Exploring Industry Standard System Integration for Hardware-Software Co-Design of PCI Express

Joonho Whangbo, Kevin Anderson, Coleman Hooper

December 16, 2022

Abstract

There is a range of integration techniques for accelerators. Close-core integration provides low-latency access time, but since the device must be integrated on-chip, it has higher design and verification costs. Conversely, far-away integration strategies allow for greater ease of integration, but have longer communication latencies. These extremes demonstrate a critical tradeoff for designers between performance and ease of integration. However, there is currently no comprehensive platform for exploring the tradeoffs with different integration techniques at the full system level. One of the many missing pieces is PCIe, which is a widely used protocol for far-away integration in industry. Despite various attempts to model PCIe, previous models do not provide both functional and timing modeling in a high-performance hardware simulation environment, and they are not integrated within a comprehensive framework for system design exploration.

In this project, we developed a validated and synthesizable high-performance PCIe model that is integrated into Chipyard [4] and FireSim [12] and can be added to a system design to evaluate device performance over PCIe. Our model includes core functionalities related to bandwidth and latency on top of a timing model. In addition, we provide a shim that allows for existing MMIO peripherals to be attached without modifications, enabling design reuse and agile design space exploration. Finally, we demonstrated the use of our platform through a case study examining how a Network Interface Controller (NIC) behaves when placed close-core and far-away. With these contributions we aim to provide a platform for design-space exploration of system-level tradeoffs for different accelerator placements.

1. Introduction

With the end of Dennard scaling and Moore's law, integrating specialized hardware accelerators into systems is crucial to achieve performance improvements. Developing these domain specific accelerators for various disciplines such as ML, genomics[8], graph traversal[10] and warehouse scale computing [11] has been a major area of interest for both industry and academia. However, there has not been enough focus on how the placement of accelerators affects the full system design, and on the analysis of optimizations along the latency and integration effort spectrum.

Currently, there are a number of ways to integrate Domain Specific Accelerators (DSAs) into systems. In industry, DSAs are attached to Network-on-Chips (NoCs) or attached as I/O devices using interconnects such as PCIe or CXL. Motivations to place devices at greater distance from the core include restricted chip area, power constraints, and capability extensions for new use cases. The high-level tradeoffs between close-core integration and far-away integration are outlined in Table 1. On the other hand, in academia, close-core accelerator integration is the de facto practice due to lack of full-system infrastructure to experiment with placing accelerators off-chip (further from the core). Accelerators are often attached to NoCs [18] or are closely-coupled by being integrated into the processor's pipeline [5]. In summary, specific companies may have undisclosed infrastructure to allow for design space exploration of different placement strategies, but there is no open-source infrastructure for exploring accelerator placements within the overall system. This exposes a deficiency in the current infrastructure; there is no design platform and corresponding OS abstractions that can be leveraged by both industry and academia to allow for design space exploration of different accelerator placement strategies.

Peripheral Component Interconnect Express (PCIe) is a widely adopted communication standard. Many accelerators are attached over PCIe as it is more cost effective and places less constraints on resource and power consumption compared to close-core integration. This presents an interesting opportunity to develop a full-stack PCIe model: a model that is synthesizable on FPGAs and runs with a Linux driver can not only provide accurate latency and bandwidth models for the PCIe protocol, but also open hardware-software co-optimization opportunities.

Hence, we propose a performance verified PCIe model **generator** written in Chisel [6] as an extension of Chipyard [4]. This model emulates core PCIe functionality by supporting the exact packet formats, virtual channels, flow control, and ACK/NACK protocol. The model's parameters are highly configurable, enabling exhaustive design space exploration of the device microarchitecture for different PCIe configurations.

To the best of our knowledge, our PCIe model is the first that can be incorporated into a cycle-exact simulation platform like Firesim [12]. With our model, system

designers can plug in their custom software abstraction layers to coordinate the transactions between general purpose processors and DSAs, enabling software/hardware co-design opportunities.

