be described, along with experiments measuring the performance of the created PCI Express interconnect with varying parameters. Lastly, the results obtained shall be described, along with ideas on improvements to the PCI Express model.

CHAPTER 2: BACKGROUND

In this chapter, basic concepts regarding the PCI bus, PCI Express and the base `gem5` platform that was used will be explained.

2.1 PCI Bus

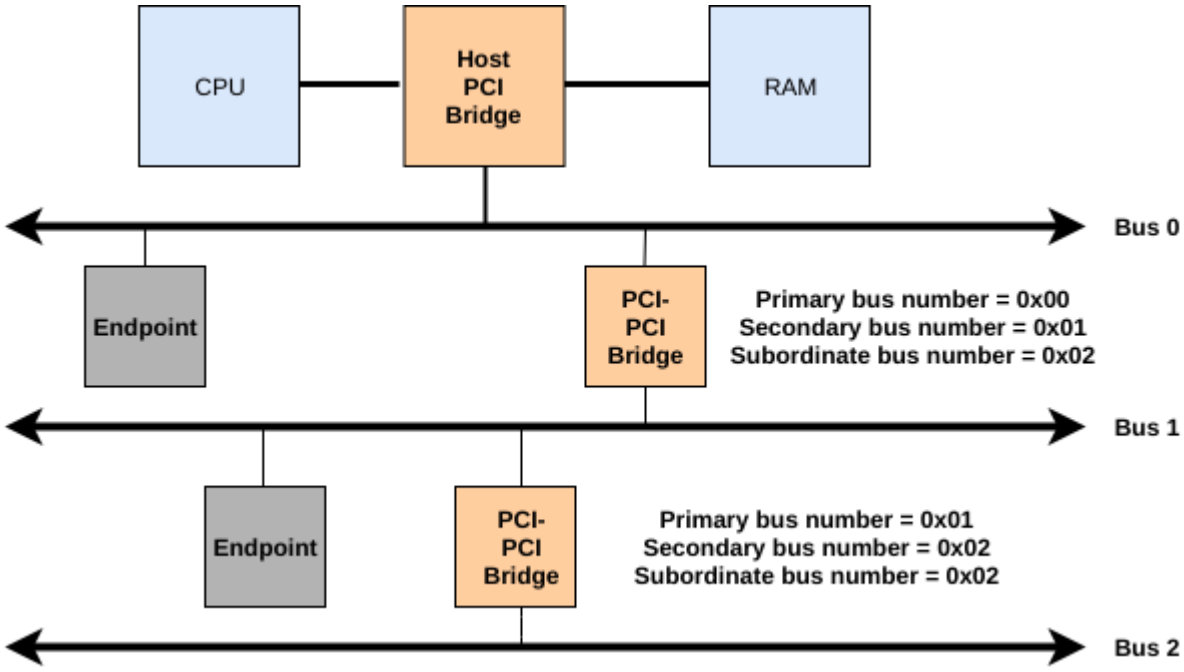

Fig. 2.1 A PCI bus hierarchy consisting of endpoints and bridges [16]

PCI (peripheral component interconnect) is a shared bus, consisting of several devices connected to a single bus, clocked at 33 or 66 MHz [17]. Each PCI bus-based device can either be an endpoint or bridge. Endpoints are devices such as NICs, IDE controller, etc. which are connected to only a single PCI bus, as shown in Fig. 2.1. Bridges, on the other hand, are used for connecting two PCI buses together. They route transactions from one bus interface to the other depending on the address of the transaction.

The Host-PCI bridge converts CPU requests into PCI bus transactions. It also converts requests from the PCI bus hierarchy to memory requests. The bus immediately downstream of the Host-PCI bridge is PCI Bus 0.

Both PCI endpoints and bridges expose three address spaces to device drivers and enumeration software (described shortly). *Configuration space* is an address space used by PCI devices to map their configuration registers into [18]. Configuration space is 256B in size for PCI devices, and the first 64B consist of standard registers and is termed a *header*. PCI endpoints expose a type 0 header, while PCI bridges expose a type 1 header. The remaining 192B of configuration space is used for implementing device-specific registers in both endpoints and bridges. In addition to configuration space, PCI devices expose two other address spaces - *memory space* and *I/O space* respectively. Both memory and I/O spaces are used by device drivers to communicate with a particular device. Memory space refers to address locations that can be accessed through using memory mapped I/O (MMIO) using regular memory load and store instructions, while I/O space refers to address locations that can be accessed through port-mapped I/O using special instructions, such as the *in* and *out* instructions in x86 [19]. Device drivers need to obtain the base address of a device's memory or I/O space and read or write registers present at particular offsets within that region.

During system startup, the devices present in the PCI hierarchy are not known to the kernel. The PCI devices present must be dynamically detected every time during startup. This is performed in a process called *bus enumeration*, by software present in the kernel or BIOS [20]. Enumeration includes numbering a particular PCI bus, followed by attempting to read the Device and Vendor ID configuration registers of devices on all slots on the bus. If a device is indeed present at a particular slot on the bus, the Vendor and Device ID used to identify the device are a

valid value, or else the bridge attached to the bus returns 0xFFFF for both registers to signal to the software that a particular slot is empty.

Enumeration is performed in a depth-first manner. While scanning the devices present on a particular bus, if the enumeration software detects a PCI bridge, the bus immediately downstream of the bridge is enumerated. In addition to assigning bus numbers and detecting devices, enumeration software also grants I/O space and memory space resources to devices so that they can be managed by their device driver later.

At the end of the enumeration process, each PCI bridge is configured with a primary, secondary and subordinate bus number as shown in Fig. 2.1. The primary bus number is the number of the bus upstream of the PCI bridge, the secondary bus number is the number of the bus present immediately downstream of the bridge, while the subordinate bus number is the highest numbered bus downstream of the bridge respectively. Here, upstream refers to in the direction of the Host-PCI bridge, while downstream refers to the opposite direction. Each PCI bridge is also configured with “windows” containing the range of memory and I/O space addresses of devices located downstream of the bridge. This window is used by the bridge to route PCI memory and I/O transactions from its primary bus to secondary bus.

Both endpoints and bridges on the PCI bus are capable of acting as a PCI bus master. As a consequence, DMA transactions and peer - peer transactions by PCI devices are possible [17]. A device that wants to be a PCI bus master asserts a request signal to an arbiter, which proceeds to eventually grant that device ownership of the PCI bus. Once a PCI bus master receives this grant and if the PCI bus is idle, it can start a transaction over the bus [17].

PCI buses do not support split transactions. If a device acting as a slave for a particular transaction is not able to immediately service the request provided by the bus master, it can insert

wait states while trying to service the request. If the request cannot be serviced within a certain number of cycles, a slave must signal a disconnect, which forces the master to give up ownership of the PCI bus and try again later [17]. This allows the PCI bus to be used by other waiting bus masters.

Lastly, each PCI device can signal an interrupt using one of four interrupt signals, termed as INTA, INTB, INTC and INTD [17]. Multiple PCI devices can share a single interrupt signal, and consequently there is an OS overhead to check which device actually signaled an interrupt among all devices using a particular interrupt signal.

2.2 PCI Express

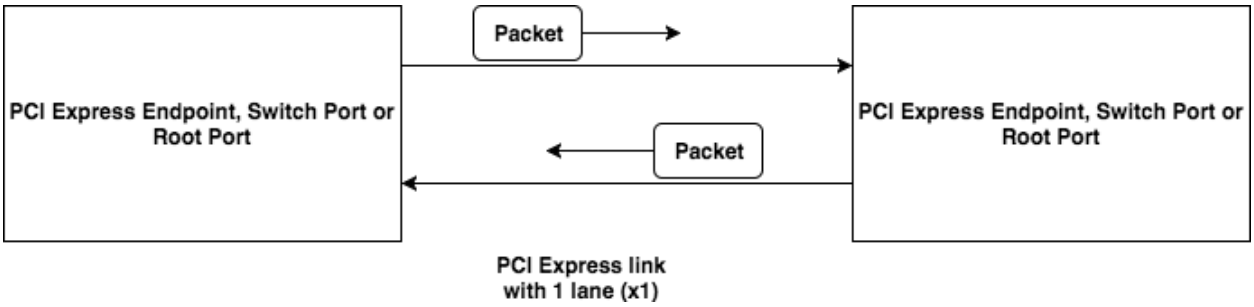

Fig. 2.2 A representation of a PCI Express link with a single lane between a pair of devices

A PCI Express interconnect consists of point-point links between devices. As shown in Fig. 2.2, a PCI Express link can be subdivided into two unidirectional links, with each unidirectional link used to transmit data in a single direction. Each unidirectional link consists of a pair of wires that use differential signaling to indicate a 0 or a 1 transmitted on the unidirectional link [17]. (The sign of the voltage difference between the two wires making up a unidirectional link determines whether a 0 or 1 is transmitted.) Each PCI Express link can be of 1, 2, 4, 8, 12, 16 or 32 *lanes*,

with each lane consisting of a pair of unidirectional links. The number of lanes in a PCI Express link is referred to as link width, and is denoted by “x1”, “x2”, “x4”, etc. Bytes can be transmitted in parallel across all lanes forming a PCI Express link, and thus link bandwidth increases proportionally with the number of lanes in the link.

A Gen 1 PCI Express link can transmit data at a rate of 2.5 Gbps per lane in each direction, a Gen 2 PCI Express link at 5 Gbps per lane in each direction and a Gen 3 PCI Express link at 8 Gbps per lane in each direction [1]. These transfer rates do not take encoding overhead into account, which may be as much as 20%, reducing the effective bandwidth of the link.

Similar to PCI devices, PCI Express devices expose three address spaces to enumeration software and device drivers [17]. These are configuration space for configuration by the device driver and enumeration software, memory space for memory mapped I/O by device drivers and I/O space for port mapped I/O by device drivers. PCI Express devices are assigned regions in memory and I/O space depending on their requirements. All routing components in the PCI Express hierarchy are informed of these regions.

PCI Express is a packet-based protocol, where all data exchanged between devices is encapsulated in packets. Using packets allows usage of a serial link, and getting rid of several bus signals, since all the information required to perform a transaction is stored in the packet itself, along with the data. There are several types of packets used in PCI Express. The packets that actually carry data are called *transaction layer packets* or TLPs. Each TLP can either be a request, response or message. A TLP consists of a header indicating the type of packet, target address, payload length, etc. and optionally a payload. Request TLPs are issued by a *requestor device* and consumed by a *completer device*, which services the request. For some request TLPs, responses are required, and a response TLP is generated by the completer once it services the

request and is consumed by the requestor that issued the request. Since packets are used, and the response TLP is decoupled from the request TLP, a particular link is idle while the completer is processing the request. Thus, unlike PCI, PCI Express supports split transactions.

A TLP can be routed based on either an address or an “ID”. In address-based routing, the address field in the packet’s header is used by routing components in PCI Express to decide which link to send the packet on [17]. In ID-based routing, a combination of the target device’s bus, device and function number is used to decide the next link to send the packet on.

Some request TLPs do not require a response TLP. These are called *posted requests*. Posted requests are used mainly for messages and for writes to memory space [17]. Messages are mainly used to implement Message Signaled Interrupts (MSI), where a programmed value is written to a specified address location to raise an interrupt [17].

In addition to TLPs, there is a class of packets called data link layer packets (DLLPs). These are local to a specific link and are issued by a device on one end of the link and consumed by the device on the other end. Another link-specific class of packets called physical layer packets (PLPs) are used for link training and initialization.

As seen in Fig. 2.3, a *root complex* connects the PCI Express hierarchy to the CPU and DRAM. A root complex converts requests from the CPU into PCI Express transactions, and converts PCI Express transactions from endpoints to memory requests, thereby enabling DMA to take place. The root complex thus acts as both a requestor and completor. The root complex consists of multiple root ports, and PCI Express links are connected to each root port. The device connected to the root port can be either an endpoint or a PCI Express switch, as shown in Fig. 2.3. All the devices downstream of a particular root port form a hierarchy domain [17].

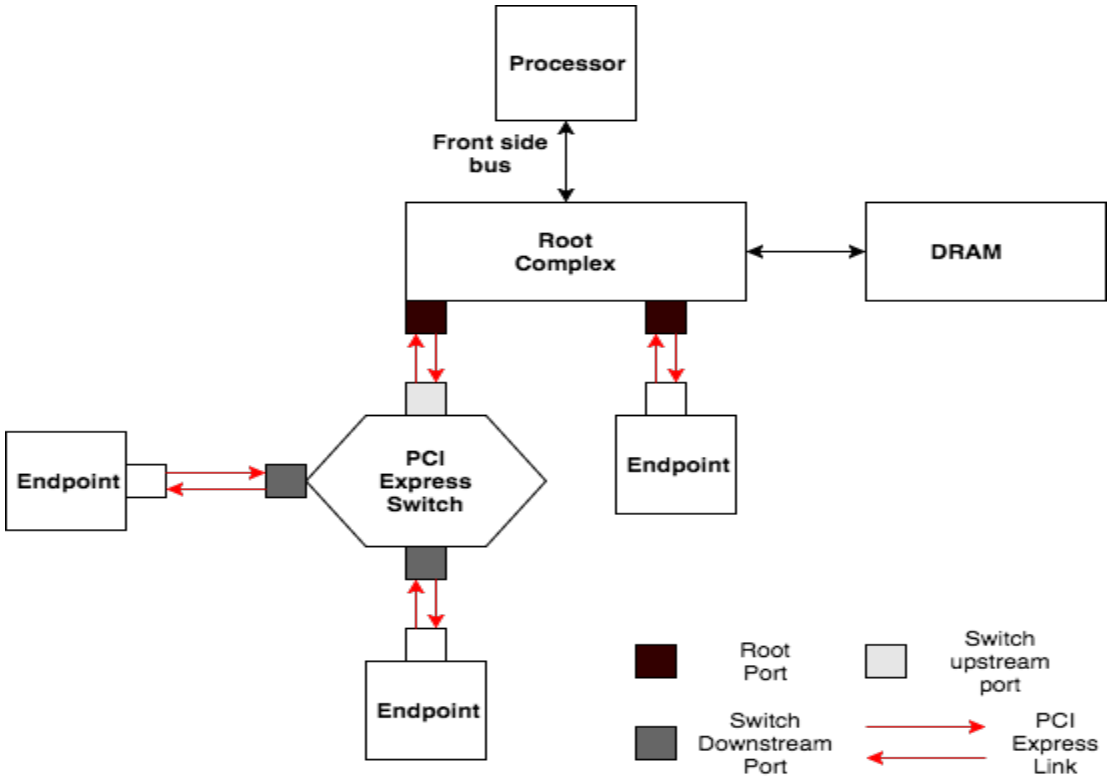

Fig. 2.3 A PCI Express hierarchy consisting of a root complex, switch and endpoints

A *PCI Express switch*, as shown in Fig. 2.3, is used for connecting multiple devices to the PCI Express hierarchy. Since point-point links are used in PCI Express, without switches, the maximum number of endpoints is the same as the number of root ports. A PCI Express switch consists of only one upstream port and one or more downstream ports. The primary function of the switch is to perform forwarding of TLPs received on one port to an appropriate output port using either ID-based or address-based routing. A switch is characterized by its latency, which is defined as the time between the first bit of a packet arriving at the switch on the ingress port and the first bit of the packet leaving the egress port [21].

