Simulating PCI-Express Interconnect for Future System Exploration

Mohammad Alian*, Krishna Parasuram Srinivasan*, Nam Sung Kim
Electrical and Computer Engineering, University of Illinois, Urbana-Champaign
Email: {malian2, kprini2, nskim}@illinois.edu

Abstract—**The PCI-Express interconnect is the dominant interconnection technology within a single computer node that is used for connecting off-chip devices such as network interface cards (NICs) and GPUs to the processor chip. The PCI-Express bandwidth and latency are often the bottleneck in the processor, memory and device interactions and impacts the overall performance of the connected devices. Architecture simulators often focus on modeling the performance of processor and memory and lack a performance model for the I/O devices and interconnections. In this work, we implement a flexible and detailed model for the PCI-Express interconnect in a widely known architecture simulator. We also implement a PCI-Express device model that is configured by a PCI-Express device driver. We validate our PCI-Express interconnect performance against a physical Gen 2 PCI-Express link. Our evaluation results show that the PCI-Express model bandwidth is within 19.0% of the physical setup. We use our model to evaluate different PCI-Express link widths and latency and show its impact on the overall I/O performance of an I/O intensive application.**

I. Introduction

The failure of Dennard scaling and the increase in the dark silicon area on a multi-core chip resulted in a shift from the single node to distributed computing and the use of off-chip accelerators such as GPUs and FPGAs [1], [2]. The I/O performance plays an important role in the performance and power efficiency of such modern systems [3], [4]. However, in most of the computer architecture studies, the common practice is to only model the performance of the processor and memory subsystems while ignoring that of I/O subsystems.

`gem5` is the state of the art architecture simulator that is widely used for the performance modeling of a wide range of computer systems. Several efforts have been made to add performance models for different I/O devices, such as NIC, GPU, SSD, and NVM, to the mainstream `gem5` [4]–[8]. Although these effort are valuable and help researchers to more accurately model a full computer system, the current `gem5` release lacks a device model for an off-chip interconnection and all the devices are connected through a crossbar to the processor chip. Having a detailed off-chip interconnection model is necessary to accurately model the interactions between a processor and off-chip devices.

PCI-Express is the dominant interconnection technology for connecting off-chip devices such as NICs, accelerators (*i.e.*, GPU), and storage devices (*i.e.*, SSD) to a computer system. Unlike a PCI bus that is shared by several devices, PCI-Express provides a virtual point-to-point connection between a device and a processor, enabling the processor to simultaneously communicate with multiple devices. PCI-Express significantly improves the bandwidth between the processor and a device. Nevertheless, the PCI-Express bandwidth and latency often limit the performance of the modern devices such as 100Gbps NICs and GPUs. For example, a recent Intel® Xeon™ processor [9] provides 6 DDR4 memory channels with an aggregate bandwidth of 153.6GBps. In contrast, an Intel® Xeon™ processor offers 48 PCI-Express Gen 3 lanes with an aggregate bandwidth of 48GBps. Therefore, this is critical to accurately model the PCI-Express bandwidth and latency to capture the exact behaviour of modern off-chip devices.

In this work, we provide a generic template for enabling and implementing a PCI-Express device in `gem5`. Moreover, we implement models for a set of PCI-Express components, such as Root Complex, Switch, and Link, to enable accurate modeling of a full computer system with off-chip devices. We validate our model against a physical PCI-Express system and show that the PCI-Express performance impacts the I/O throughput and possibly the overall performance of an application.

II. Background

In this section we provide some background on PCI bus and PCI-Express interconnect and explain their main differences.

A. PCI Bus

Figure 1 illustrates the PCI bus architecture. PCI is a parallel bus interconnect where multiple I/O devices share a single bus, clocked at 33 or 66 MHz [10]. Different PCI buses are connected together via PCI Bridges to form a hierarchy. Each device on a PCI Bus is capable of acting as a bus master [11], and thus DMA transactions are possible without the need for a dedicated DMA controller [10].

Each PCI Bus can support up to 32 devices. Each device is identified by a bus, device and function number. A PCI device maps its configuration registers into a configuration space, that is an address space exposing the PCI specific registers of the device to *enumeration software* and the corresponding driver [12]. Enumeration software is a part of BIOS and operating system kernel that attempts to detect the devices installed on each PCI Bus (slot), by reading their Vendor

* Both authors contributed equally to this work.

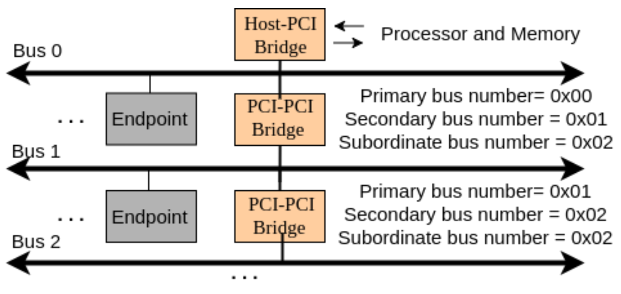


Fig. 1. PCI bus architecture

and Device ID configuration registers [13]. The enumeration software performs a depth-first search to discover devices on all the PCI buses in the system [10]. This allows a device driver to identify its corresponding device and communicate with it through a memory mapped I/O.

PCI devices are grouped into *bridges* and *endpoints*. Bridges are configured with 3 bus numbers [12]: (1) a primary bus number denoting the upstream bus that the bridge is connected to; (2) a secondary bus number denoting the immediate downstream bus of the bridge; and (3) a subordinate bus number denoting the downstream bus with the largest number. A PCI bridge transfers configuration transactions from its primary interface to its secondary interface if the transactions are destined for a device with a bus number which lies in between the secondary and subordinate bus number of the bridge [14]. In addition to the bus numbers, each bridge is programmed with memory regions that are used to transfer memory or I/O transactions from the primary to a secondary interface. Endpoints are devices that are connected to only one PCI bus, and do not allow transactions to go through them (they either send request or consume a request). A PCI bus enumeration ends at an endpoint.

A PCI bus does not support split transactions. When a PCI bus master communicates with a slave, a wait state is used if the slave is not ready to supply or accept the data. After spending a certain time at a wait state, the slave device (*i.e.* an endpoint or a bridge) signals the master to stop the transaction, to allow other masters to utilize the bus. Thus, PCI bus has a low efficiency, where only approximately half of the bus cycles are actually used to transfer data [10]. Furthermore, although the operating system supports up to 32 PCI devices to be enumerated, a single PCI bus can support only 12 electrical loads. Therefore, PCI supports at most 12 devices per bus [10]. Lastly, a PCI bus has only 4 interrupts lines that is shared between all the installed PCI devices.