This paper is organized as follows. Section 2 provides background on the PCIe protocol stack and its message types, as well as on the hardware infrastructure used in this project and on PCIe modeling. Sections 3 and 4 explain the microarchitecture of our model and how it is integrated into an SoC. Section 5 explains how the device driver is designed. Section 6 shows validation of our model and a case study of attaching an existing Network Interface Controller (NIC) over PCIe. Section 7 provides a discussion about our results and next steps and Section 8 provides related work to contextualize our efforts.

	Proximity	Latency	Integration Costs
RoCC	Near	Low	High
MMIO	Near	Low	Medium-High
UCIe	Near-Far	Medium	Medium-High
CXL	Near-Far	Medium	Low
PCIe	Far	High	Low

Table 1: ***Simplified relative comparison between common interfaces and protocols.***

2. Background

2.1. PCIe

A critical design point within a system is the connectivity between devices. Designers are increasingly opting for PCIe becaue of its high throughput, high transfer rate, and relatively small footprint. PCIe is a high speed, point-to-point (P2P), dual simplex, serial communication protocol widely adopted in both small-scale and large-scale computing platforms. PCIe was developed from the earlier iterations of PCI and PCI-X which replaced the older ISA bus. PCIe 5.0, the latest generation currently being adopted in industry, has 64 GB/s unidirectional throughput (when utilizing 16 lanes). Common devices attached through PCIe include: graphics cards (GPUs), networking controllers (NICs), and storage devices (HDD, SSD). The specification is elaborate, therefore exact details will be omitted, and only a general overview of the architecture, structure, and core functionality relevant to the presented work will follow.

2.1.1. Architecture PCIe is a bus-based architecture with a central controller termed the root complex (synonymously known as a PCI/PCIe controller). The root complex is a single or multi-ported device orchestrating all PCIe communication on behalf of the processor and is connected to the memory bus granting DMA capabilities to the attached devices. All buses in the architecture are numbered, with the root bus, the bus directly underneath the root complex, numbered as bus 0. Bridges connect two buses together and switches connect multiple devices to a single port (internally virtual PCI-PCI bridges). Devices live on a PCI bus where information and data is sent as packets across links composed of up to 16 lanes. An example of a simple topology is shown in Figure 1.

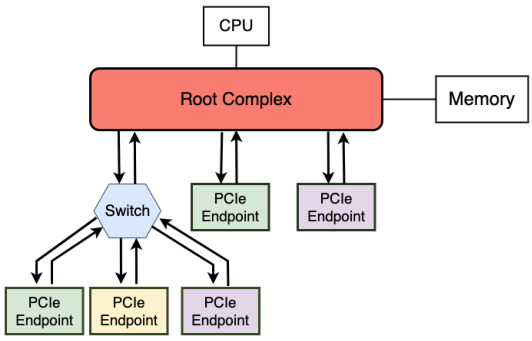

Figure 1: ***Diagram of the PCIe bus topology.***

2.1.2. Protocol Stack The specification defines a three layer protocol stack, containing a Transaction Layer (TL), a Data Link Layer (DLL), and a Physical Layer. Each device, including the root complex, follows this protocol using these layers to participate in communication. All transactions from applications originate as Transaction Layer Packet (TLPs) in the Transaction Layer, which are then passed to the Data Link Layer where additional metadata is prepended and appended, before finally being transmitted to the Physical Layer where additional metadata is prepended and appended prior to transmission. In addition to TLPs, the DLL can form and transmit Data Link Layer Packets (DLLPs), which are for control signals initiated by the DLL. The TLP and DLLP packet formats are outlined in Figure 2.

2.1.3. Configuration Space Configuration Spaces are a set of registers contained in each device, including the root complex, which define particular behavior within the PCIe topology. Each configuration space has a 64 byte header of either Type 0 (root complex or endpoints) or Type 1 (bridges). PCIe introduces 4K-byte Extended Configuration Spaces, increasing the size of the configuration spaces from 256 bytes in PCI and PCI-X. This additional space contains optional capability registers which specify functions such as power budget. Configuration registers are accessed through the legacy I/O indirect access or through memory mapped registers.