2.3 gem5 Platform Description

The PCI Express interconnect was implemented on the ARM Vexpress_Gem5_V1 platform

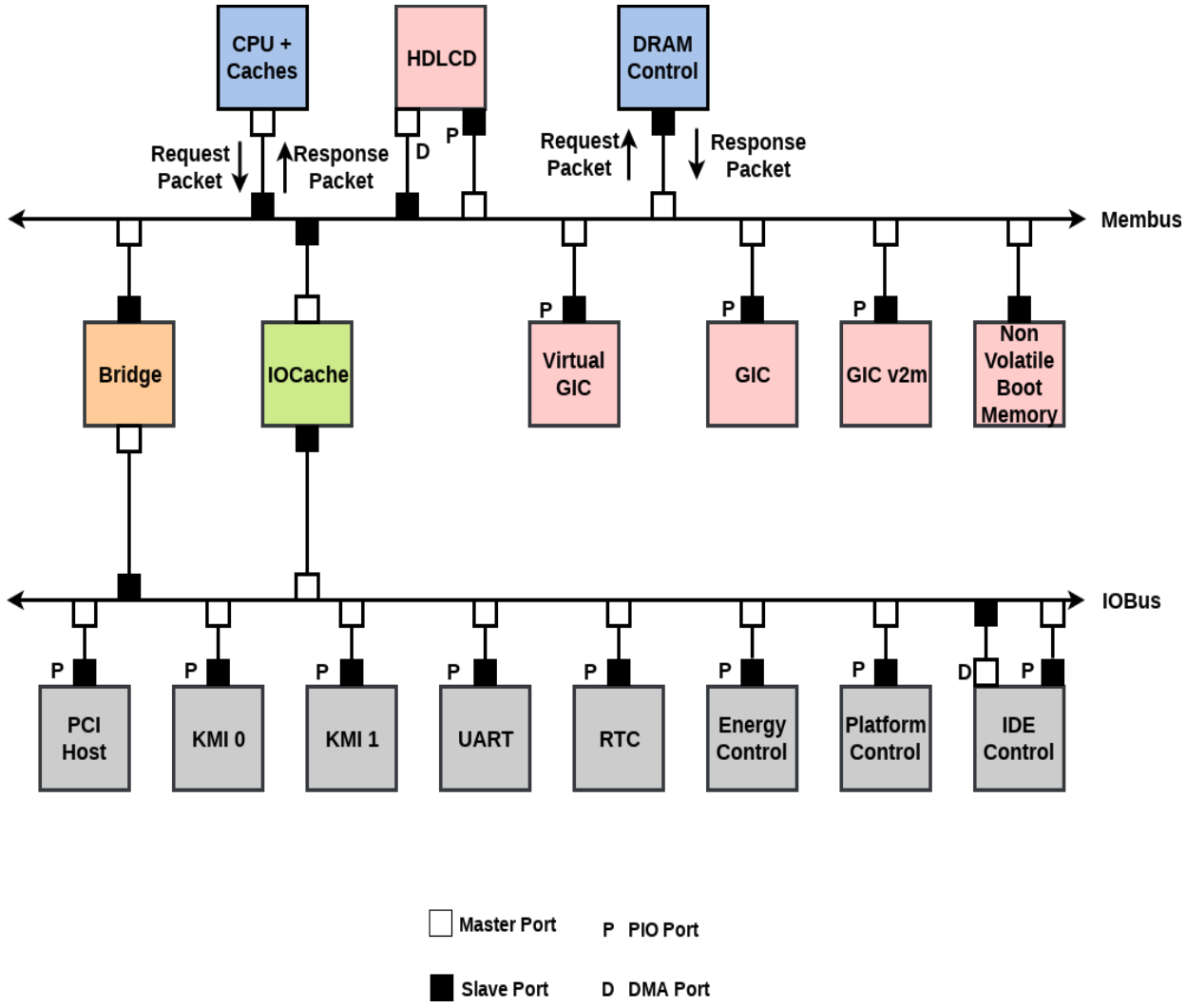

Fig. 2.4 Overview of the devices present in the base Vexpress_Gem5_V1 platform

in gem5, an overview of which is depicted in Fig. 2.4. In this platform model, on-chip devices include a Generic Interrupt Controller (GIC) and an LCD controller. Off-chip devices include a PCI Host, Real Time Clock, UART, keyboard and mouse interface (KMI), as well as a PCI-based IDE controller implemented in gem5. On-chip and off-chip devices are placed in distinct memory ranges. Since gem5 does not differentiate between requests to Configuration, I/O and Memory space, different Configuration, I/O and Memory space ranges must be assigned to PCI devices in gem5. In the Vexpress_Gem5_V1 platform, 256MB (0x30000000 - 0x3fffffff) is

assigned for PCI configuration space, 16MB (0x2f000000 - 0x2fffffff) for PCI I/O space and 1GB (0x40000000 - 0x80000000) for PCI Memory Space respectively, as shown in the system memory map in Fig. 2.5. DRAM is mapped to addresses from 2 GB - 512 GB. A consequence of this memory map is that all PCI devices can use 32-bit address registers instead of 64-bit ones.

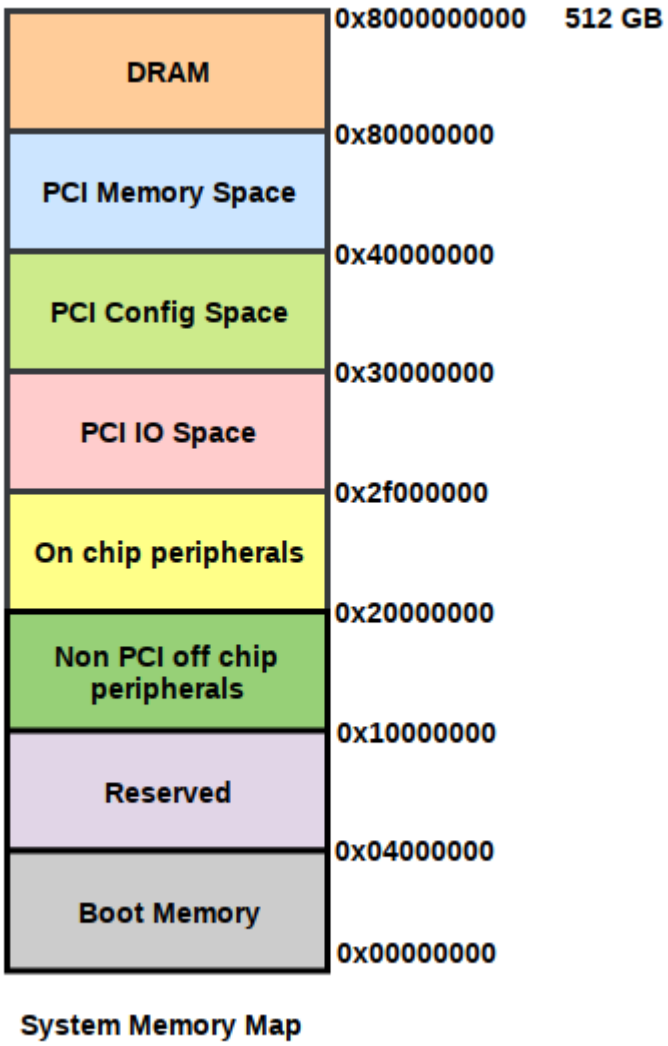

Fig. 2.5 System memory map of Vexpress_Gem5_V1 platform

On-chip devices, caches and the DRAM controller reside on a coherent crossbar implementation in `gem5`, which we call the *MemBus*. Off-chip devices reside on a non- coherent

crossbar, which is called *IOBus*, as shown in Fig. 2.4. Crossbar implementations in gem5 have *master* and *slave* ports, which are used for sending requests to and accepting requests from connected slave and master devices respectively. These crossbars are loosely modeled on the ARM AXI interconnect, where transactions are routed to different slave devices based on the address range each slave device registers with the crossbar [22]. The crossbars have latencies associated with making forwarding decisions, as well as for moving data across the crossbar. Often crossbar latencies have to be accounted for by devices attached to the crossbar such as bridges, which add these latencies to their own internal delays.

All gem5 memory transactions are represented in the form of "packets", which are transported through the system depending on the target address. Both read and write request packets need a response packet by design. This proves to be a hindrance to our gem5 PCI Express implementation, which needs posted write requests that do not require a response to model real PCI Express interconnects more accurately. However, the packet-based nature of the gem5 memory and I/O systems make modeling PCI Express packets easier. In the gem5 memory and I/O system, every component needs a standard interface to send and receive packets, and this is implemented in the form of ports, subdivided into slave and master ports, as shown in Fig. 2.4. A slave port needs to be paired with a master port and vice versa. Slave ports are used by gem5 components to accept a request packet or send a response packet, while master ports are used to accept a response packet or send a request packet. PCI-based peripheral devices in gem5 implement two ports - A *PIO slave port* to accept request packets originating from the CPU, and a *DMA master port* to send DMA request packets to memory through the interconnect. In the base implementation of gem5, a PIO port of a PCI device is connected to the master port of the IOBus while the DMA port is connected to the slave port of the IOBus.

gem5's PCI host is used to functionally model a Host-PCI bridge. The PCI host claims the whole PCI configuration space address range, so that any request from the CPU to PCI configuration space reaches the PCI host. All PCI-based peripherals in gem5 need to register themselves with the PCI host, which provides an interface to do so. Each device registers itself using a geographical address consisting of its bus, device and function numbers. The PCI host maps 256 MB of configuration space to itself with a base address of 0x30000000 using the Enhanced Configuration Access mechanism, where up to 4096 B of configuration registers can be accessed per a device's function. On receiving a configuration request from the CPU, the PCI Host parses the bus, device and function numbers from the request's destination address and passes the request packet to a corresponding device that has registered with it. If there is no registered device matching the configuration request's target, the PCI host simply fills in all 1's in the data field of the request packet and sends back a response to the CPU.

A *bridge* is used to connect the *MemBus* to the *IOBus* and is a slave device on the MemBus and a master device on the IOBus. The bridge is configured to accept packets from the MemBus that are destined for an address in the off-chip address range. In my gem5 PCI Express implementation, this gem5 bridge model is built upon to create a root complex and a switch. In addition to the bridge, gem5 employs an IOCache, a 1 KB cache which is a master on the MemBus and a slave on the IOBus. The IOCache is used to service DMA requests from PCI devices that reside on the IOBus.

There are only two categories of PCI-based peripheral devices in gem5 - an IDE Controller and several Network Interface Cards. While the base version of the Vexpress_GEM5_V1 platform does not feature an NIC, a couple of NIC models are integrated later on, as described in section 3.1.

CHAPTER 3: IMPLEMENTATION

The PCI Express model in gem5 involved three main steps.

(1) Verifying that future gem5 models for PCI Express-based devices would be feasible regardless of the interconnect used in gem5.

(2) Creating a Root Complex and a PCI Express switch.

(3) Creating individual PCI Express links and implementing the Ack/NAK protocol for reliable transmission.

The modifications made to gem5 in each step shall be described, along with an explanation of each of these features in a real PCI Express system.

3.1 Verifying Feasibility of Future Models for PCI Express-Based Devices in gem5

In this step, the objective was to ensure that PCI Express devices could indeed be modeled in gem5, even if gem5 did not employ a PCI Express interconnect. Since there are no PCI Express-based devices implemented in the base version of gem5, a driver for a PCI Express-based device was chosen, and used to configure an existing PCI device in gem5. If a driver for a PCI Express-based device was able to successfully configure a gem5 device regardless of the underlying gem5 interconnect, this would prove that future PCI Express device models can be implemented in gem5. The work done in this section was similar to what Gouk et al. [23] did in their implementation of a PCI Express-based NVMe device in gem5, where PCI Express configuration space for the NVMe device had to be exposed to the NVMe driver.

PCI/PCI Express configuration space

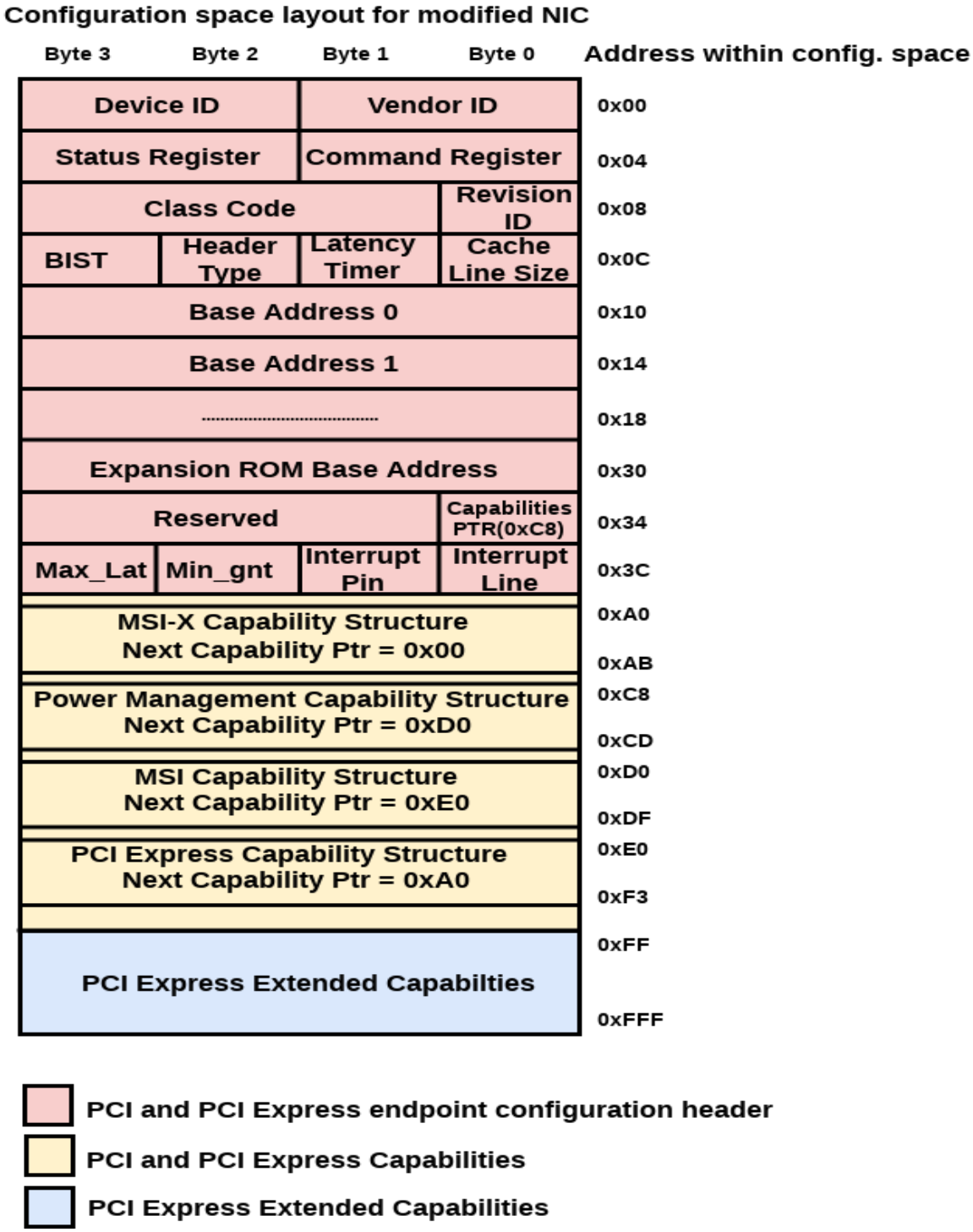

Fig. 3.1 Configuration space layout for *pcie-nic*

Both PCI and PCI Express devices map registers into configuration space, so that they can be detected by enumeration software [24] in the kernel, and their device drivers identified. PCI devices have a configuration space of 256B per function, while PCI Express devices have a configuration space of 4KB per function. For the remainder of this thesis, the assumption that a PCI/PCI Express device consists of only a single function shall be made, and *device* and *function* shall be used interchangeably. Both PCI and PCI Express devices expose a standard 64B *header* in the first 64B of configuration space to kernel enumeration software, as shown in Fig. 3.1. PCI/PCI Express endpoints expose a type 0 configuration header, while PCI-PCI bridges expose a type 1 configuration header respectively. Both PCI and PCI Express devices can implement *capability structures* between addresses 0x40 and 0xFF in their configuration space, as shown in Fig. 3.1.