B. PCI-Express Interconnect

PCI-Express is a layered protocol that defines data exchange between peripheral devices, CPU and memory. PCI-Express interconnect, from bottom up, consists of the following layers: *physical*, *data link*, and *transaction* layers [10]. A PCI-Express interconnect consists of multiple point-to-point links between a pair of devices [10]. Each link can be broken down into one or more lanes. Each lane consists of two pairs of differential signaling wires one pair for transmission in each direction. A

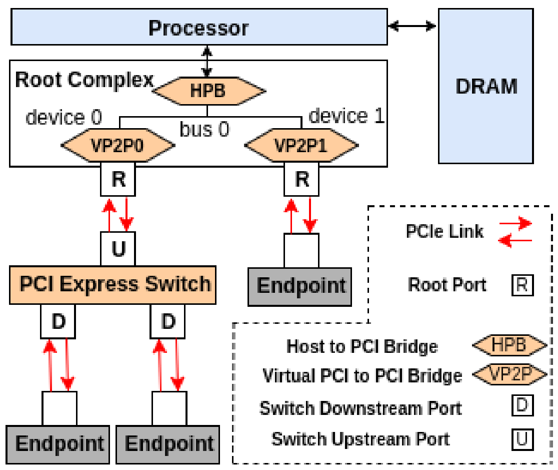


Fig. 2. PCI-Express interconnection architecture

PCI-Express Link can have up to 32 lanes. PCI-Express links are capable of transmitting data at a rate of 2.5 Gbps, 5 Gbps, and 8 Gbps in a single direction per lane in generation 1 (Gen 1), 2 (Gen 2), and 3 (Gen 3), respectively.

In contrast to the PCI standard, where devices share a single bus, a PCI-Express link has only 2 devices. As shown in Figure 2, switches are used to connect multiple devices together in the PCI-Express fabric. A PCI-Express switch has a single upstream port and multiple downstream ports. A switch is characterized by its latency, which is described as the time between the first bit of a packet arriving at an ingress port and the first bit of a packet leaving the egress port [15].

Similar to a PCI device, a PCI-Express device offers three address spaces to the enumeration software and the device driver as follows: *Configuration space* which is used by the device to map its configuration registers into; *memory space* which is used by a device to map the device-specific registers to the memory space to facilitate Memory Mapped I/O (MMIO) accesses by the device driver; and *I/O space* which is used for port-mapped I/O (PMIO) [1] by the device driver. The configuration space of a device is initially accessed during the enumeration to discover the identity of the device and its memory and I/O space requirements, and also assign interrupt resources to the device. A PCI-Express device driver is provided with the base address assigned to the device in memory and I/O space, which is used to access device specific registers at particular offsets from the provided base address. Routing components (*e.g.* switches) along the path from the CPU to the device are informed of the address ranges occupied by the device (*i.e.* memory and I/O space) [10].

PCI-Express is a packet-based protocol, where packets are exchanged between the devices. There are several different types of packets in the protocol such as *request*, *response*, *message*, and *link-specific* packets. Requests, responses, and messages can carry data and are classified as transaction layer packets (TLP). A TLP consist of a header and optionally a payload. The header is used for routing and contains information

[1]PMIO or isolated I/O is a method for performing I/O which is the complementary of the MMIO method [16]

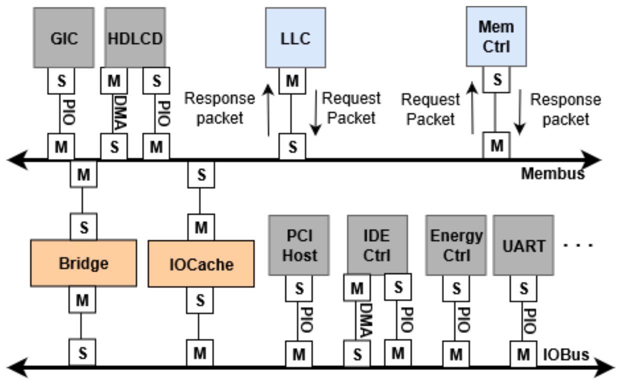


Fig. 3. gem5 off-chip interconnection architecture. "S" and "M" represent Slave and Master ports, respectively.

such as the destination address, payload size, and transaction type (*i.e.* request or response). A request originates at a *requestor*, and are routed to a *completor* based on the address in the packet header. A write request packet contains data while a read request carries no data. A response packet originates from the completor and is forwarded to the requestor. The response is routed based on the requestor's bus, device and function number. In some instances, a response packet is not needed for a request packet. These requests are called *posted requests* [10]. A *message* is a posted request that is mainly used for implementing message signaled interrupts (MSI). A device uses MSI to write a programmed value to a specified address location in order to raise an interrupt [17]. The use of packets instead of bus cycles allows split transactions since a link is occupied only for the transmission time of a packet across the link.

A root complex connects the PCI-Express interconnect to the processor and Memory. As show in Figure 2, it consists of a host to PCI bridge (HPB) and a virtual PCI to PCI bridge (VP2P) for each root port [18]. The internal root complex bus is enumerated as bus 0, and each VP2P is enumerated as a device on the bus 0. The root complex converts I/O and memory requests from the processor into TLPs destined for a particular PCI-Express device. The root complex also converts DMA transaction requests from devices into memory requests. In this way, the root complex acts both as a requester as well as a completor [10].

III. GEM5 OFF-CHIP INTERCONNECT

We use ARM ISA in `gem5` with machine type[2] set to "Vexpress_GEM5_V1" to test our PCI-Express model. Figure 3 shows an overview of the overall architecture of the Vexpress_GEM5_V1 platform. The connected devices in this platform are either on-chip (*e.g.* generic interrupt controller (GIC) and LCD controller (HDLCD)) or off-chip devices (*e.g.* UART, IDE controller, etc.). 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 ranges must be assigned to the PCI devices in gem5.

In the ARM Vexpress_GEM5_V1 platform, 256MB (address range 0x30000000 - 0x3fffffff), 16MB (address range 0x2f000000 - 0x2fffffff), and 1GB (address range 0x40000000 - 0x80000000), is assigned for the PCI configuration space, I/O space, and Memory space, respectively. DRAM is mapped to addresses from 2GB to 512GB. Because the address spaces of the PCI devices is mapped to the physical addresses lower than 2GB, all PCI devices can use 32 bit address registers instead of 64 bit ones.

On-chip devices, caches and the memory controller reside on a coherent crossbar implementation in `gem5`, called *MemBus*. Off-chip devices reside on a non coherent crossbar called *IOBus*. A crossbar in `gem5` has multiple master and slave ports, each of which is used for sending/receiving requests to/from a connected slave/master device, respectively. These crossbars are loosely modelled based 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 [19]. In `gem5`, a crossbar has latency associated with making forwarding decision as well as for moving data from one port to another.