2.1.4. ACK/NACK The ACK/NACK protocol occurs with TLP transmission. On a TLP reception, the DLL of the completer transmits an ACK to the transmitting device

if no errors are detected or a NACK if errors are detected (where ACK and NACK are both DLLPs). This handshaking is used to flush stale TLPs from the replay buffer in the DLL of the requester, which holds transmitted TLPs until receiving the corresponding ACK DLLP (and retransmits the TLP in the case of a NACK). These packets consume bandwidth without transferring information.

2.1.5. Completions Completions are TLPs sent by the completer to respond to any non-posted transactions. Completion packets can return data depending on the request transaction type and are used to indicate errors to the requestor. Similar to ACK/NAK TLPs, completion packets without data consume bandwidth without transferring information.

2.1.6. Quality of Service QoS is maintained through the use of Virtual Channels (VC). VCs are separate logical communication channels with different priority that reside in the Transaction Layer. Each VC contains buffers which hold transactions (both TX and RX). A device can have a maximum of 8 VCs. Several possible VC arbitration schemes exist. For simplicity, we categorized the options as software-dependent or hardware-dependent. Software-dependent arbitration uses the configuration space to implement several distinct arbitration schemes. Hardware-dependent arbitration implements a fixed strict arbitration scheme in hardware.

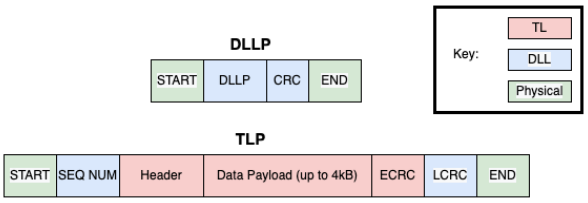

Figure 2: ***Transaction Layer Packet (TLP) and Data Link Layer Packet (DLLP) packet formats.***

2.1.7. Flow Control The flow control mechanism in PCIe is credit-based and is handled entirely by the DLL. The receive side of the TL sends the buffer size in credit units to the transmission side of the DLL. The DLL sends DLPs indicating the amount of credits that can be received. The completer of these DLLPs forwards the credit information to the transmission side of its TL. This exchange of credit information occurs both ways. Each device will check the credit information in the TL prior to sending TLPs to ensure the completer can receive the full amount of data.

2.1.8. MSI The specification requires PCIe to support either MSI or MSI-X, which are messaged-signaled interrupts that are passed as packets across the PCIe bus. However, it is not required for PCIe to support legacy interrupts (INTx).

2.1.9. DMA A noteworthy feature of PCIe is the ability for endpoints to become the bus master and execute DMA accesses to memory through the root complex. The exact mechanics of enabling this feature are beyond the scope of this section, but will be discussed at length in Section 3.

2.2. PCIe Modelling

Several PCIe models exist, however these models either model PCIe functionality or approximately match the timing of the protocol [3], [15]. The QeMU PCIe C++ model is an example of a simpler model which only mimics functional behavior of the PCIe protocol [7, 1]. In our work, we aim to model the core functionality and timing of PCIe in order to accurately capture its performance characteristics. Another disadvantage of existing models is that they are written in C++. These CPU-based simulators can bottleneck high-performance hardware simulations if the computation has to be offloaded to the core to be computed in software. In contrast, our model is synthesizable on FPGAs and runs in a cycle-exact RTL simulation to allow for much faster simulation performance.

2.3. Leveraging Infrastructure

Our work utilizes several different pieces of hardware infrastructure, and is integrated with these tools in order to provide a full-system simulation platform. Chipyard is an integrated design, simulation, and verification platform for full system-on-chips (SoCs) [4]. It is an open-source platform that contains a collection of tools and libraries to help with integration (as well as providing hardware peripherals). FireSim is an open-source platform for cycle-exact RTL simulations on cloud FPGAs. Firesim uses the Golden-Gate compiler[13] to perform RTL transformations on the target RTL to enable decoupling of the target clock and the FPGA clock. The transformed RTL is then lowered to Verilog to be synthesized and placed on a FPGA. Furthermore, it allows for scaled-out simulations across many FPGAs. Our model was integrated into Chipyard such that it can be instantiated in the same manner as any other Chipyard peripheral, and it can be used with FireSim for high-performance RTL simulations.