PCI Express devices must implement a PCI Express capability structure in the first 256B of configuration space, described in the next subsection. Unlike PCI devices, a PCI Express device can also implement *extended capability structures* after address 0x100 in its configuration space. These extended capabilities can be for advanced error reporting, virtual channels, etc. and are configured by a device's driver [17]. Each capability structure consists of a set of registers [17], and is identified by a capability id. The capability structures are organized in a chain, where one capability structure has a field containing the address of the next capability structure within a device's configuration space, as illustrated in Fig. 3.1. The *capability pointer* register in the 64B device header contains the address of the first capability structure within configuration space.

`gem5` has support for the first 64B (header) of a PCI/PCI Express endpoint's configuration space. In addition to the header, `gem5` also provides data structures for the

capability structures used for Power Management, MSI interrupts and MSI-X interrupts. Most importantly, gem5 includes support for the PCI Express capability structure, shown in Fig. 3.2.

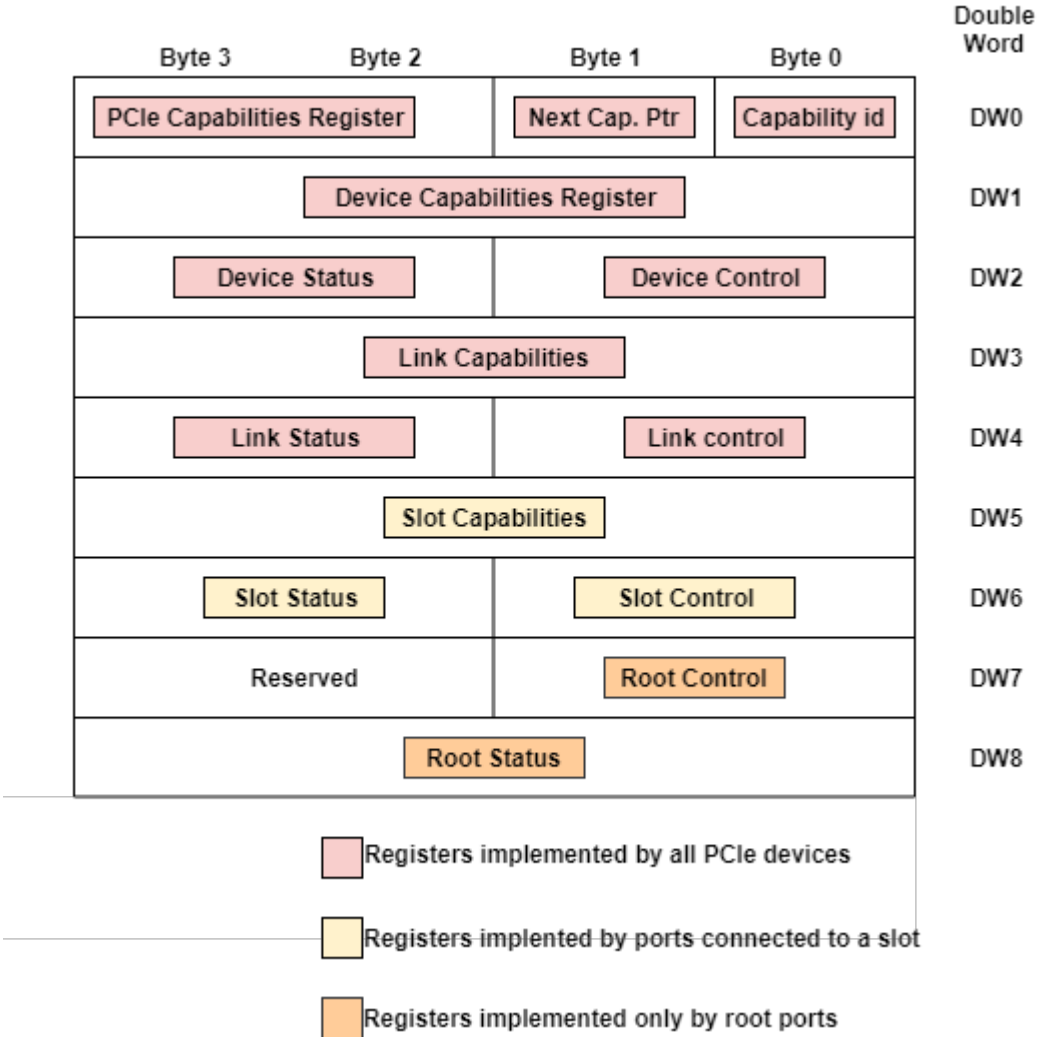

Fig. 3.2 PCI Express capability structure

Configuring a gem5 PCI Device Using a Driver for a PCI Express-Based Device

For the base device in gem5, the PCI-based Intel *8254x* NIC model was chosen. The goal was to get the *e1000e* driver to configure a modified version of the *8254x* NIC. The *e1000e* driver is designed for the PCI Express-based Intel *82574l* NIC, among other devices. The *8254x* NIC was

modified to more closely resemble the *82574l* NIC, and was successfully configured by the *e1000e* driver, even though there was no PCI Express interconnect implemented in `gem5` at this point. We shall refer to this modified version of the *8254x* NIC as *pcie-nic* for the remainder of this section.

The first step in creating *pcie-nic* was to change the device ID register in the *8254x*'s configuration header, shown in Fig. 3.1. PCI device drivers expose a Module Device Table to the kernel [25], which lists the vendor and device IDs of all devices supported by that driver. The driver's *probe* function is invoked by the kernel if a device that the driver supports is discovered during enumeration. *pcie-nic* was assigned a device ID of 0x10D3, which corresponds to the *82574l*'s device ID, to invoke the *e1000e* driver's probe function.

According to its datasheet [26], the Intel *82574l* NIC implements Power management (PM), Message Signaled Interrupts (MSI) and MSI-X capability structures in addition to the PCI Express capability structure. First, the capability pointer register in *pcie-nic*'s 64B header is assigned the base address of the PM capability structure in *pcie-nic*'s configuration space, as shown in Fig. 3.1. The PM structure "points" to the MSI capability structure, which in turn is followed by the PCI Express capability structure and MSI-X capability structure respectively. Since `gem5` does not have support for PM, MSI and MSI-X, these functionalities are disabled by appropriately setting register values in each structure.

The PCI Express capability structure, shown in Fig. 3.2, is configured to trick the *e1000e* device driver into believing that *pcie-nic* resides on a PCI Express link, even though it resides on the `gem5` IOBus in reality. The physical layer is hidden from the device driver, and the *e1000e* driver believes that *pcie-nic* resides on a Gen1 x1 PCI Express link. The PCI Express capability

structure is configured based on the *82574l*'s datasheet and indicates the following properties of the *pcie-nic* to the *e1000e* driver, shown in Table 3.1.

Table 3.1 PCI Express related properties of *pcie-nic* as configured

Property	Value
PCI Express Link Width	x1
PCI Express Maximum Payload Size	256 B
PCI Express Link Speed	2.5 Gbps (Gen 1)

In the base version of `gem5`, access to a device's configuration space in the range of 0x40-0xFF would generate a warning that device-specific configuration space was not implemented. Only the registers belonging to the 64B header could be accessed by device drivers/enumeration software. To provide access to the capability structures created in the configuration space of *pcie-nic*, a check is made to see whether a configuration access falls within a particular capability structure's address range, and the corresponding structure's registers are read/written if it does. The starting (base) address of each capability structure within configuration space is a parameter when creating a PCI device in `gem5`, and in *pcie-nic*, the base addresses of the capability structures are set to the base addresses mentioned in the *82574l*'s datasheet. By knowing the size of each capability structure, which is constant for a given device, configuration accesses could be mapped to a particular capability structure's registers.

Figure 3.3 shows the boot log output when the *e1000e* driver successfully configures *pcie-nic.* As mentioned, the *e1000e* driver identifies *pcie-nic* as the *82574l* NIC, and is tricked into believing that *pcie-nic* is located on a Gen 1 x1 PCI Express link.

[0.982399] e1000e 0000:03:00.0 eth0: registered PHC clock

[0.982446] e1000e 0000:03:00.0 eth0: (PCI Express:2.5GT/s:Width x1) 00:a0:c9:01:01:05

[0.982506] e1000e 0000:03:00.0 eth0: Intel(R) PRO/1000 Network Connection

Fig. 3.3 Boot log showing *e1000e driver's output*

3.2 Root Complex

Once we verified that future PCI Express-based devices could be used in `gem5` without concern about the physical interconnect, the next step was to try and create a root complex that would allow PCI Express links to be used to connect devices in a PCI Express hierarchy. In the baseline `gem5` implementation, there was no PCI-PCI bridge, which meant that only one IOBus could be used. Although `gem5` has a generic bridge model, the address ranges passed by the bridge from the primary to secondary interface are not programmable by the PCI/PCI Express enumeration software in the kernel.

A root complex in a real system consists of a Host-PCI bridge and virtual PCI-PCI bridges [27] for each root port as shown in Fig. 3.4. The bus internal to the root complex is enumerated as bus 0 by software, and each virtual PCI-PCI bridge of the root complex is enumerated as a device on bus 0. The Host-PCI bridge and bus 0 are already known to enumeration software before enumeration begins [17].

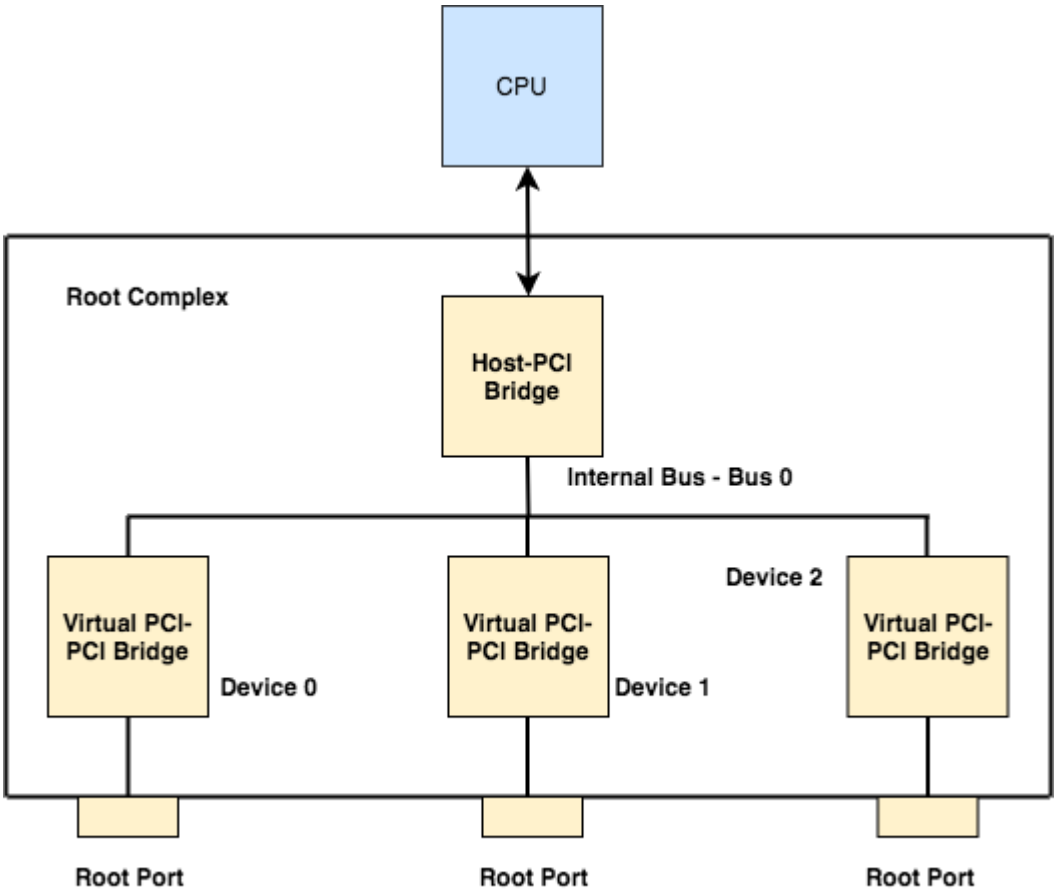

Fig. 3.4 Internal structure of root complex in a real system

Root Complex Implementation in gem5

In the gem5 implementation, the root complex accepts CPU requests directly from the MemBus, as shown in Fig. 3.5. DMA requests from PCI Express devices are passed onto an IOCache that sits between the root complex and the MemBus. A Host-PCI bridge is not included within the gem5 root complex, and gem5's version of the Host-PCI bridge is used instead. The root complex implements three root ports, and a *Virtual PCI-PCI bridge* (VP2P) is associated with each root port as shown in Fig. 3.5. Each VP2P can be assigned a bus and device number. Each root port consists of a gem5 master and slave port respectively.