All `gem5` memory transactions are represented in the form of a packet, which is transported through the memory system depending on its destination address. This packet based nature of the gem5 memory and I/O systems makes modelling PCI-Express packets easier. Every component in `gem5` memory and I/O subsystems needs a standard interface to send and receive packets. This is implemented in the form of ports. 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 implement two ports: a PIO slave port to accept requests originated from the processor, and a DMA master port to send DMA request packets to the memory through the interconnect. In `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 any requests from the processor 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 an address based on the bus, device, and function numbers. The PCI Host maps 256 MB of configuration space to itself with a base address of 0x30000000, where up to 4096 bytes of configuration registers can be accessed per function of a device. This method is called *Enhanced Configuration Access Mechanism* (ECAM) [10], [20]. On receiving a configuration request from the processor, the PCI Host parses the bus, device and function numbers from the request and passes the request packet to a corresponding device that has registered with it. If there is no registered

[2]machine type represents different ARM evaluation board platforms

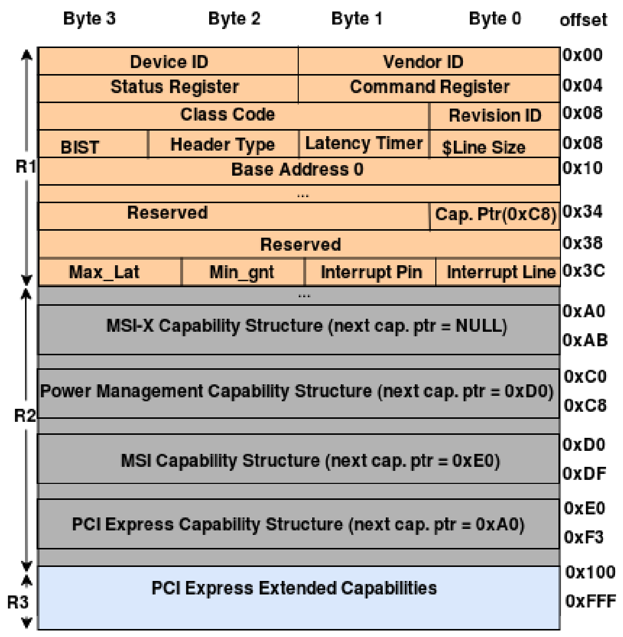


Fig. 4. Configuration space of a (endpoint) device in gem5. R1, R2, and R3 show the *PCI/PCI-Express endpoint configuration header (header type 0)*, *PCI/PCI-Express capabilities space*, and *PCI-Express extended capabilities space*.

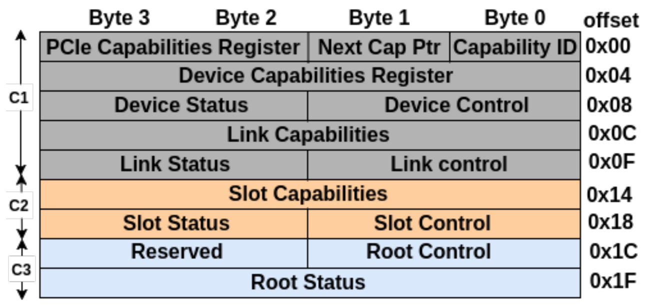


Fig. 5. PCI-Express capability structure. **C1**, **C2**, and **C3** show the registers implemented by *all the PCI-Express functions*, *only the ports connected to a slot*, and *only the root port* respectively.

device matching the configuration request's target, the PCI Host simply fills the data field of the request packet with 1's and sends back a response to the processor. In the PCI-Express protocol, a configuration response packet with its data field set to 1's represents an attempted access to a non-existent device [10].

A bridge is used to connect the Membus to the IObus. It is a slave device on the Membus and a master device on the IOBus. The bridge is configured to accept packets that are destined for an address in the off-chip address range from the Membus. We use the gem5 bridge model and build a root complex and a PCI-Express switch model upon that. In addition to the bridge, gem5 employs an IOCache, a small cache which is used to ensure the coherency of DMA accesses from the off-chip devices as well as act as a bandwidth buffer between connections of different widths.

IV. PCI-Express Device Model and Driver

In this subsection we explain how we enable a device driver designed for a PCI-Express based device to successfully detect and configure a corresponding device in gem5. Since all the device models in gem5 are for PCI based devices, we choose a PCI based device model, make certain changes, and get it work with a driver designed for a PCI-Express device. As our base device, we take the Intel 8254x NIC model in gem5 and enable the Intel e1000e device driver to detect and configure the NIC. We call the new NIC model *8254x-pcie*. The e1000e driver is used for the PCI-Express based Intel 82574l NIC. Although we enable a specific PCI-Express device model, it can serve as a template for the future PCI-Express device model developments in gem5. We explain our changes in the context of our 8254x-pcie model.

The configuration space of PCI and PCI-Express devices contains the configuration registers that is needed by the enumeration software [21] and the device driver to detect and configure the device. Figure 4 shows the configuration space of the devices in gem5. A PCI device has a configuration space of 256B per function [3] (**R1** + **R2** in Figure 4), while a PCI-Express device has configuration space of 4KB per function (**R1** + **R2** + **R3** in Figure 4) [10]. Both PCI and PCI-Express devices expose a standard 64B "header" (**R1** in Figure 4) to the enumeration software. A PCI-Express device must implement the PCI-Express capability structure in the first 256B of the configuration space. Unlike a PCI device, a PCI-Express device can implement extended capability structures starting from offset 0x100 of the configuration space (**R3** in Figure 4). These extended capabilities are configured by the device driver and include features such as advanced error reporting and virtual channels [10].

A device driver exposes a *Module Device Table* to the kernel [22], which lists the *Vendor ID* and *device ID* of all the devices supported by that driver. We set the *Device ID* register in the 8254x-pcie configuration header to 0x10D3 to invoke the probe function of the e1000e driver.

The baseline gem5 implements the *endpoint configuration header* (*i.e.* **R1** in Figure 4), but it does not implement *PCI/PCI-Express capability* and *PCI-Express extended capability* spaces (**R2** and **R3**, respectively). There are four capability structures defined by gem5 in the *PCI/PCI-Express capability space*: MSI-X, power management, MSI, and PCI-Express capability structures. As shown in Figure 5, each capability structure consists of a set of registers, and is identified by a *Capability ID*. The capability structures are organized in a chain, where the next capability pointer (*Next Cap Ptr* in Figure 5) points to the next capability structure within a device's configuration space.

Based on the datasheet [23], an Intel 82574l NIC implements power management (PM), MSI and MSI-X capability structures in addition to the PCI-Express capability structure.

[3]We assume single function devices and use "device" and "function" interchangeably.

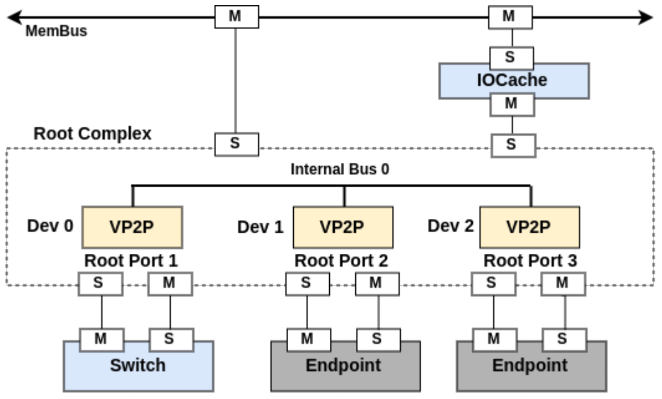


Fig. 6. Overview of our root complex model.

We set the capability pointer register (*Cap Ptr* in Figure 4) to point to the address of the PM capability structure. The *Next Cap Ptr* of the PM points to the MSI capability structure, which in turn is followed by the PCI-Express and MSI-X capability structures, respectively. Since there is no support for PM, MSI and MSI-X in `gem5`, we disable these capabilities by appropriately setting register values in each structure. As a result, the device driver is forced to register a legacy interrupt handler instead of MSI or MSI-X. The PCI Express capability structure shown in Figure 5, is configured to trick the device driver into believing that the device is installed on a PCI-Express link, even though it can still be connected through a `gem5` crossbar.

V. Modeling PCI-Express Components

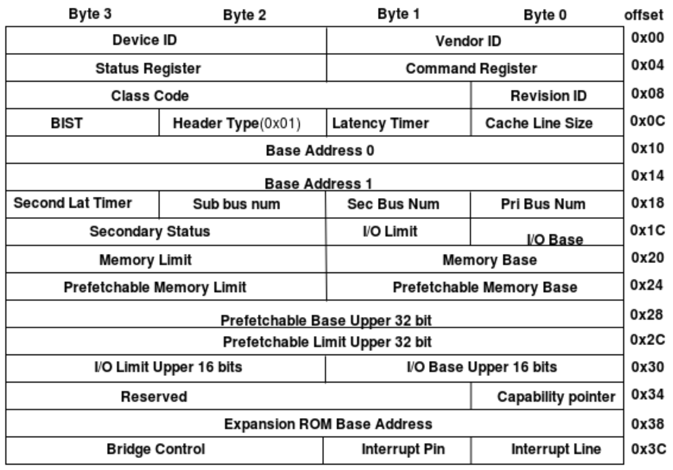
In the Sec. IV, we explained how to enable a device model in `gem5` to be detectable by the enumeration software and PCI-Express driver in the Linux kernel, regardless of the physical layer organization. In this subsection, we explain our event-driven performance models for the PCI-Express root complex, switch, and links to accurately model the behaviour of a physical interconnect.

A. Root Complex

Figure 6(a) overviews the PCI-Express components that we implement in `gem5`. We connect the root complex to the Membus. To ensure coherent I/O accesses, we pass all the memory requests generated by DMA transactions through an IOCache and then send them to the Membus. We do not include a host to PCI bridge (HPB) within our root complex. Instead we use the `gem5` host to PCI bridge (PCI Host). Our root complex implements three root ports and one upstream port. Each port consists of a `gem5` master and slave port. Also, there is one virtual PCI to PCI bridge (VP2P) associated with each root port. The upstream slave port is used to accept requests from the processor destined for a PCI-Express device, while the upstream master port is used to send DMA requests from the PCI-Express devices to the IOCache. As memory requests/responses in `gem5` are in the form of packets, we do not create a separate PCI-Express transaction packet in the root complex and use `gem5` packets to communicate with PCI-Express devices. Each port associated with the root complex has configurable buffers and models the congestion at the port. Also, the there is a configurable latency for request/response processing at the root complex. This latency accounts for the switching latency within the root complex. The root complex simulation object can work either with a conventional `gem5` crossbar or with a PCI-Express link that is explained later in this section.

Virtual PCI to PCI Bridge (VP2P). To implement a VP2P for a root port, we create a PCI to PCI bridge (P2P) configuration header [10], and expose this header to the enumeration software in the kernel. The enumeration software configures the necessary fields of the P2P header and assigns distinct memory and I/O space ranges to each P2P. As explained in Sec III, *PCI Host* is a dummy device in `gem5` that is responsible for configuring all the PCI devices in the system. Therefore, we need to register our VP2P model with the *PCI Host* similar to a PCI endpoint. We configure each VP2P with a bus, device and a function number. The bus number is 0 for the VP2P associated with a root port and is incremented when enumeration software traverses down the device tree in a depth-first search manner.

Figure 7 shows the 64B P2P configuration header that we implement for each VP2P. Following is a description of some important registers in the header:
– *Vendor Id, Device Id*. These two registers are needed to detect and identify a P2P. We set the *Vendor Id* of the three VP2Ps to 0x8086 and their *device ID* to 0x9c90, 0x9c92, and 0x9c94, respectively. These values are corresponding to an Intel Wildcat chipset root port configuration [24].
– *Status Register*. Indicates interrupt and error status among other features. All the bits except the 4^{th} bit are set to 0 which indicates that we have implemented the PCI-Express capability structure for this bridge.
– *Command Register*. Configures the P2P primary interface. We set the *Command register* to indicate that transaction pack-



Byte 3	Byte 2	Byte 1	Byte 0	offset
Device ID		Vendor ID		0x00
Status Register		Command Register		0x04
Class Code			Revision ID	0x08
BIST	Header Type(0x01)	Latency Timer	Cache Line Size	0x0C
Base Address 0				0x10
Base Address 1				0x14
Second Lat Timer	Sub bus num	Sec Bus Num	Pri Bus Num	0x18
Secondary Status		I/O Limit	I/O Base	0x1C
Memory Limit		Memory Base		0x20
Prefetchable Memory Limit		Prefetchable Memory Base		0x24
Prefetchable Base Upper 32 bit				0x28
Prefetchable Limit Upper 32 bit				0x2C
I/O Limit Upper 16 bits		I/O Base Upper 16 bits		0x30
Reserved			Capability pointer	0x34
Expansion ROM Base Address				0x38
Bridge Control		Interrupt Pin	Interrupt Line	0x3C

Fig. 7. PCI brdige configuration header (header type 1).

ets received on the VP2P secondary interface are forwarded to the primary interface. To enable DMA transactions from the downstream port, we configure the *Command Register* to indicate that the downstream devices can act as a requestor for the I/O or memory space transactions.

– *Header Type*. Is set to 1 that indicates this is a P2P configuration header not an endpoint header.

– *Base Address Registers*. Set to 0 to indicate that the VP2P 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 VP2P. These are configured by software and we initialize them to 0s.

– *I/O Base and Limit Registers*. These registers define the I/O address space downstream of a VP2P root port. Since the PCI I/O space range in the ARM_Vexpress_GEM5_V1 platform is between 0x2f000000 to 0x2fffffff, 32 bit addresses are needed to access the I/O space range. To implement this I/O space range, we utilize both *I/O Base Upper* and *I/O Limit Upper* registers.

– *Memory Base and Limit Registers*. These registers define a window that encompasses the memory space (MMIO) locations that are downstream of a VP2P corresponding root port. They are configured by the enumeration software and we initialize them to 0.

– *Capability Pointer*. Set to 0xD8 to indicate the starting address of the PCI Express capability structure in the configuration space.