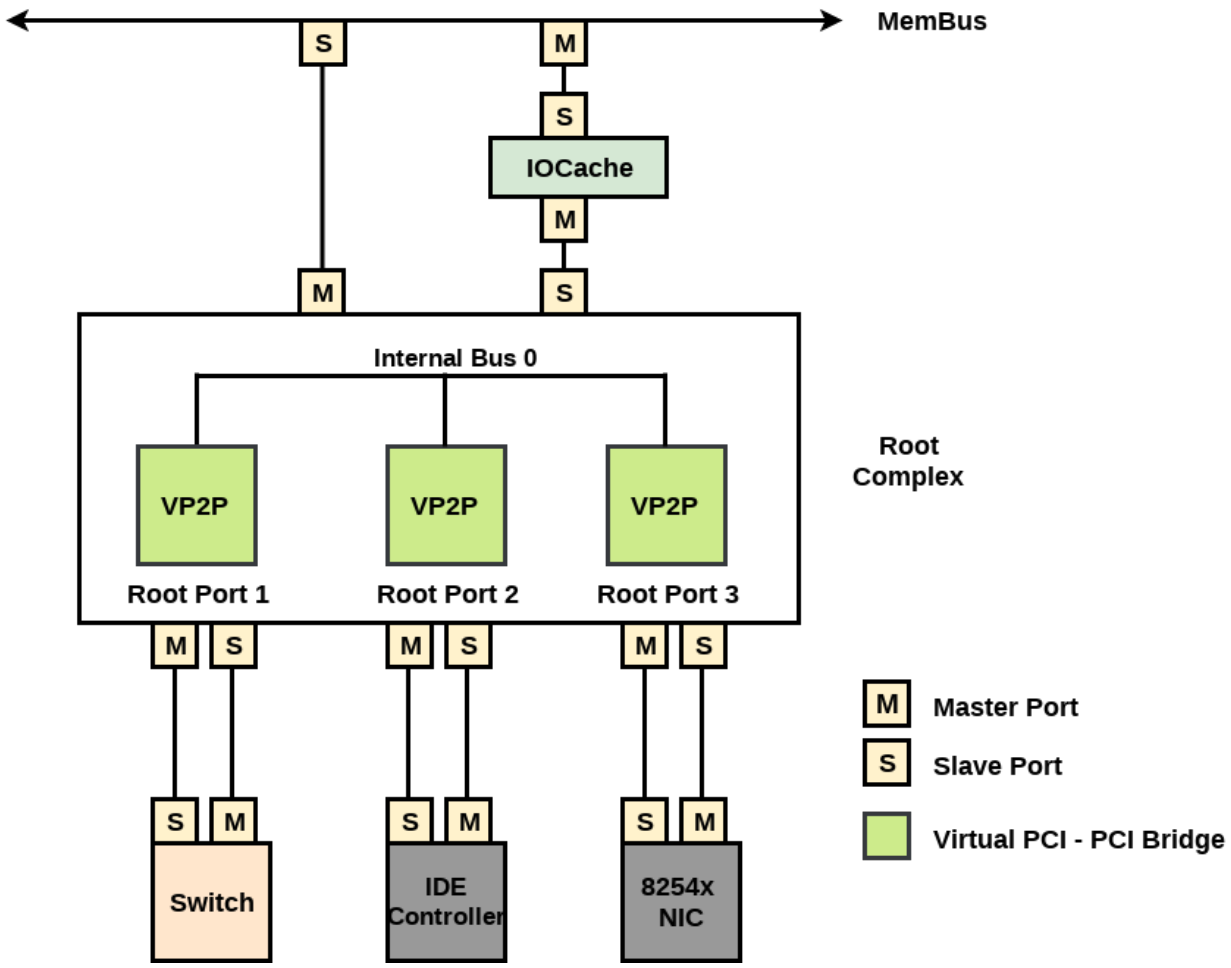

Fig. 3.5 gem5 implementation of root complex

In addition to the slave and master ports associated with each root port, the root complex also has an upstream (toward CPU and memory) master port and slave port. The upstream slave port is used to accept requests from the CPU that are destined for PCI Express devices, while the upstream master port is used to send DMA requests from PCI Express devices to the IOCache. The three downstream slave ports are used to accept DMA requests from devices connected to each root port, while the three downstream master ports are used to send CPU requests to devices connected to each root port. As mentioned earlier, all memory system requests/responses in gem5 are in the form of packets, so there is no need to create separate PCI Express TLPs in the root complex. It is worth noting that unlike in real systems, configuration accesses do not flow

through the implemented root complex in gem5. Configuration packets are destined for the PCI Host, which directly transfers the packet to the respective device depending on the packet's address.

As mentioned earlier, the root complex is built upon the pre-existing bridge model in gem5. Consequently, each port associated with the root complex has buffers to store outgoing request packets in the case of the master ports or outgoing response packets in the case of the slave ports. The response or request processing latency of the root complex is configurable, and decides the period after which packets will be transmitted from the buffers out to the interconnect. The root complex is designed to work either with the pre-existing gem5 crossbars or with the PCI Express links implemented later.

Creating a Virtual PCI-PCI Bridge

A virtual PCI-PCI bridge (VP2P) for a particular root port implements a PCI-PCI bridge's configuration header (shown in Fig. 3.6) and exposes it to the enumeration software in the kernel. The necessary fields in the configuration header are configured by the kernel similar to how they would be configured for an actual PCI-PCI bridge that is connected to PCI buses, and consequently each VP2P comes to be associated with distinct portions of memory and I/O space. Since all configuration accesses go through the PCI Host in gem5, the VP2P is registered with the PCI Host when the simulation starts. For this reason, the VP2P needs to be manually configured with a bus, device and function number when launching the gem5 simulation. The bus number is always 0 for the VP2P associated with a root port, but this is not the case for a VP2P associated with a switch port. The VP2P also implements a PCI Express capability structure within configuration space as shown in Fig. 2.7. The configuration header of a PCI-PCI

bridge is a type 1 header, as opposed to the type 0 header found in PCI Express endpoints. The following list describes a few important registers of the type 1 header, along with their purpose in a PCI-PCI bridge and the values used to configure them.

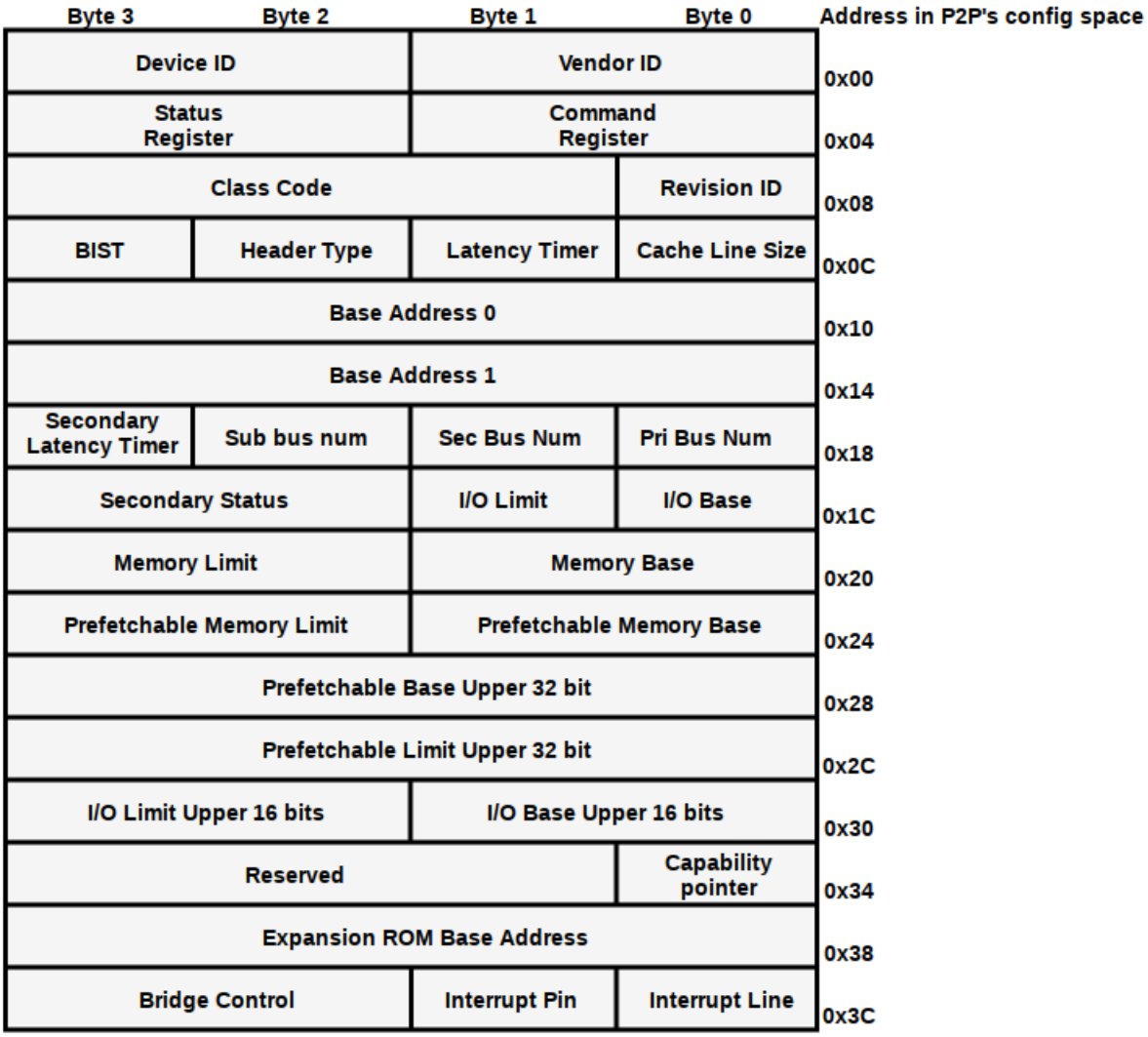

Fig. 3.6 A PCI-PCI bridge's type 1 configuration header

- *Vendor ID*, *device ID* - These two registers are used to identify the PCI-PCI bridge. We set the *vendor ID* to be 0x8086 in all three VP2Ps, corresponding to Intel, and the *device ID*s to be 0x9c90, 0x9c92 and 0x9c94 respectively, corresponding to Intel Wildcat chipset root ports [28].

- *Status register* – This register indicates interrupt and error status of the PCI-PCI bridge among other features. We set all bits to 0, except bit 4 which is set to 1 indicating that the c*apability pointer register* in the header contains a valid value.

- *Command register* – This register is used to configure settings related to the PCI-PCI bridge's primary interface. We set the corresponding bits in this register to indicate that I/O or Memory space transactions that are received on the bridge's secondary interface are forwarded to its primary interface. We also set a bit to indicate that devices downstream of the PCI-PCI bridge can act as a bus master. This enables DMA.

- *Header type* - We set this register to 1 to indicate that this configuration header is for a PCI-PCI Bridge instead of for an endpoint.

- *Base address registers* - We set these registers to 0 to indicate that the PCI-PCI bridge does not implement memory-mapped registers of its own and requires no memory or I/O space.

- *Primary, secondary, subordinate bus number registers* - These registers indicate the immediate upstream, immediate downstream and largest downstream bus numbers with respect to the PCI-PCI bridge. These are configured by software and we initialize them to 0s.

- *Memory base and limit registers* - These registers define a window that encompasses the memory space (MMIO) locations that are downstream of a PCI-PCI bridge. Both the *base* and *limit* registers are 16 bits. The base and limit addresses of a MMIO window are both aligned on

1MB boundaries, so only the top 12 bits of the 32-bit addresses need to be stored in these registers. The bottom 20 bits are taken to be all 0s in the calculation of the base address and all 1s in the case of the limit address respectively. The *Memory Base* and *Limit* registers are configured by enumeration software and we initialize them to 0.

- *I/O base and limit registers* - Similar to the *memory base* and *limit* registers, these registers define the I/O space locations downstream of a PCI-PCI bridge. The I/O Base and Limit addresses are 32 bits in `gem5`, since the PCI I/O space range starts from 0x2f000000 in the Vexpress_Gem5_V1 platform. Thus, the *I/O Base Upper* and *I/O Limit Upper* registers need to be used to implement this I/O space window. However, the kernel used assigns I/O addresses only up to 16 bits, and to circumvent this, we hardcode 0x2f00 in the *I/O Base Upper* and *I/O Limit Upper* registers. I/O base and limit addresses are aligned on a 4KB boundary, and the lower 12 bits of the I/O base address is assumed to be all 0's and the lower 12 bits of the I/O limit address is taken to be all 1's.

- *Capability pointer* – This is used to point to capability structures implemented in the PCI-PCI bridge's configuration space. We set this to 0xD8 to indicate the starting address of the PCI Express capability structure in configuration space.

It is worth noting that the values assigned by enumeration software to the *status* and *command* registers in the type 1 header have no bearing on the behavior of a particular root port in the `gem5` root complex model. The initial values for these registers are set manually to

indicate the behavior of the root complex, but subsequent writes to these registers have no effect other than just changing their values.

Routing of Requests and Responses

As mentioned earlier, TLP request packets are routed based on the addresses in their header in the PCI Express protocol. The same was done in our root complex. When a slave port of the root complex receives a `gem5` request packet, the packet's address is examined. The slave port then looks at the base and limit registers in the PCI-PCI bridge headers of the three VP2Ps, each corresponding to a root port. If the packet's address falls within the range defined by the memory or I/O base and limit registers of a particular VP2P, the packet is forwarded out the corresponding root port. Note that the response packet needs to be routed back to the device that issued the request packet. To route the response packet, we create a PCI bus number field in the `gem5` packet class, initialized to -1. Whenever a slave port of the root complex receives a request packet, it checks the packet's PCI bus number. If the PCI bus number is -1, it sets the PCI bus number of the request packet to be its secondary bus number, as configured by software in the header of the corresponding root port's VP2P. The upstream root complex slave port sets the bus number in the request packet to be 0. When a root complex master port receives a response, packet corresponding to a particular request, it compares the packet's bus number with the secondary and subordinate bus numbers of each VP2P. If the packet's bus number falls within the range defined by a particular VP2P's secondary and subordinate bus numbers, the response packet is forwarded out the corresponding slave port. If no match is found, the response packet is forwarded out the upstream slave port.

Compatibility with gem5 Crossbar

To make the gem5 root complex compatible with gem5 crossbars, used for connecting each root port to a downstream device, verification was needed that all the root slave ports of the root complex reported the upstream address ranges to the crossbar correctly. This is so that when the crossbar receives a DMA request packet, the connected slave port to forward it to is known. An intermediate list of address ranges was formed at each root port, created by taking the union of the individual address ranges defined by the base and limit registers in the corresponding VP2P's header. Also implemented was a way to make sure that a particular base/limit register pair was actually configured by software, so as to avoid taking invalid ranges into account. Since the address ranges obtained henceforth defined the addresses downstream of the root port, it was needed to take the complement of these address ranges to define the address ranges upstream of the root port. This complemented list of ranges was then reported to the crossbar as the list of address ranges which that particular root slave port would accept. Furthermore, whenever software wrote to a base or limit register in the header of a particular VP2P, the crossbar connected to the corresponding root slave port was informed of the updated address ranges.

PCI Express Switch

Switches serve to connect several endpoints to the PCI Express hierarchy. Like the root complex, switches perform routing of both the TLP request and response packets. PCI Express switches can be categorized into store and forward switches and cut through switches [29]. Store and forward switches wait to receive an entire packet, before processing it and sending it out the egress port. Cut through switches on the other hand start to forward a packet out the egress port before the whole packet is received, based on the packet's header alone [29]. Since gem5 deals

with individual packets, instead of bits, the created PCI-Express switch is a store and forward model. We note that a typical PCI-Express switch present on the market has a latency of 150ns and uses cut through switching [30]. A PCI Express switch consists of only one upstream and one or more downstream switch ports. Each switch port is represented by a virtual PCI-PCI (VP2P) bridge. This is in contrast to the root complex, where only the downstream ports (root ports) are represented by VP2Ps, whereas in a switch, the upstream port is represented by a VP2P too, as depicted in Fig. 3.*7*.

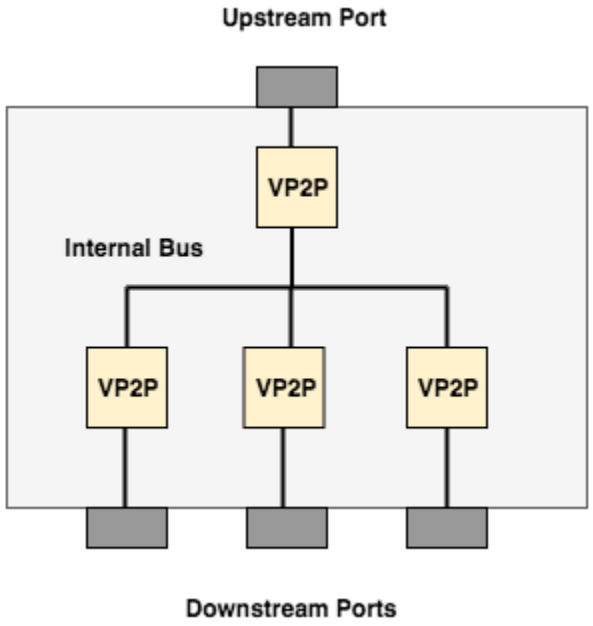

Fig. 3.7 Internal representation of PCI Express switch

The `gem5` switch model was built upon the root complex model designed earlier. Each switch port is associated with a VP2P, and is made up of a master and slave port. Values were changed in the PCI Express capability structure belonging to switch VP2Ps to indicate to enumeration software that a particular VP2P belonged to either a switch upstream or downstream port. In the root complex model, the upstream slave port accepted an address range that was the union of address ranges programmed into the VP2P headers of each root port. In this switch

model, the upstream slave port accepted an address range based on the base and limit register values stored in the upstream switch port VP2P's header. In essence, the root complex was configured to act as a switch based on an input parameter, since the only major change is that an upstream VP2P needs to be exposed to software.