Routing of Requests and Responses. As mentioned in Sec. II-B, PCI-Express transaction packets are routed based on the addresses in their header. We attempt to do the same in our root complex. If the packet 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 to the corresponding root port. Note that the response packet needs to be routed back to the device that had issued the request. To do this, we create a PCI bus number field in the packet class, and initialize it to -1. Whenever a slave port of the Root Complex receives a request packet, it checks the packet PCI bus number. If the PCI bus number is -1, it sets it to its secondary bus number, as configured by the enumeration software in the corresponding VP2P. The upstream root complex slave port sets the bus number 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 response packet's bus number falls within the range defined by a particular VP2P secondary and subordinate bus numbers, the response packet is forwarded out to the corresponding slave port. If no match is found, the response packet is forwarded to the upstream slave port.

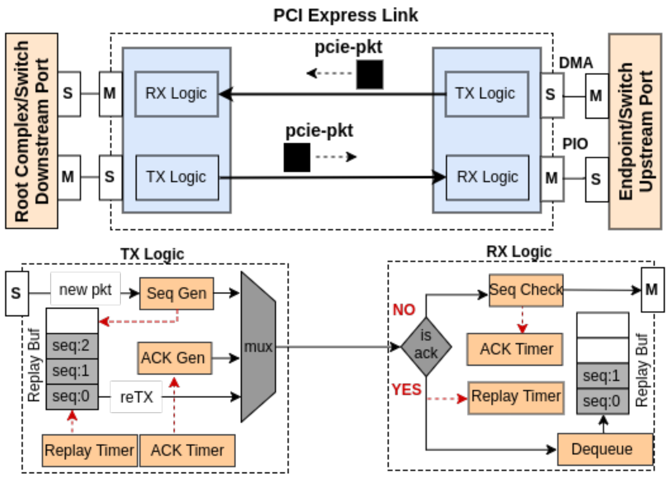


Fig. 8. Overview of our PCI-Express link model.

B. *PCI-Express Switch*

In PCI-Express, switches serve to interconnect PCI-Express links together. Similar to the Root Complex, switches perform routing of both transaction requests and response packets. PCI-Express switches can be either *store and forward* or *cut through* [25]. A store and forward switch waits 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 [25]. Since `gem5` deals with individual packets, instead of bits, our PCI-Express switch is a store and forward model. A typical PCI-Express switch present on the market has a latency of 150ns and uses cut through switching [26]. A PCI-Express switch consists of only one upstream and one or more downstream switch ports. Each switch port is represented by a VP2P. This is in contrast to the root complex, where only the downstream ports (root ports) are represented by VP2P [18].

We built our `gem5` switch model 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. We configure the PCI-Express capability structure of the switch VP2Ps to show themselves to the enumeration software as either a switch upstream or downstream port. In the root complex model, the upstream slave port accepts an address range that is the union of the address ranges programmed into the VP2Ps of each root port. In contrast, in the switch model, the upstream slave port accepts an address range based on the (I/O and memory) base and limit register values stored in the upstream VP2P.

C. *PCI Express Links*

Figure 8 illustrates our `gem5` model for a PCI-Express link. It consists of two unidirectional links, one used for transmitting packets upstream (toward the root complex), and one used for transmitting packets downstream. Each link interface consists of a master and slave port pair that can be connected to a component slave and master port, respectively. For example, the master port can be connected to a PIO port of a device or a slave port of a switch/root port, and the slave port can be connected to a DMA port of a device or a master port of a switch/root port. Each unidirectional link transmits a packet to the target interface after adding a configurable transmission and propagation latency.

PCI-Express Layered Protocol. Every PCI-Express based device needs to implement a PCI-Express interface consisting of three layers. We review these layers again in the context of our PCI-Express link model implementation:

–Transaction Layer. This layer accepts a request from the device core, and creates a TLP [10]. As explained in Sec. II-B 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 for a response TLP). In our PCI-Express link model, we use `gem5` request and response packets as TLPs and do not introduce another packet type.

–Data Link Layer. This layer sits below the transaction layer and is responsible for reliable delivery of TLPs across a single PCI-Express link. Data link layer packets (DLLPs) are used for flow control, acknowledging the transmission of TLPs, and link power management. In a physical PCI-Express protocol, a DLLP originates at the data link layer of a source device and is consumed by the data link layer of the device on the other end of the link. The data link layer also appends a Cyclic Redundancy Check (CRC) and sequence number to the TLPs before they are transmitted across the link [10]. In our PCI-Express link model, we implement a simplified data link layer. That is, each link sends ACK/NAK DLLPs to ensure reliable transmission of TLPs across a link.

–Physical Layer. The physical layer is involved in framing, encoding and serializing TLPs and DLLPs before transmitting them on the link [10]. A TLP or DLLP is first appended with STP and END control symbols [27], 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 encoded using a 8b/10b (Gen1 and Gen2) or 128b/130b (in Gen3) encoding, and is serialized and transmitted to the wire. In our PCI-Express link model, although we do not implement the physical layer, we take the encoding and physical layer framing overhead into account.

Overheads accounted for in `gem5`. We use `gem5` memory packet as TLPs. The maximum TLP payload size is 0 for a read request or a write response and is cache line size for a write request or read response. Table I summarizes the overheads that we take into account for TLPs and DLLPs.

ACK/NAK protocol. We first explain the ACK/NAK protocol implementation in a real PCI-Express data link layer, and then explain our model in `gem5`.

Sender Side: When the data link layer receives a TLP from the transaction layer, it appends a sequence number and a CRC to the TLP. A copy of the TLP is stored in a *replay buffer*, before the TLP is sent to the physical layer. 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. Replay buffer holds 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 link. This confirmation comes in the form of an ACK DLLP. On the contrary, if the receiver on the other end of the link (it is not necessarily the completer) receives a TLP with an error, it sends back a NAK DLLP across the link to the TLP sender [10]. In this scenario, the sender first removes the TLPs from the replay buffer that are acknowledged by the NAK, and then retransmits (*i.e.* replays) the remaining TLPs in the Replay Buffer in increasing sequence number order. Both the ACK and NAK DLLPs identify a TLP via its sequence number. When the TLP sender receives a ACK DLLP, all TLPs with a lower sequence number are removed from the replay buffer. A full replay buffer halts TLP transmission.

TABLE I
TRANSACTION, DATA LINK, AND PHYSICAL LAYER OVERHEADS IN OUR PCI-EXPRESS LINK MODEL.

Overhead	Type of Overhead	Packet Type
12B	TLP header	TLP
2B	sequence number appended by data link layer	TLP
4B	Link CRC appended by data link layer	TLP
2B	Framing symbols appended by Physical Layer	TLP and DLLP
8/10-128/130	Overhead caused by 8b/10b or 128b/130b encoding	TLP and DLLP

The data link layer of a PCI-Express device maintains a *replay timer* to retransmit TLPs from the replay buffer on timeouts. The timer is reset to 0 when an ACK DLLP is received. On a timeout, all TLPs from the replay buffer are retransmitted and the data link layer stops accepting packets from the transaction layer during this retransmission. The timeout mechanism is necessary when a NAK is lost or when a receiver experiences temporary errors that prevents it from sending ACKs to the sender [10].

Receiver side: On the receiver side, a TLP arriving from the physical layer is buffered in the data link layer and undergoes the CRC and sequence number checks. The receiver needs to keep track of the sequence number of the next expected TLP.If a TLP is error free, it is passed on to the transaction layer and the next expected sequence number is updated. If the TLP has an error, it is discarded and the the next expected 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 reduce the link traffic, the receiver sends back a single ACK/NAK to the sender for several processed TLPs. To do so, the receiver maintains an *ACK timer* to schedule sending Ack/NAKs back to the sender [10]. When the timer expires, an ACK/NAK has to be sent back to the sender.

The data link layer of all PCI-Express devices has the sender and receiver logic for the ACK/NAK protocol described above (TX logic and RX logic in Figure 8). The expiration period of the replay and ACK timers are set by the PCI-Express standard which is describe in the next subsection.

Our PCI-Express link model in gem5.

Figure 8 overview our PCI-Express link model. As mentioned earlier, we use the gem5 memory packets as our PCI-Express TLPs. A gem5 memory packet has the necessary information presented in a TLP header such as requestor ID, packet type (request or response), completor address, and the payload size. Each link interface receives a gem5 packet from the attached slave or master port and sends it on the assigned unidirectional link. Since we transmit both DLLPs and TLPs across the same link, we create a new wrapper class, called *pcie-pkt*, to encapsulate both DLLPs and TLPs. A sequence number is assigned to a pcie-pkt encapsulating a TLP prior to transmission. Each pcie-pkt returns a size depending on whether it encapsulates a TLP or a DLLP. The overheads such as headers, encoding schemes, and sequence number are taken into account, based on the values in Table I. The size of the pcie-pkt is used to determine the delay of each unidirectional link when transmitting a packet.

Each link interface receives a pcie-pkt from the corresponding unidirectional link. If the pcie-pkt encapsulates a TLP, then its sequence number is examined before sending it to the master or slave ports attached to the interface. If the packet is a DLLP, we perform an action based on our ACK/NAK protocol implementation.

The goal of implementing the ACK/NAK protocol is to ensure a reliable and in order transmission of packets even when the buffers of the components attached to the PCI-Express link are full. In our PCI-Express link model, the interfaces transmit TLPs as long as their replay buffer has space. Once the replay buffer is filled up due to not receiving ACKs, the packet transmission is throttled. A timeout and retransmition mechanism guarantees that TLPs eventually reach their destination when the buffers are freed up.

Each link interface maintains a replay buffer to hold transmitted pcie-pkts that encapsulate a TLP. The replay buffer is designed as a queue, and new TLPs are added to the back of the buffer. The replay buffer can hold a limited number of TLPs, hence, it can filled up quickly if a large number of small sized TLPs are transmitted in a short time period.

Each link interface maintains a "sending" and a "receiving" sequence number. The sending sequence number is assigned to each TLP transmitted by the interface and is incremented after each transmission. The receiving sequence number is used to decide whether to accept or discard a TLP that the interface receives from a link. The receiving sequence number is incremented for every successful TLP reception.

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. The replay timer is also restarted on a timeout. We set the timeout interval based on the PCI-Express specification [10] which defines the timeout interval as follows:

*(((MaxPayloadSize + TLPOverhead) / Width) * AckFactor + InternalDelay) * 3 + RxL0sAdjustment*

This time is in symbol times which is the time to transmit a single byte of data over a unidirectional link. We set the *MaxPayloadSize* in our calculation to be equal to the cacheline size, the *Width* as the number of lanes configured in the PCI-Express link, and the *InternalDelay* and *RxL0sAdjustment* as 0, since we do not take internal delay in to account and do not implement different power states, respectively. The *AckFactor* is determined based on the maximum payload size and the *Width*. The ACK timer period is set to 1/3 of the the replay timer timeout value.

After a link interface transmits a TLP on a unidirectional link, it stores the packet in the replay buffer. When the interface on the other end of the link receives the TLP, it checks to see if the sequence number is equal to the receiving sequence number. If the check passes, and the TLP is successfully sent to the connected master or slave port of the interface, the receiving sequence number is incremented and an ACK for the corresponding sequence number is sent to the sending interface, either immediately or after the ACK timer expires. If the connected master or slave ports refuse to accept the TLP, the receiving interface does not increment the receiving sequence number and the sender retransmits the packets in its replay buffer after a timeout.

When a link interface receives an ACK DLLP, it removes all the TLPs with a sequence number smaller or equal to the ACK sequence number from the replay buffer. The replay timer is restarted if any TLP remains in the replay buffer once the ACK is processed. We give priority to the pcie-pkts in the following order: (1) ACK DLLP; (2) Retransmitted pcie-pkts; (3) pcie-pkts containing TLPs received from a connected port.

VI. Evaluation

A. Methodology

Measuring the performance of a real PCI-Express link in isolation is not simple. Also, validating the full-system performance of gem5 against a corresponding real system is not feasible due to the inaccuracies in modeling the various parts of a system. Moreover, the focus of this work is only to validate the PCI-Express subsystem. To minimize the effects of inaccurate processor, memory and device modeling in gem5 on our PCI-Express subsystem performance measurements, we choose *dd*, which is a simple I/O intensive application that measures the performance of a storage device [28]. *dd* simply floods the storage device with read/write accesses. If the internal bandwidth of the storage device is higher than the PCI-Express bandwidth, then the PCI-Express becomes the bottleneck and we can estimate the performance of different PCI-Express configurations with simply running *dd*. *dd* transfers data in the units of "blocks" that is a configurable parameter. For our measurements, we only transfer a single block of data at a time, with a block size varied between 64MB and 512MB. We run *dd* with direct IO [29] to avoid the page cache lookup overhead.