3.3 Creating PCI Express Links

Once we designed and tested the `gem5` root complex, we decided to replace the `gem5` crossbar with a PCI Express link implementation, to model the performance of PCI Express. Since the pre-existing `gem5` crossbar needed a memory range for each slave, this caused additional complexity for devices situated on the crossbar. The crossbar needed to be informed of updated address ranges accepted by a slave port of a device, and consequently the device would have to inform the crossbar whenever enumeration software wrote its address registers. The created PCI Express link solves this issue since only one upstream and one downstream device can be connected to the link. Thus, there is no need for address mapping of slave ports. An overview of the created link is given, and a description of various aspects of the PCI Express protocol taken into account when implementing the PCI Express link follows.

`gem5` PCI Express Link Overview

Our PCI Express link consists of two unidirectional links, one used for transmitting packets upstream (toward Root Complex), and one used for transmitting packets downstream, as shown in Fig. 3.8. The PCI Express link provides an interface on either side of the link, to connect ports to. Each interface includes a master and slave port pair that can be connected to a component's

slave and master port respectively. For example, an interface's master port can be connected to a device's PIO port, and the interface's slave port can be connected to the device's DMA port.

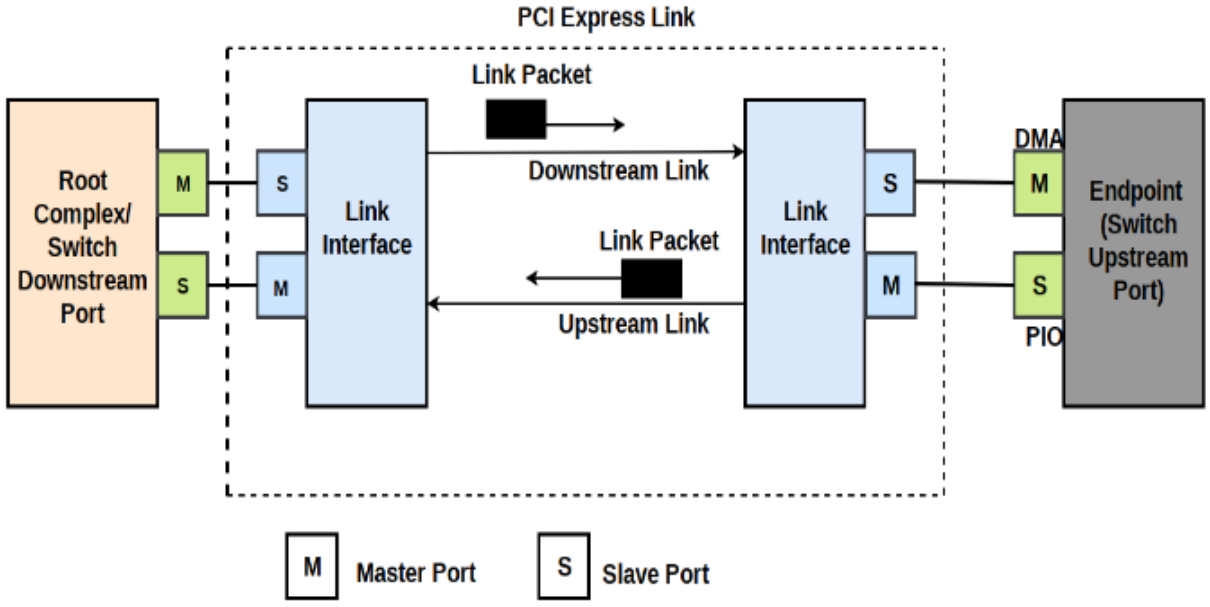

Fig. 3.8 Overview of `gem5` PCI Express link

Alternately, an interface's master port can be connected to a switch/root port's slave port and the interface's slave port can be connected to the switch/root port's master port. A unidirectional link is used to transmit packets from one interface and to deliver packets to the other interface of the PCI Express link. The unidirectional links used to form the PCI Express link are based on the unidirectional links used to implement a `gem5` Ethernet link. Each unidirectional link transmits a packet to the target interface after a particular delay, depending on the size of the packet and the configured bandwidth of the PCI Express link.

PCI Express - Layered Protocol

Every PCI Express-based device needs to implement a PCI Express interface consisting of three

layers [17]. The topmost layer, the transaction layer, interfaces directly with the device core, while the bottom layer, the physical layer interfaces with the wires that form a PCI express link.

- *Transaction layer* - This layer accepts a request from the device core, and creates a transaction layer packet (TLP). As mentioned earlier, only TLPs carry data through the PCI Express interconnect. A request TLP originates at the transaction layer of the requestor and is consumed in the transaction layer of the completer, and the opposite is true for a response TLP. In our `gem5` implementation of a PCI Express interconnect, the transaction layer is implicitly taken to be the master and slave ports of endpoints and root complex/switches, as these are where request and response packets originate.

- *Data link layer* - This layer is positioned below the transaction layer and is responsible for link management. The data link layer is a source for packets (DLLPs) that are used for flow control, acknowledging error-free transmission of TLPs across a link and for link power management [17]. A DLLP originates at the data link layer of one device and is consumed by the data link layer of the device on the other side of a link. The data link layer also appends a cyclic redundancy check and sequence number to TLPs before they are transmitted across the link. In our `gem5` implementation of PCI Express, each PCI Express link interface implements a simplified version of the Ack/NAK protocol used by the data link layer to ensure reliable transmission of TLPs across a link. In this way, we factor in some of the link traffic due to DLLPs in addition to the link traffic due to TLPs.

- *Physical layer* - The physical layer is involved in framing, encoding and serializing TLPs and DLLPs before transmitting them on the link [17]. A TLP or DLLP is first appended with STP and END control symbols, indicating the start and end of a packet respectively. The individual bytes of a TLP or DLLP are divided among the available (lanes). Each byte is scrambled and encoded using 8b/10b (128b/130b in Gen3) encoding, and is serialized and transmitted out onto the wire. We take the encoding and framing overhead due to the physical layer into account in our `gem5` PCI Express model, however we do not implement other aspects of the physical layer.

Overheads Accounted for in `gem5`

Transaction layer packets are replaced by `gem5` packets in my PCI Express implementation. However, to model PCI Express performance more accurately, we take overheads caused due to headers and encoding into account when calculating the time taken for a packet to travel across a unidirectional link. The TLP payload size is taken to be the size of the data transported in a `gem5` packet. The minimum TLP payload size is 0, in case of a read request/write response and the maximum TLP payload size is the `gem5` cache line size. We summarize the DLLP overheads and TLP overheads which were taken into account in Table 3.2 and Table 3.3.

Table 3.2: Overheads taken into account for a DLLP

Overhead type	Overhead size
Framing symbols appended by physical layer	2B
Encoding overhead	25% in Gen1 and Gen2, 1.5% in Gen3

Table 3.3: Overheads taken into account for a TLP

Overhead type	Overhead size
TLP header	12B
Sequence number appended by data link layer	2B
Link cyclic redundancy check	4B
Framing symbols appended by physical layer	2B
Encoding overhead	25% in Gen1 and Gen2, 1.5% in Gen3

Ack/NAK Protocol in the Data Link Layer of Real PCI Express Devices

We shall first explain how the Ack/NAK protocol is implemented in a real PCI Express device's data link layer, and then proceed to explain how we attempt to replicate this protocol in `gem5`.

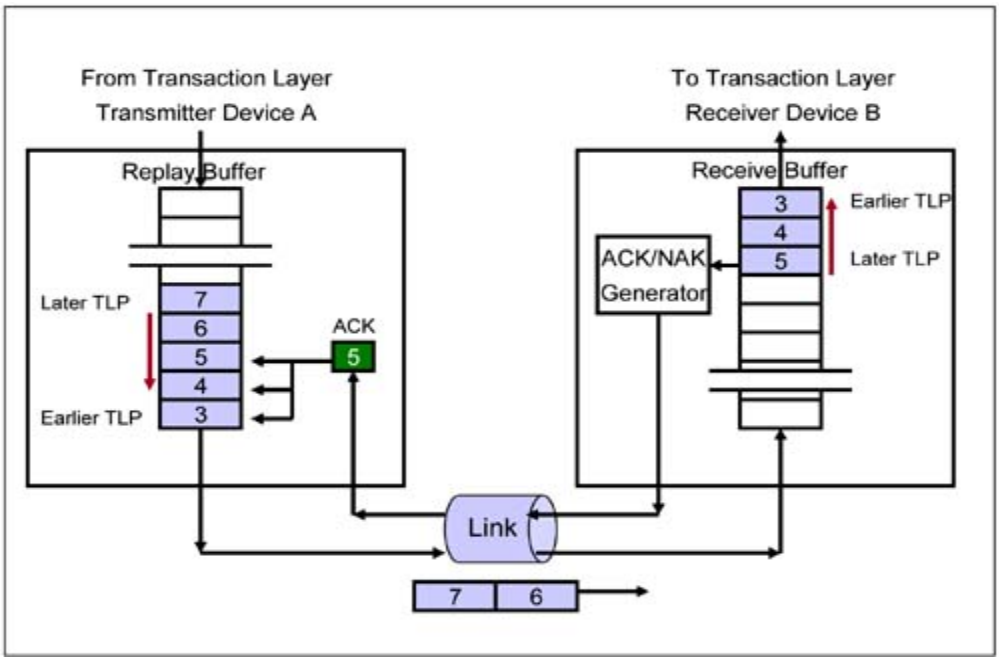

Fig. 3.9 Ack/NAK protocol example in a real system [17]

In the Sender's Side

When the data link layer of the TLP sender receives a TLP from the transaction layer, it first appends a sequence number and a CRC to the TLP. A copy of the TLP is stored in a "replay

buffer" as shown in Fig. 3.9, before the TLP is sent to the physical layer and out onto the link. The sequence number assigned is incremented for every TLP, and TLPs are stored in the replay buffer based on the order in which they arrived from the transport layer.

The purpose of the replay buffer is to hold a TLP until it is confirmed to be received without any errors by the data link layer of the device on the other end of the PCI Express link. This confirmation comes in the form of an Ack DLLP. On the contrary, if the receiver on the other end of the link (not to be confused with completer) receives a TLP with an error, it sends back an NAK DLLP across the link to the TLP sender. In this scenario, the data link layer of the sender first removes the TLPs from the replay buffer that are acknowledged by the NAK, and then retransmits (replays) the remaining TLPs present in the replay buffer in order [17]. Both the Ack and NAK DLLPs identify a TLP via its sequence number. When the TLP sender receives an Ack DLLP, all TLPs with a lower or equal sequence number are removed from the replay buffer, as indicated in Fig. 3.9, and space is freed up in the buffer. It is worth observing that a full replay buffer causes TLPs to stop being transmitted, and this is a scenario that needs to be avoided when choosing the replay buffer size. Also, the receiver's data link layer need not send Acks for every TLP successfully received, and can acknowledge multiple TLPs using a single Ack DLLP.

The data link layer of PCI Express devices also maintains a replay timer [17], to retransmit TLPs from the replay buffer on timer expiration. The timer is started from 0 when a TLP is transmitted (assuming that the timer is not already running). A timer is reset to 0 whenever an Ack DLLP is received and is restarted again if any TLPs remain in the replay buffer after the Ack is processed. On timer expiration, all TLPs from the replay buffer are retransmitted and the data link layer stops accepting packets from the transaction layer while this retransmission is happening. Whenever a retransmit event takes place, either due to an NAK

DLLP received or timer expiration, the timer is set to zero and restarted after the first packet from the replay buffer has been retransmitted. The timer is required for scenarios such as when an NAK DLLP is lost en-route to the sender or when the receiver experiences temporary errors that prevent it from sending Acks to the sender [17].

Receiver's Side

On the receiver's end, the TLPs arriving from the physical layer are buffered in the data link layer, and CRC as well as sequence number checks are performed on them, as shown in Fig. 3.9. The receiver needs to keep track of the sequence number of the TLP it expects to receive next from the link. If a buffered TLP is error free, it is passed on to the transaction layer and the "next sequence number" field is incremented. If the TLP has an error, it is discarded and the "next sequence number" remains the same.

Once a TLP is processed in the data link layer of the receiver, the receiver has the option to send an Ack/NAK back to the sender immediately. However, to avoid unnecessary link traffic, the receiver can send back a single Ack/NAK to the sender for several processed TLPs. To ensure that Ack/NAKs are sent before the replay buffer of the sender fills up or the sender experiences a timeout, the receiver maintains a timer to schedule sending Ack/NAKs back to the sender. When the timer expires, an Ack/NAK is sent back to the sender. This timer is reset to 0 when the receiver has no unacknowledged TLPs left.

The data link layer of each PCI Express device has both the sender and receiver logic for the Ack/NAK protocol described above. The "Replay_Timer" and "Ack_Timer" have an expiration period set by the PCI Express standard, which we shall describe in the next subsection.

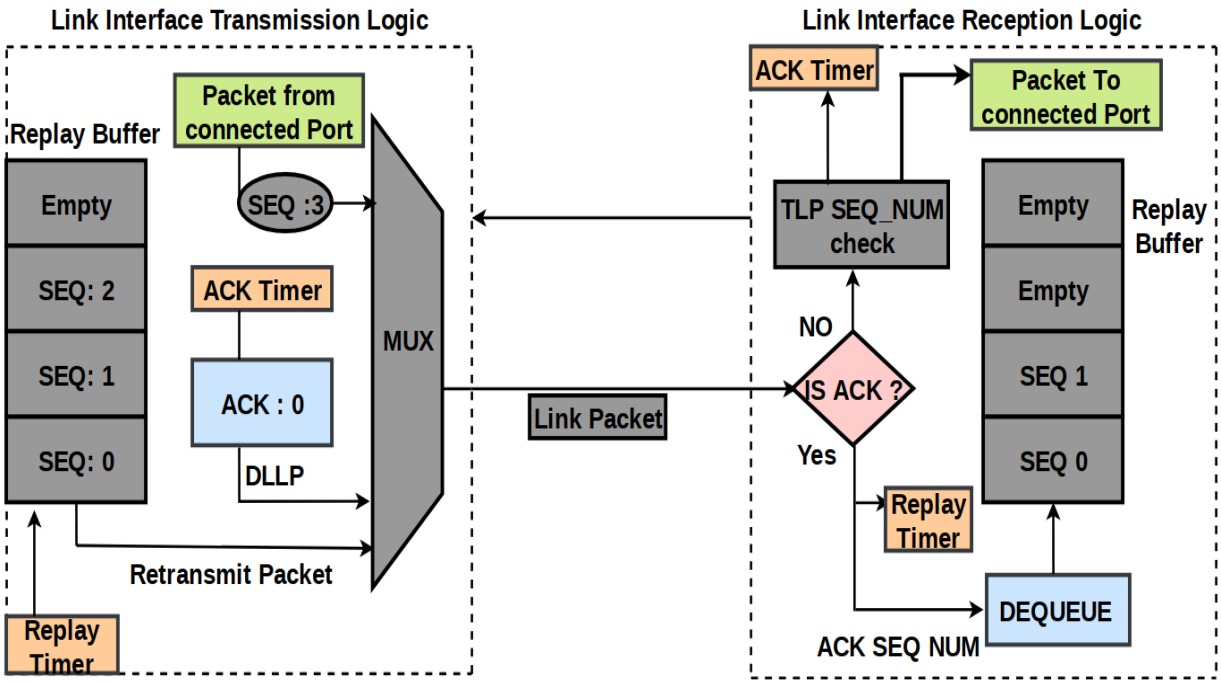

Fig. 3.10 Link interface transmission and reception logic

- *Data link layer packets*: The link interfaces act as producers and consumers of Ack DLLPs. We create a new packet class to represent DLLPs. We assume that the implemented PCI Express link is error free, and thus NAK DLLPs are not implemented. Credit-based flow control is not implemented in the PCI Express links, and thus only Acks form the DLLP traffic across the unidirectional links.