To benchmark the performance of the PCI-Express in a real system, we use an Intel server with Xeon V4 E5 2660 processor and an Intel p3700 Solid State Drive as the storage

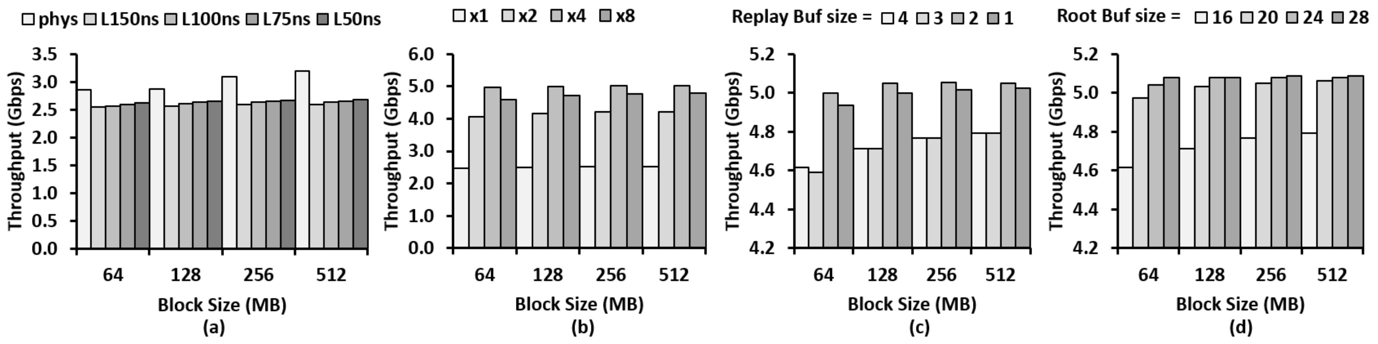


Fig. 9. Throughput of dd **(a)** when running on a physical machine and gem5 with different switch latency; **(b)** when running on gem5 with different number of PCI-Express lanes; **(c)** when running on gem5 with various replay buffer sizes; **(d)** running on gem5 with various root and switch port buffer sizes

device. The p3700 provides a sequential read bandwidth of 2800 MBps and is designed for PCI-Express Gen3 x4 [30]. The Xeon processor is connected to a X99 Platform Controller Hub (PCH) through a DMI 2.0 x4 link [31]. The DMI link provides a bandwidth similar to a PCI-Express Gen 2 X4 link, that is 20 Gbps [31]. We attach the p3700 to a PCH x1 PCI-Express slot to make the PCI-Express the bottleneck in our physical system measurements. This limits the offered PCI-Express bandwidth to 5 Gbps in each direction. Taking the 8b/10b encoding overhead into account, the p3700 read bandwidth is effectively limited to a maximum of 4 Gbps. Thus, when running *dd*, the reported bandwidth should be similar to a PCI-Express Gen 2 x1 link bandwidth.

We run gem5 with the detailed out of order processor type and tune it to model the Xeon processor as close as possible. We use the conventional gem5 IDE disk as the storage device in our evaluations. Note that the gem5 IDE disk model does not impose any bandwidth bottleneck for the data transfer (its access latency is a constant 1us value). Therefore, when we read a block from the IDE disk with *dd*, the transfer time is determined by the performance of our PCI-Express interconnect model. For validation against the physical system, we instantiate a PCI-Express switch, connect it to a root complex root port with a Gen 2 x4 link and attach the IDE disk to one of the switch downstream ports using a Gen 2 x1 link. That is to model the same PCI-Express topology as our physical setup. Unless stated otherwise, we configure our PCI-Express to model a Gen 2 interconnect. We configure the replay buffer size to 4 packets to hold enough TLP pcie-pkts until the next ACK arrives based on the ack factor [32].

B. Experimental Results

Comparing the performance of gem5 against physical setup. In this experiment, we read a single block of data with different size from the storage device into /dev/zero using *dd*. Along with varying the block size, we also sweep the switch latency in gem5 from 50 to 150ns. 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.

Figure 9(a) compares the reported throughput of *dd* when running on physical system and gem5 (*phys* and *L#switch latency* configurations in Figure 9(a), respectively). As shown in the figure, the performance of our IDE disk is within 80%$\sim$90% of the Intel p3700 SSD attached to the PCH root port, and more importantly, it follows the same trend as *phys*. Our PCI-Express interconnect shows lower *dd* throughput than the *phys* configuration. We suspect that the main culprits for such throughput mismatch are the OS overheads in gem5 for setting up the transfer and also the inaccuracies in the gem5 processor modeling. If we remove the OS overheads and make our measurements at the gem5 device level, each sector (4KB) of the IDE disk is transferred with a throughput of 3.072 Gbps over our PCI-Express link that is close to the throughput of *phys* configuration.

Another factor that reduces the bandwidth offered by the gem5 PCI-Express model is the fact that we do not support *posted write* requests. Therefore, once a sector is transmitted by the IDE disk over the link, responses for all gem5 write packets need to be obtained before the next sector can be transmitted. This is unlike the physical PCI-Express protocol where write TLPs do not need a response.

As expected, we get higher throughput across all the block size when we decrease the gem5 switch latency. *dd* throughput increases by 80 Mbps when changing the switch latency from 150ns to 50ns, across all the block size used. This throughput improvement is very minimal and accounts for $\sim$3% of the total throughput. This shows that the latency is not the only factor in determining the performance of a PCI-Express interconnect.

Comparing the I/O performance when using different PCI-Express link widths. In this experiment, we configure gem5 to model a Gen 2 PCI-Express links and vary the link widths to x2, x4, and x8. Figure 9(b) shows the results. In this experiment, we change the width of all links in the PCI-Express interconnect, including the links from the root port to the switch upstream port. We measure the throughput reported by *dd* for different block size while vary the link width.

We observe a 1.67$\times$ increase in the throughput when increasing the link width from x1 to x2. The throughput does not exactly double across all the block sizes since the OS overhead does not scale with the increase in the link width. We have a smaller increase in the throughput when doubling the link width from x2 to x4 as shown in Figure 9(b). Surprisingly, we see a drop in the *dd* throughput when doubling the link width from x4 to x8. This happens because the x8 link transmits packets too fast for the switch port to handle, causing the buffers in the switch ports to fill up. We notice that 27% of the transmitted packets experience replay when using x8 configuration, while the replay percentage for x2 and x4 configuration is almost zero. This high replay rate explains the throughput drop observed in the x8 configuration.

Comparing the performance of *dd* on a x8 link, using different replay buffer size. In this experiment, we keep the link width constant and vary the replay buffer size in each link interface. We configure the maximum number of packets a replay buffer can hold to be 1, 2, 3, and 4 and measure the reported *dd* throughput.

As shown in Figure 9(c), the *dd* throughput is considerably lower when replay buffer size is set to 3 or 4 compared with 1 or 2. We measure the number of 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 link. We observe that when the replay buffer has 3 or 4 entries, $\sim$27% of total transmitted packets experience timeout. When the replay buffer size is set to 2 and 1, 6% and 0% of the transmitted packets experience timeout, respectively.

A larger Replay Buffer ensures that more packets can be transmitted across the unidirectional link without waiting for an ACK for the previously transmitted packets. However, when the link speed is high, along with a long root complex/switch latency, the root and switch port buffers get filled up very fast and we experience timeouts and retranmissions of packets. In this scenario, source throttling the packet transmission by reducing the replay buffer considerably reduces the timeouts. This experiment shows the a complex and non intuitive behaviour of the PCI-Express interconnect while running a simple application.

Comparing the *dd* throughput when using different switch and root port buffer sizes. In this experiment, we use a x8 PCI-Express configuration, while varying the switch and root port buffer size. The replay buffer size is restored to 4, and we aim to see whether increasing the switch and root port buffers increases the *dd* throughput. Figure 9(d) shows the results.