- *Transaction layer packets*: The packets used by `gem5` for memory/IO system accesses is taken to be equivalent to a PCI Express TLP. The `gem5` packet contains some of the

relevant information present in a TLP header such as requestor id, packet type (request or response), completer address and length of data to be read or written.

- *Packet transmission*: Each link interface receives a `gem5` packet from the attached slave or master port and sends it on the assigned unidirectional link. Since both DLLPs and `gem5` packets are transmitted across the unidirectional links, we create a new C++ class to encapsulate both DLLPs and `gem5` packets. We shall refer to this class as *Link Packet*. A *Link Packet* encapsulating a `gem5` packet (TLP) is assigned a sequence number prior to transmission, as shown in Fig. 3.10. Each *Link Packet* returns a size depending on whether it encapsulates a TLP or a DLLP and factors such as headers, encoding schemes, sequence numbers, etc. are taken into account while returning the size, as described in Table 3.2 and Table 3.3. The size of the *Link Packet* is used to determine the delay added by each unidirectional link when transmitting a packet.

- *Packet reception*: Each link interface receives a *Link Packet* from the corresponding unidirectional link. The TLP or DLLP is extracted from the *Link Packet*. If the packet is a TLP, its sequence number is examined before sending it to the master or slave ports attached to the interface as shown in Fig. 3.10. If the packet is a DLLP, actions are performed based on my implementation of the Ack/NAK protocol.

Ack/NAK Protocol in `gem5`

The goal of our implementation of the Ack/NAK protocol is to ensure reliable and in order transmission of packets even when the buffers of components attached to the PCI Express link

fill up. For example, if a root slave port's buffer is full, it will stop accepting packets. Instead of buffering packets in the attached link interface, packets are retransmitted from a replay buffer in our PCI Express link implementation. In real systems, PCI Express devices implement a credit-based flow control scheme, where the transaction layer of a PCI Express-based device refuses to transmit any more TLPs while the transaction layer buffers of the device at the other end of the link are full. In contrast, our PCI Express link interfaces transmit TLPs across the unidirectional links as long as their replay buffers have space. Once the replay buffer is filled up due to an absence of Acks, packet transmission is throttled. A timeout and retransmit mechanism makes sure that TLPs eventually reach their destination component when its buffers free up.

- *Replay buffer*: Each link interface maintains a replay buffer of a configured size to hold transmitted TLP *Link Packets*. The replay buffer is designed as a queue, and new *Link Packets* are added to the back of the buffer. The replay buffer size is based on the number of *Link Packets*, as opposed to a size in bytes. Hence, the replay buffer would be getting filled very quickly if `gem5` packets with size far below the maximum payload size are being transmitted.

- *Sequence numbers*: Each link interface maintains a sending sequence number and receiving sequence number respectively. The sending sequence number is assigned to each TLP *Link Packet* transmitted by the interface across the unidirectional link and is incremented, as shown in Fig. 3.10. The receiving sequence number is used to decide whether to accept or discard a *Link Packet* that the interface receives from the

unidirectional link. The receiving sequence number is incremented for every TLP *Link Packet* received if the TLP is successfully sent to the attached device.

- *Replay timer*: Each link interface maintains a replay timer to retransmit packets from the replay buffer on a timeout. The replay timer is started for every packet transmitted on the unidirectional link. The replay timer is reset whenever an interface receives an Ack DLLP, and restarted if the replay buffer has packets even after processing the Ack. The replay timer is also reset and restarted on a timeout. The timeout interval is set based on the PCI Express specification [17], which defines the timeout interval in *symbol times* as

$$\left[\frac{(\text{Max\_Payload\_Size} + \text{TLP\_Overhead}) * \text{AckFactor}}{\text{LinkWidth}} + \text{Internal\_Delay}\right] * 3 + \text{RX\_L0s\_Adjustment}$$

where one symbol time is the time taken to transmit a byte of data on the link. We set the max payload size in the above equation to be equal to the cache line size, the width as the number of lanes configured in the PCI Express Link, and the internal delay and Rx_L0s_Adjustment as 0, since we do not implement different power states or take internal delay into account. The AckFactor values are determined based on the maximum payload size and the link width, and are taken from the specification used for Gen1 PCI Express.

- *Ack timer*: The Ack timer is used to determine the maximum time after which an Ack can be returned by an interface after a successfully received TLP [16]. The timeout value is set to 1/3 of the replay timer's timeout value.

Mechanism

After a link interface transmits a TLP *Link Packet* on the unidirectional link, it stores the packet in the replay buffer. When the interface on the other end of the link receives the TLP *Link Packet*, it checks to see if the sequence number of the packet is equal to the receiving sequence number. If the check passes and the TLP is successfully sent to the connected master or slave port (ports shown in green in Fig. 3.8*)*, the receiving sequence number is incremented. In this scenario, an Ack for the corresponding sequence number is returned to the sending interface, either immediately or once the Ack Timer expires. On the other hand, if the master or slave ports connected to the receiving interface refuse to accept the TLP due to their buffers being full, the receiving interface does not increment the receiving sequence number and does not send an Ack back for the received TLP. In this scenario, a retransmission will be required from the sending interface's replay buffer after a timeout.

When a link interface receives an Ack DLLP *Link Packet*, it removes from the replay buffer all TLP *Link Packets* with an equal to or lower sequence number than the sequence number contained in the Ack. The replay timer is reset, and restarted if any packets remain in the replay buffer once the Ack is processed.

Particular care is taken to ensure that priority is given to *Link Packets* in the following order for transmission across a unidirectional link, as mentioned in [17],

(1) Currently transmitted *Link Packet*

(2) Ack DLLP *Link Packet*

(3) Retransmitted *Link Packet*

(4) *Link Packet* containing TLPs received from connected ports

CHAPTER 4: EXPERIMENTAL SETUP

4.1 Overview

To validate our gem5 model, we had to compare the performance of the implemented PCI Express interconnect against a PCI Express interconnect in a real system. Doing this was not straightforward since we needed a way to measure PCI Express bandwidth in a real system without taking into account the characteristics of the specific device transferring data through the PCI Express interconnect. In most cases, the device's transfer speed is the bottleneck and not the PCI Express link speed.

Both in gem5 and in a real system, data is read from storage devices to measure the offered PCI Express bandwidth. Data is read using the Linux utility "*dd*", which is used for copying data between different storage devices [31]. *dd* can also be used to measure a storage device's performance [31], since it reports the throughput of a data transfer. *dd* transfers data in units of "blocks", with the size of each block and the number of blocks to transfer being parameters. In the measurements made, only a single block of data is transferred at a time, with the block size varied between 64MB and 512MB. We clear kernel caches when running *dd* so that the kernel reads the entire block of data from the storage device [32]. The page cache as well as dentries and inodes are cleared. We also use the *dd iflag = direct* argument for direct I/O, avoiding the page cache and kernel buffers [33]. Using *dd*, a block of data is read from the storage device into /dev/zero [34].

4.2 Experiment Setup in Real System

The Intel p3700 Solid State Drive (SSD) is the storage device used to benchmark the

performance of the PCI Express interconnect in a real system. The p3700 provides a sequential read bandwidth of 2800 MB/s and is designed for PCI Express Gen3 x4 [35].

dd is run on a system with an Intel Xeon V4 E5 2660 processor, with a base frequency of 2.0 GHz and a maximum frequency of 3.2 GHz [36]. The processor is connected to the X99 Platform Controller Hub (PCH) through a DMI 2.0 x4 link [37]. The DMI link provides a bandwidth similar to a PCI Express Gen 2 x4 link, 20 Gbps [37]. The DMI link is connected to the CPU's Integrated I/O (IIO) block through a DMI host bridge [38]. The DMI Host Bridge exposes a header corresponding to a PCI endpoint, unlike the root ports used to connect PCI Express links to the CPU, which expose a header corresponding to a PCI bridge.

The Intel Xeon CPU has PCI Express Gen3 slots [36], where a device such as the P3700 that requires a high-speed interconnect would usually be attached to. However, the PCI Express link bandwidth needs to be the bottleneck, so the p3700 is attached to the PCH's PCI Express slot instead. The PCH contains Gen2 PCI Express links, with the slot the P3700 was attached to configured with only one lane. This limits the offered PCI Express bandwidth to the Gen2 x1 bandwidth of 5 Gbps in each direction. Taking the 8b/10b encoding overhead into account, the p3700's read bandwidth is effectively limited to a maximum of 500 MB/s.

Thus, when running *dd* to read a block of data from the p3700 attached to the PCH's PCI Express slot, the bandwidth reported by *dd* is expected to correlate with the bandwidth offered by a PCI Express Gen 2 x1 link.

After clearing caches, *dd* is run several times on the real system until its reported throughput stabilizes, using direct I/O. It is observed that although direct I/O bypasses kernel caches and reads data directly from the SSD, the read from the SSD immediately following

clearing of caches is a lot slower than subsequent reads. Thus, *dd* is run a few times using direct I/O till consistent throughput values are reported, which are used for measurements.

4.3 Experimental Setup in gem5

Table 4.1. gem5 CPU and cache configuration

gem5 component	Configuration
Processor	gem5 DerivO3CPU, 1 core, 3.2 GHz
L1 I Cache	32KB
L1 D Cache	64KB
Unified L2 Cache	2MB
IOCache	1KB
Cache Line Size	128B
RAM size	1GB

CPU and Cache Setup

A gem5 full system simulation is run using the gem5 detailed processor (DerivO3CPU), clocked at 3.2 GHz. The other parameters are shown as configured in Table 4.1. A single core CPU is used with two levels of caches. In gem5, the maximum amount of data that can be transmitted in one packet is equal to the cache line size. Since the PCI Express maximum payload size is equal to 128B for the root ports present in the X99 PCH [39], the cache line size is set to 128B in gem5, so as to make the implemented PCI Express interconnect's maximum payload size match that of the PCH. A possible drawback of increasing the cache line size to 128B in gem5 is that the Intel Xeon CPU uses a cache line size of 64B, so there could be added

disparities between gem5 measurements and measurements made on the Intel Xeon due to this difference.

Storage Device

The gem5 IDE disk is used as the device whose performance is measured with *dd*. The PCI-based IDE controller is attached to one of the gem5 switch ports, as shown in Fig. 4.1. The IDE controller is not connected directly to the root complex since the p3700 is connected to the PCH and not to the processor in the real system, and the DMI link is modeled using the implemented PCI Express link in gem5. The gem5 root complex is implicitly taken to correspond to the Intel Xeon CPU's *IIO* in this setup. The IDE disk model in gem5 transfers a page (4KB) of data at a time. However, while the IDE disk has a latency of 1µs, there is no bandwidth limitation on the time taken by the disk to transfer a page of data to memory other than the interconnect. Thus, when a block is read from the gem5 IDE disk with *dd*, the transfer time can be expected to correspond to the performance of the implemented PCI Express links.

PCI Express Setup

We set the PCI Express link between the IDE controller and switch port to Gen2 x1, as shown in Fig. 4.1, to model the PCI Express Link between the PCH and p3700 SSD. A Gen2 PCI Express link provides a bandwidth of 5 Gbps per lane in either direction, not taking encoding overheads into account. We model the DMI link between the processor and PCH in the real system by using a PCI Express Gen2 x4 link to connect the root port to the upstream switch port in gem5. The PCI Express replay buffer size [40] is configured to hold enough *Link Packets* until an Ack returns. The Ack factor determines the maximum number of maximum size TLPs [17] that can

be received by a data link layer before an Ack must be sent. Given that the Ack factor for x1, x2, x4 links is 1.4 and 2.5 for x8, we decided to configure the replay buffer size as four packets.

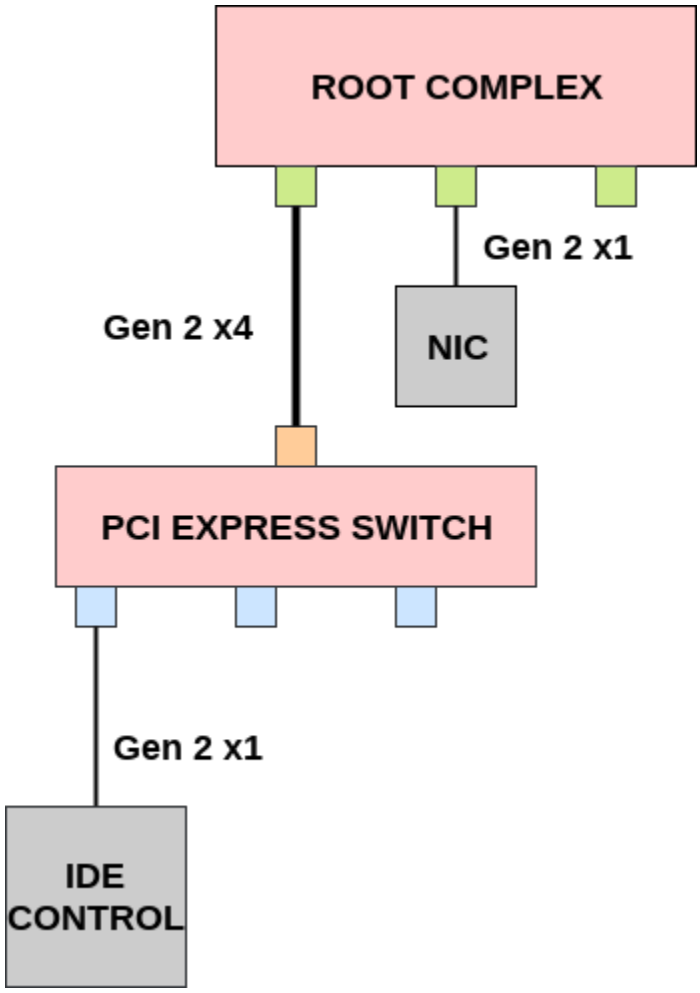

Fig. 4.1 gem5 PCI Express setup

The switch and root complex latencies are a bit more difficult to configure. While a typical market switch has a latency of 150ns, it also implements cut through switching, which the gem5 switch model does not. The gem5 root complex and switch is configured with a latency of 150ns, however we sweep the switch latency when comparing the performance of the gem5 PCI Express interconnect to the PCI Express interconnect on the real system.

CHAPTER 5: RESULTS AND EVALUATION

5.1 Comparing Performance of gem5 IDE Disk Against p3700 SSD

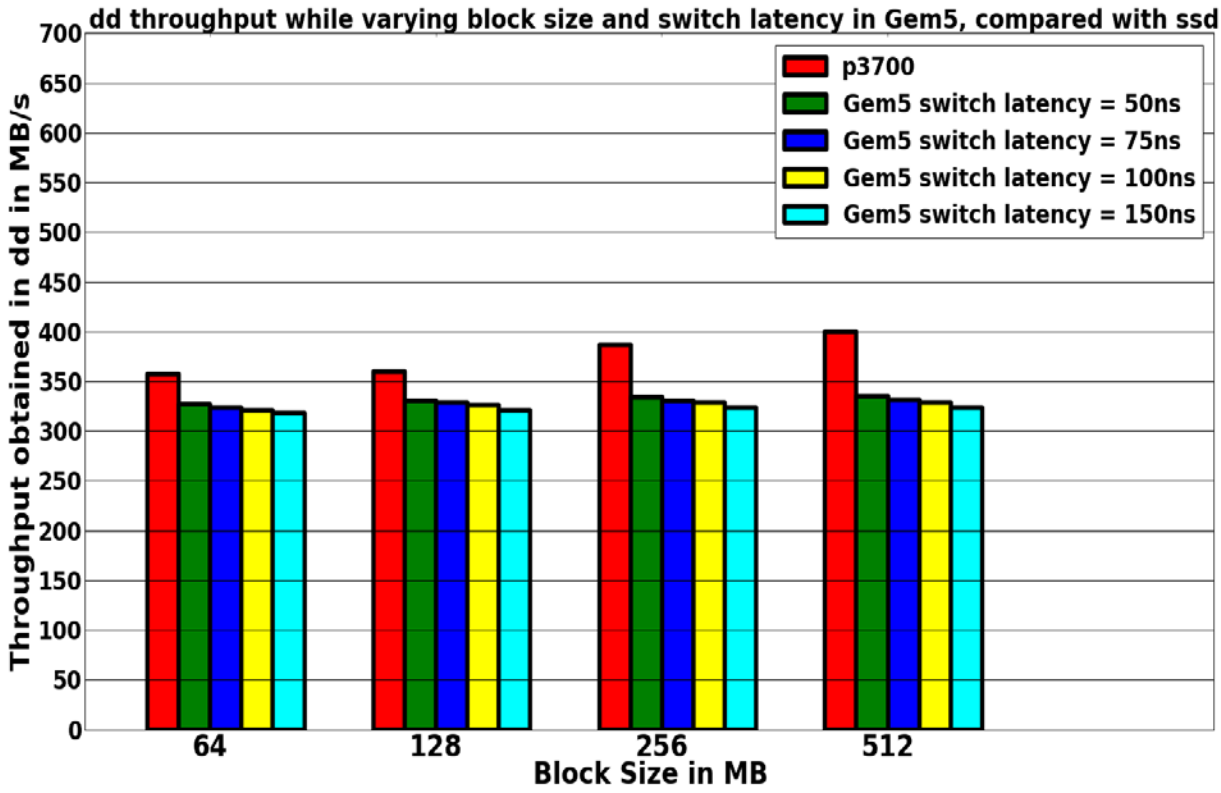

Fig. 5.1 Comparing the performance of gem5 IDE disk against p3700 SSD

In this experiment, a single block of data of different sizes is read from the storage device into /dev/zero using *dd*, both on the Intel Xeon and in gem5. Along with varying the block sizes used in *dd*, the switch latency in gem5 is swept from 150ns-50ns. The root complex latency is kept fixed at 150ns. Both the root complex and switch use buffers that can store a maximum of 16 packets per master or slave port.

The performance of the gem5 IDE disk is around 80%-90% the performance of the p3700 attached to the PCH's root port, across all block sizes, as shown in Fig. 5.1. The gem5 PCI Express interconnect is a bit slower than the PCI Express interconnect that the p3700 is connected to. However, the OS overhead in setting up the read from the storage device and the

underlying CPU type variation between the real system and gem5 cannot be ignored. There is significant overhead in communication between the gem5 IDE controller and the OS, with the IDE controller having to frequently read descriptors from system memory. These descriptors contain the addresses of buffers in RAM that the disk should transfer data into [41].

Another factor that reduces the bandwidth offered by the gem5 PCI Express interconnect is the fact that *posted write requests* are not used. Although this does not affect the performance of any single link, once a page of data is transferred by the IDE disk, responses for all gem5 write packets used to transfer that page of data need to be obtained before the next page can be transferred. This is unlike the PCI Express protocol in real systems, where write TLPs to memory space do not need a response TLP.

We notice a general trend both in the real system with the p3700 and in gem5 with the IDE disk that larger the block size used in *dd*, larger the throughput reported, as shown in Fig. 5.1. This could be due to the OS overhead forming a lower fraction of the total transfer time when transferring a larger block as compared to when transferring a smaller one.

As expected we get higher throughputs across all block sizes when we decrease the gem5 switch latency. We get close to 10 MB/s increase in throughput reported by *dd* when changing the switch latency from 150ns to 50ns, across all block sizes used. While this is a noticeable increase in throughput, this change is not more than 3%, which leads to the conclusion that the switch latency alone does not play a huge role in determining the performance of the gem5 PCI Express interconnect.

5.2 Comparing the Performance of the IDE Disk Using Different PCI Express Link Widths in gem5

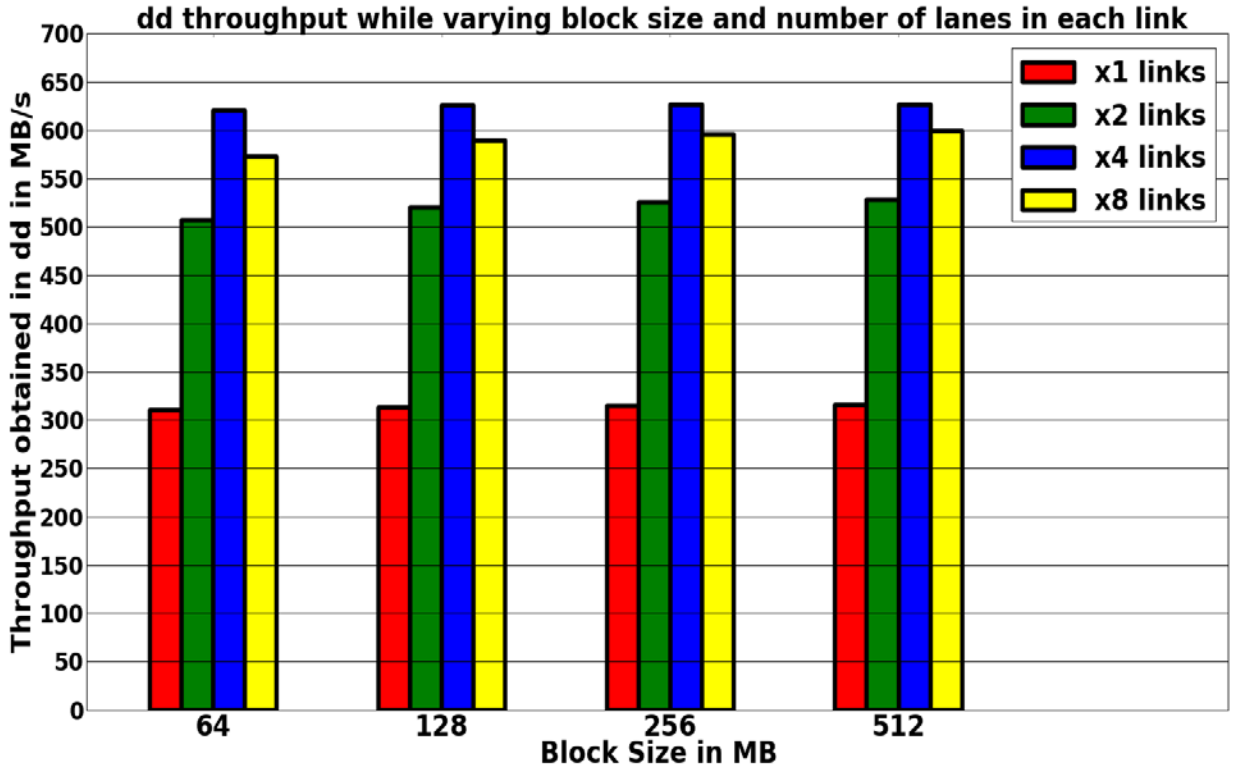

Fig. 5.2 *dd* throughput across different block sizes when varying the link width

In this experiment, Gen2 PCI Express links are still used in gem5, however the link widths are varied between x1, x2, x4 and x8 respectively. The gem5 IDE disk is still connected to a switch port, as in the previous experiment. In this experiment, all links are given the same width including the link from the root port to switch upstream port since we are no longer comparing the gem5 PCI Express interconnect with the PCI Express interconnect in the real system. The throughput reported by *dd* for different block sizes are measured and the link widths are varied.

We observe a large increase in throughput when increasing the link width from x1 to x2, as seen in Fig. 5.2. The throughput does not exactly double across all block sizes since the OS and communication overhead and switch and root complex latencies do not scale with the increase in link widths. We have a smaller increase in throughput when doubling the link width

from x2 to x4 as shown in the graph. Surprisingly, we see a drop in throughput when doubling the link width from x4 to x8, even though each individual PCI Express link's bandwidth would be doubled. This is the result of the x8 link transmitting packets too fast for the switch port to handle, causing the buffers to fill up. The number of replay timer timeouts is around 27% of the total unique packets transmitted on the link between the IDE Controller and the switch, as described in the next section.

5.3 Comparing the Performance of the IDE Disk Using x8 Links, Different Replay Buffer Sizes

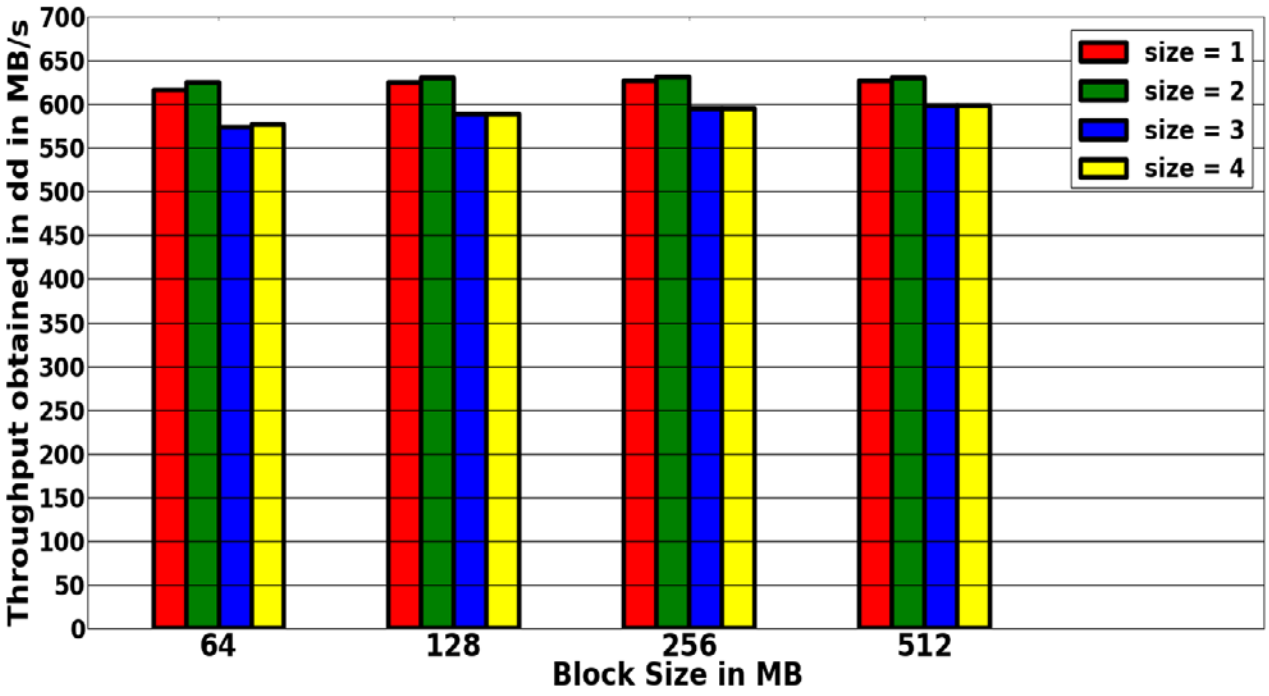

Fig. 5.3 *dd* throughput when varying replay buffer sizes in each x8 link

In this experiment, the link width is kept constant at eight lanes for all PCI Express links, and the replay buffer size in each link interface is varied. We configure the maximum number of packets a replay buffer can hold to be 1, 2, 3 and 4 and measure the performance of the IDE disk using *dd*.

We observe that the optimal replay buffer size seems to be one or two packets across all *dd* block sizes, as shown in Fig. 5.3. The larger replay buffer sizes of three and four packets hinder the performance of our PCI Express interconnect. We measure the number of replay timer timeouts that occur on the upstream unidirectional link that connects the IDE disk to the switch port, and calculate this number as a percentage of the total number of packets transmitted on the upstream unidirectional link. We find that when the replay buffer has sizes 3 and 4, the percentage of timeouts is around 27%. When the replay buffer is of size 2, the percentage of timeouts is around 6%, and there is close to 0% timeouts when the replay buffer has size 1.

A larger replay buffer ensures that more TLP *Link Packets* can be transmitted across the unidirectional link without waiting for an Ack for previously transmitted packets. However, when the link speed is high, along with a long root complex/switch latency, the root port and switch port buffers get filled up very fast, and retransmits are required. In this scenario, slowing down the arrival of packets to the switch port buffers by reducing the replay buffer size seems to help considerably to reduce timeouts.

5.4 Comparing the Performance of the IDE Disk on an x8 Link, Using Different Switch and Root Port Buffer Sizes

In this experiment, PCI Express links are kept as x8, while varying the switch and root port buffer sizes. The replay buffer size is restored to 4, and the objective is to see whether increasing the switch and root port buffer sizes increases the *dd* throughput in `gem5`. Measurements are taken for different block sizes.

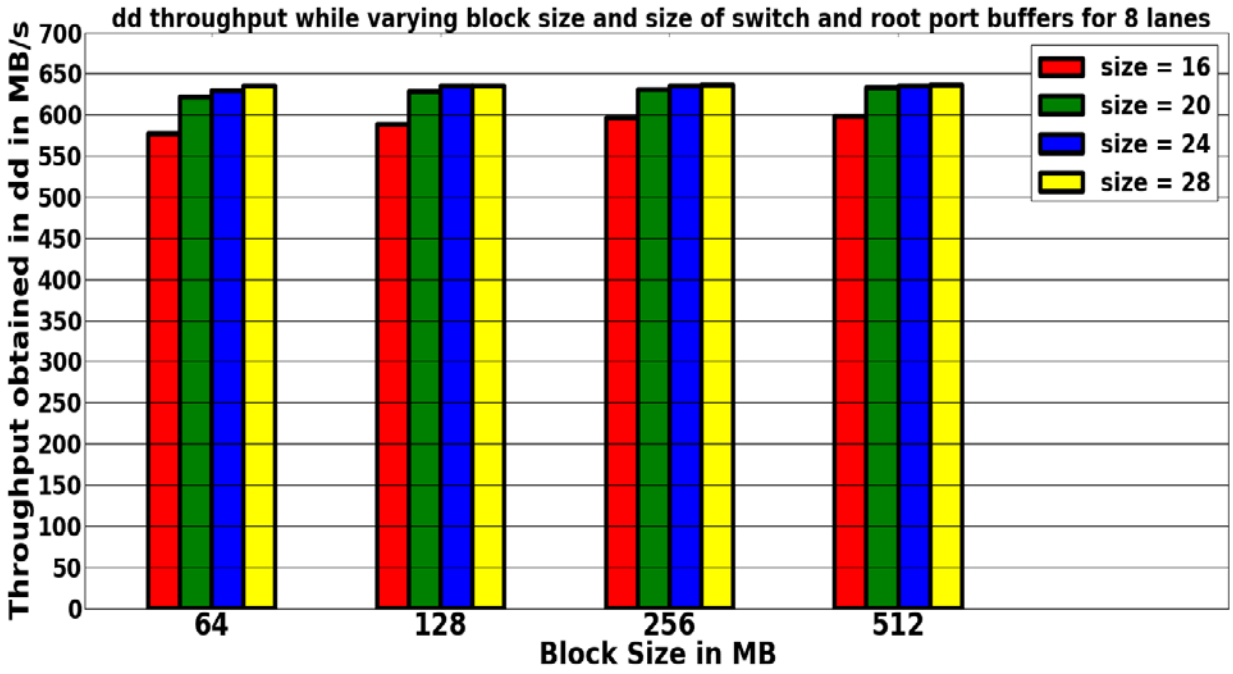

Fig. 5.4 *dd* throughput while varying the root and switch port buffer sizes

We observe that for a particular block size, there is a large increase in the *dd* throughput when increasing the switch/root port buffer sizes to 20 from 16, as shown in Fig. 5.4. When the port buffer sizes are increased to 24 and 28, we see a minor increase in the *dd* throughput, but the *dd* throughput seems to saturate at around 635 MB/s. This is higher than the value obtained with the x8 links in the previous experiment when we set the replay buffer size to 2.