We observe that for a particular block size, there is a large increase in the *dd* throughput when we increase the switch/root port buffer size from 16 to 20. This holds true for all the different block sizes. When the port buffer size is increased to 24 and 28, we see a minor increase in the *dd* throughput. The *dd* throughput seems to be saturated at $\sim$5.08 Gbps. This is close to 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 rate from 27% to 20%. However, we still see a huge increase in the throughput. This observation suggests that the throughput increase mainly comes from the increased space in the root complex and switch port buffers as opposed to a reduction in the timeouts. Increasing the port buffer size to 24 and 28 removes all the packet timeouts.

TABLE II
ROOT COMPLEX LATENCY VS. MMIO READ ACCESS TIME.

root complex latency(ns)	50	75	100	125	150
MMIO read access latency(ns)	318	358	398	438	517

Comparing register access latency when varying root complex latency. In this experiment, we connect a `gem5` NIC model to a root port and sweep the root complex latency from 50ns to 150ns. We measure the time taken to perform a 4 Byte MMIO read from a NIC register. We create a kernel module and measure the time taken to access a location in the NIC memory space.

Table II shows the measured latency values. As expected, we observe a decrease in the time to perform the MMIO read as the root complex latency is reduced. For every 25ns decrease in the root complex latency, we see around a 40ns decrease in the MMIO access latency. This is to be expected, since an MMIO read generates a `gem5` read request packet, along with a read response packet containing the data. Both the request and response packet latency are affected by the root complex, so the latency reduction is more than 25ns for an MMIO read.

VII. Conclusion and Future Work

In this work, we provide a PCI-Express inteconnect model that can be utilized for the future system exploration with PCI-Express based devices. Firstly, we enabled a `gem5` Linux kernel to enumerate and configure PCI-Express devices, regardless of the hardware models, for a PCI-Express interconnect. Then we created hardware models for PCI-Express components such as root complex, switches, and links to allow `gem5` to precisely model the PCI-Express subsystem. Even though we provide a generic template and performance model for future PCI-Express device developments in `gem5`, our work can be further improved in several aspects to add more detailed implementation of different PCI-Express protocol layers. We believe that our work is a strong base for more detailed PCI-Express modeling in `gem5`.

VIII. Acknowledgement

This work is supported by grants from NSF (CNS-1557244 and CNS-1705047).

References

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011.

[2] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim *et al.*, "A cloud-scale acceleration architecture," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016.

[3] R. Azimi, T. Fox, and S. Reda, "Understanding the Role of GPGPU-Accelerated SoC-Based ARM Clusters," in *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 2017.

[4] A. Mohammad, U. Darbaz, G. Dozsa, S. Diestelhorst, D. Kim, and N. S. Kim, "dist-gem5: Distributed simulation of computer clusters," in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 2017.

[5] M. Alian, D. Kim, and N. S. Kim, "pd-gem5: Simulation infrastructure for parallel/distributed computer systems," *IEEE Computer Architecture Letters*, vol. 15, 2016.

[6] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters*, vol. 14, 2015.

[7] 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.

[8] M. Jung, J. Zhang, A. Abulila, M. Kwon, N. Shahidi, J. Shalf, N. S. Kim, and M. Kandemir, "Simplessd: modeling solid state drives for holistic system simulation," *IEEE Computer Architecture Letters*, vol. 17, 2018.

[9] Intel®, "Xeon^{TM} platinum 8180m processor," https://ark.intel.com/products/120498/Intel-Xeon-Platinum-8180M-Processor-38_5M-Cache-2_50-GHz.

[10] R. Budruk, D. Anderson, and T. Shanley, *PCI express system architecture*. Addison-Wesley Professional, 2004.

[11] "Bus mastering — Wikipedia, the free encyclopedia," [Online; accessed 18-May-2018]. [Online]. Available: https://en.wikipedia.org/wiki/Bus_mastering

[12] "pci." [Online]. Available: https://www.tldp.org/LDP/tlk/tlk.html

[13] "Pci configuration space." [Online]. Available: https://en.wikipedia.org/wiki/PCI_configuration_space#Bus_enumeration

[14] "Pci local bus." [Online]. Available: https://docs.oracle.com/cd/E19455-01/805-7378/hwovr-22/index.html

[15] "Pci express packet latency matters," 2007, [Online; accessed 18-May-2018]. [Online]. Available: https://www.mindshare.com/files/resources/PLX_PCIe_Packet_Latency_Matters.pdf

[16] "Memory-mapped i/o — Wikipedia, the free encyclopedia," [Online; accessed 18-May-2018]. [Online]. Available: https://en.wikipedia.org/wiki/Memory-mapped_I/O

[17] "Message signaled interrupts — Wikipedia, the free encyclopedia," [Online; accessed 18-May-2018]. [Online]. Available: https://en.wikipedia.org/wiki/Message_Signaled_Interrupts

[18] "Pci local bus." [Online]. Available: https://webcourse.cs.technion.ac.il/236376/Spring2016/ho/WCFiles/chipset_microarch.pdf

[19] M. Sadri. Zynq training - session 02 - what is an axi interconnect? Youtube. [Online]. Available: https://www.youtube.com/watch?v=BASCRxR2L-c&list=PLfYnEbg9Uqu5q6-XcfJkMN7O0P0dwJCn7

[20] "Pci configuration space — Wikipedia, the free encyclopedia," [Online; accessed 18-May-2018]. [Online]. Available: https://en.wikipedia.org/wiki/PCI_configuration_space

[21] "Pci express switch enumeration using vmm-based designware verification ip," [Online; accessed 18-May-2018]. [Online]. Available: https://www.synopsys.com/dw/dwtb.php?a=pcie_switch_enumeration

[22] A. Rubini and J. Corbet, *Linux device drivers*. " O'Reilly Media, Inc.", 2001.

[23] *Intel 82574 GbE Controller Family*, Intel, 6 2014, rev. 3.4.

[24] "The pci id repository." [Online]. Available: https://pci-ids.ucw.cz/read/PC/8086

[25] M. Rodriguez, "Addressing latency issues in pcie." [Online]. Available: https://www.eetindia.co.in/STATIC/PDF/200909/EEIOL_2009SEP21_INTD_TA_01.pdf

[26] P. technology, "Pex8796 product brief." [Online]. Available: https://docs.broadcom.com/docs/12351860

[27] J. Ajanovic, "Pci express 3.0 overview."

[28] "Dd (unix) — Wikipedia, the free encyclopedia," [Online; accessed 24-May-2018]. [Online]. Available: https://en.wikipedia.org/wiki/Dd_(Unix)

[29] "dd: Convert and copy a file." [Online]. Available: https://www.gnu.org/software/coreutils/manual/html_node/dd-invocation.html

[30] "Intel solid-state drive dc p3700 series." [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

[31] "Intel x99." [Online]. Available: https://en.wikipedia.org/wiki/Intel_X99

[32] J. Wrinkles, "Sizing of the replay buffer in pci express devices," 2003.