Interestingly, increasing the switch and port buffer sizes to 20 reduces the timeout percentage to around 20% from 27%. However, we still see a huge increase in throughput. This leads to the conclusion that the increased throughput also comes from increased space in the root complex and switch port buffers as opposed to solely due to a reduction in timeouts on a particular link. Increasing the port buffer sizes to 24 and 28 results in a timeout percentage of almost 0. Using switch/root port buffer sizes of 24 seems to be sufficient for Gen2 x8 links, since there is no real increase in throughput when increasing the buffer sizes to 28.

5.5 Comparing Register Access Latencies with Varying Root Complex Latency

In this experiment, the `gem5` *8254x* NIC model is connected to a root port. The root complex latency is swept from 150ns - 50ns, and the time taken to perform a MMIO read of 4B from a NIC register is measured. A kernel module was created, and used to measure the time taken to access a location in the NIC's memory space, as described in [42].

Several reads to the NIC's register mapped into memory space were made. A range of read latencies were obtained corresponding to each root complex latency value. Table 5.1 shows the lower the root complex latency, the lower the range of MMIO read latencies obtained.

Table 5.1: MMIO read latencies vs root complex latency

Root complex latency (ns)	Range of latencies of MMIO reads (ns)
150	477 - 517
125	437 - 477
100	398 - 437
75	318 - 357
50	278 - 318

However, the lack of precision of MMIO read latency values obtained via the method described in [42] needs to be mentioned. Obtained latency values always differed by a multiple of around 40ns, and could not be obtained at a finer granularity. This is the reason there is an overlap of ranges found in Table 5.1. However, as can be seen, both the upper and lower limits of the range of read latency values obtained decreases along with the root complex latency.

5.6 Further Analysis and Drawbacks

The implemented PCI Express interconnect in `gem5` provides anywhere between 81% - 91.6%

of the bandwidth of the PCI Express interconnect in the real system. However, it must be stressed that there are several differences between the gem5 architecture used and the Intel Xeon's architecture, which could influence the PCI Express interconnect comparisons obtained. An example being the presence of an IOCache in gem5, which is absent in the Intel Xeon system. Also, the latency of the PCI Express switch model in gem5 was only a conservative estimate of the latency of an actual PCI Express switch, since cut through switching is not possible in gem5. Equating the PCI Express switch latency with the latency introduced by the PCH could result in further inaccuracies in results. The gem5 root complex latency was based on the switch latency, and was not modeled to mimic the latency introduced by a root complex in a real system.

In addition to architectural factors, device to OS communication overhead during a *dd* transfer plays a large part in determining the throughput obtained. It was observed that once kernel caches were cleared, the reported *dd* throughput with direct I/O in gem5 decreased, even though the same amount of data was read from the IDE disk for each *dd* run. It was also observed that reported *dd* throughputs would vary slightly over different gem5 simulations, using the same setup in gem5.

Furthermore, the PCI Express interconnect in gem5 does not implement protocols that are widely used in a real PCI Express interconnect, such as credit-based flow control [17]. Credit-based flow control prevents buffer overflow at a receiver, and also creates additional link traffic through the usage of flow control DLLPs. In addition to flow control, the gem5 PCI Express model also does not take into account different link power states, physical layer packets or power management DLLPs, which are all present in a real PCI Express interconnect. Also, the

additional latency created by the physical, data link and transport layers of a device is not taken into account in the implemented PCI Express model.

Another drawback of the model in gem5 is the absence of the MSI interrupt scheme implemented by real PCI Express devices. Instead, in the implemented PCI Express interconnect in gem5, a new interrupt scheme is used where all devices under a particular root port share the same interrupt line. This was done because the kernel enumeration software assigns different interrupt numbers to devices based on the root port they are connected to, while the pre-existing interrupt scheme for PCI devices in gem5 assigned interrupt lines based only on the device number of a particular device.

While the implemented PCI Express interconnect does have flaws involved with performance, it is able to ensure reliable transmission of data through the interconnect. The root complex and switch that were created in gem5 successfully route both gem5 request and response packets through the PCI Express hierarchy. Both the root complex and switch successfully map their registers into configuration space, and are assigned the correct address "windows" by the kernel enumeration software. Furthermore, the Ack/NAK protocol works successfully, and no gem5 packets are lost in the event of buffers getting filled up. Overheads present in a PCI Express TLP such as packet headers, along with encoding overhead is taken into account in the implemented model when calculating the time taken to transmit a packet on the link. This gives a more accurate model of a single link's performance

CHAPTER 6: CONCLUSION

The gem5 PCI Express interconnect was successfully created and consisted of a root complex, PCI Express switch and individual PCI Express links. The PCI Express switch and root complex were designed to be compatible with the existing gem5 IO architecture, and perform routing of gem5 packets. Each port of the switch and the root ports of the root complex is associated with a virtual PCI-PCI bridge, which is configured by kernel enumeration software and contains information needed by the root complex and switch to route request and response packets.

The created PCI Express links were based upon the existing gem5 Ethernet link. Each PCI Express link can be configured with a bandwidth value based on the number of lanes in the link and the PCI Express generation of the link. Each link takes overheads present in a PCI Express TLP and DLLP into account, including headers and encoding. Each PCI Express link also implements the Ack/NAK protocol for reliable transmission, where a timeout mechanism ensures that packets are retransmitted across the link after buffers free up, in the event of congestion.

The PCI Express interconnect in gem5 was evaluated against a PCI Express interconnect in a real system. The Linux utility *dd* was used to compare the performance of the gem5 IDE disk against an Intel p3700 SSD on a real system. The p3700 was attached to the PCI Express interconnect in a real system in such a way that the speed of the interconnect was the bottleneck, while the gem5 IDE disk was attached to the implemented PCI Express interconnect. The implemented PCI Express interconnect in gem5 provides between 81% - 91.6% of the bandwidth provided by the real PCI Express interconnect. However, several factors such as vastly differing architectural characteristics of gem5 and the real system along with OS - device communication overheads involved in the *dd* data transfer could affect the accuracy of the result.

Overall, while the gem5 PCI Express interconnect is functionally correct, it needs more stringent evaluation against a real PCI Express interconnect, across different PCI Express generations and link widths. A more accurate timing model for the created PCI Express switch and root complex needs to be developed too. The PCI Express interconnect created forms a good base upon which a more accurate model for PCI Express in gem5 can be developed. The basic components needed for a PCI Express interconnect are now present in gem5, and the PCI Express links have taken some aspects of the PCI Express protocol such as overheads and the Ack/NAK protocol into account. The gem5 PCI Express interconnect is tentatively close to the real PCI Express interconnect it was evaluated against, and provides good accuracy for lower link widths.

REFERENCES

[1] "PCI Express," *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/PCI_Express. [Accessed: 01- Jul- 2018].

[2] "Expansion card," *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Expansion_card. [Accessed: 01- Jul- 2018].

[3] "What Is PCI Express?" *Lifewire*, 2018. [Online]. Available: https://www.lifewire.com/pci-express-pcie-2625962. [Accessed: 01- Jul- 2018].

[4] "Intel® Ethernet Controller I217 Delivers Superior Connectivity," *Intel*, 2018. [Online]. Available: https://www.intel.com/content/www/us/en/embedded/products/networking/ethernet-controller-i217-family-brief.html. [Accessed: 01- Jul- 2018].

[5] "A Case for PCI Express as a High-Performance Cluster Interconnect," *HPCwire*, 2018. [Online]. Available: https://www.hpcwire.com/2011/01/24/a_case_for_pci_express_as_a_high-performance_cluster_interconnect/. [Accessed: 01- Jul- 2018].

[6] "Single Root I/O Virtualization (SR-IOV)," *Docs.microsoft.com*, 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/single-root-i-o-virtualization--sr-iov-. [Accessed: 01- Jul- 2018].

[7] Y. Chung, "Hardware Support Virtualization," 2013.

[8] G. Torres, "Everything You Need to Know About the PCI Express - Hardware Secrets," *Hardware Secrets*, 2018. [Online]. Available: https://www.hardwaresecrets.com/everything-you-need-to-know-about-the-pci-express/. [Accessed: 02- Jul- 2018].

[9] "PCIe over Cable Goes Mainstream | One Stop Systems," *Onestopsystems.com*, 2018. [Online]. Available: http://www.onestopsystems.com/blog-post/pcie-over-cable-goes-mainstream. [Accessed: 02- Jul- 2018].

[10] J. Hruska, "Start Your Engines: PCI Express 4.0 Standard Officially Released – ExtremeTech," *ExtremeTech*, 2018. [Online]. Available: https://www.extremetech.com/computing/257905-start-engines-pci-express-4-0-standard-officially-released. [Accessed: 02- Jul- 2018].

[11] A. Sandberg, "Architectural exploration with gem5," in *ASPLOS*, Xian, 2017.

[12] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, D. Wood, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower and T. Krishna, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, p. 1, 2011.

[13] "Disk images - gem5," *Gem5.org*, 2018. [Online]. Available: http://gem5.org/Disk_images. [Accessed: 02- Jul- 2018].

[14] B. Beckmann, "The gem5 simulator," in *ISCA*, San Jose, 2011.

[15] "O3CPU - gem5," *Gem5.org*, 2018. [Online]. Available: http://gem5.org/O3CPU. [Accessed: 02- Jul- 2018].

[16] "PCI Local Bus (Writing Device Drivers)," *Docs.oracle.com*, 2018. [Online]. Available: https://docs.oracle.com/cd/E19455-01/805-7378/hwovr-22/index.html. [Accessed: 02- Jul- 2018].

[17] R. Budruk, D. Anderson and T. Shanley, *PCI Express System Architecture*. Boston, MA.: Addison-Wesley, 2012.

[18] "The Linux Kernel," *Tldp.org*, 2018. [Online]. Available: http://tldp.org/LDP/tlk/tlk.html. [Accessed: 02- Jul- 2018].

[19] "Memory-mapped I/O," *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Memory-mapped_I/O. [Accessed: 02- Jul- 2018].

[20] "PCI configuration space," *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/PCI_configuration_space. [Accessed: 02- Jul- 2018].

[21] "PCI Express packet latency matters," *Mindshare.com*, 2007. [Online]. Available: https://www.mindshare.com/files/resources/PLX_PCIe_Packet_Latency_Matters.pdf. [Accessed: 02- Jul- 2018].

[22] M. Sadri, "What is an AXI interconnect ?" *Youtube.com*, 2018. [Online]. Available: https://www.youtube.com/watch?v= BASCRxR2L-c&list=PLfYnEbg9Uqu5q6-XcfJkMN7O0P0dwJCn7. [Accessed: 02- Jul- 2018].

[23] D. Gouk, J. Zhang and M. Jung, "Enabling realistic logical device interface and driver for NVM Express enabled full system simulations," *International Journal of Parallel Programming*, 2017.

[24] "DWTB: PCI Express Switch Enumeration Using VMM-Based DesignWare Verification IP," *Synopsys.com*, 2018. [Online]. Available: https://www.synopsys.com/dw/dwtb.php?a=pcie_switch_enumeration. [Accessed: 02- Jul- 2018].

[25] J. Corbett, A. Rubini and G. Hartman, *Linux Device Drivers*. O'Reilly & Associates Inc, 2014.

[26] "Intel® 82574 Gigabit Ethernet Controller Family: Datasheet," *Intel*, 2014. [Online]. Available: https://www.intel.com/content/www/us/en/embedded/products/networking/82574l-gbe-controller-datasheet.html. [Accessed: 02- Jul- 2018].

[27] "PCI Express Architecture in a nutshell," Technion, 2016.

[28] "The PCI ID Repository," *Pci-ids.ucw.cz*, 2018. [Online]. Available: http://pci-ids.ucw.cz/. [Accessed: 02- Jul- 2018].

[29] M. Rodriguez, *Addressing latency issues in pcie*. 2009.

[30] *Product Brief PEX 8796, PCI Express Gen3 switch, 96 lanes, 24 ports*. PLX Technology, 2013.

[31] "Dd (Unix)," *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Dd_(Unix). [Accessed: 02- Jul- 2018].

[32] "A - Z Linux Commands - Overview with Examples," *Tecmint.com*, 2018. [Online]. Available: https://www.tecmint.com/linux-commands-cheat-sheet/. [Accessed: 02- Jul- 2018].

[33] "GNU Coreutils: dd invocation," *Gnu.org*, 2018. [Online]. Available: https://www.gnu.org/software/coreutils/manual/html_node/dd-invocation.html. [Accessed: 02- Jul- 2018].

[34] "/dev/zero," *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki//dev/zero. [Accessed: 02- Jul- 2018].

[35] *Intel solid-state drive dc p3700 series.*Intel. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/ssdc/hpssd/sb/Intel SSD DC P3700 Series PCIe Product Specification-005.pdf

[36] "Intel® Xeon® Processor E5-2660 v4 (35M Cache, 2.00 GHz) Product Specifications," *Intel® ARK (Product Specs)*, 2018. [Online]. Available: https://ark.intel.com/products/91772/Intel-Xeon-Processor-E5-2660-v4-35M-Cache-2_00-GHz. [Accessed: 02- Jul- 2018].

[37] "Intel X99," *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Intel_X99. [Accessed: 02- Jul- 2018].

[38] *Intel® Xeon® Processor E5 v4 Product Family Datasheet, Volume Two: Registers*. Intel, 2016.

[39] *Intel C610 series chipset and Intel X99 Chipset Platform Controller Hub datasheet*. Intel, 2015.

[40] J. Wrinkles, *Sizing of the replay buffer in PCI Express devices*, 2003.

[41] "ATA/ATAPI using DMA - OSDev Wiki," *Wiki.osdev.org*, 2018. [Online]. Available: https://wiki.osdev.org/ATA/ATAPI_using_DMA. [Accessed: 02- Jul- 2018].

[42] R. RajKumar. “[Kernel Programming #7] ktime_get() and do_gettimeofday() APIs Use.” 2013. [Online]. Available: practicepeople.blogspot.com/2013/10/kernel-programming-7-ktimeget-and.html.