A CPU was generated with RocketChip, which is a parameterizable RISC-V core generator. The system bus in our design uses the TileLink procotol, which is a cache-coherent protocol for accessing the cache, memory, and memory mapped devices. In this project, we support both the lightweight and heavyweight uncached TileLink protocols. The heavyweight protocol is required for single read or write requests that span multiple cycles, which was necessary for supporting DMA for a broad range

of devices. TileLink uses Diplomacy for parameter exchange during configuration. Diplomacy creates a graph of TileLink bus connections and then handles parameter negotiation and exchange between nodes.

Spike[17], FireMarshal [16], and QeMU[7] were utilized to develop our custom Linux PCIe driver. Spike is a RISC-V ISA simulator that can boot Linux with custom MMIO hardware models. FireMarshal is a workload generation tool which can be used to configure and generate custom Linux kernels. QeMU was primarily used to exploit debugging features not available in Spike.

AWS F1 instances were used to run FireSim simulations with our synthesized design. We ran simulations with two nodes (each a copy of our SoC) on a single c4.xlarge AWS instance.

3. Model Architecture

Our PCIe model, outlined in Figure 3, consists of several principal components, which are the root complex, the timing model, and the endpoint shim. The root complex and endpoint shim each contain an instance of the protocol stack. The root complex also contains additional routing and packet conversion logic, as well as its own configuration registers. The endpoint shim contains routing and packet conversion logic, as well as additional protocol adapters which will be discussed in more detail in Section 4. The root complex is connected to the CPU and to the memory hierarchy via TileLink. The transaction flow through the model is as follows:

1. The core sends a request to the root complex over TileLink.
2. The root complex receives the requests and translates it into the appropriate PCIe packet (forming the packet through the protocol stack).
3. The constructed packet is passed to the timing model for transmission to the endpoint.
4. The timing model transmits the data to the protocol stack for the endpoint, which deconstructs the received packet.
5. The packet is then interpreted in the endpoint shim and potentially passed to one of the protocol adapters to be transmitted to the endpoint device.
6. The endpoint decides the correct operation, and if this requires a response, the endpoint will reply with a transaction that is issued by following this sequence in reverse.

3.1. Protocol Stack

The protocol stack implements all of the core operations of the PCIe three-layer protocol stack and contains an additional packetization layer. A diagram of the protocol stack is shown in Figure 4. The diagram highlights the components involved in packet construction and deconstruction on the TX and RX sides of the transaction layer,

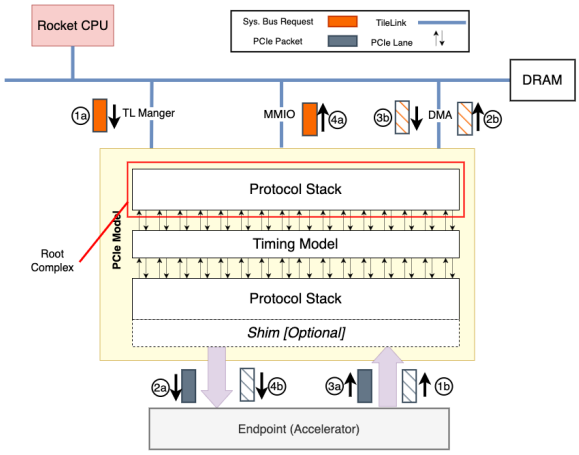

Figure 3: ***System architecture of PCIe model. The operation sequence labelled a (solid fill) shows a CPU read request and the sequence labelled b (hashed fill) shows a DMA read request.***

data link layer, and physical layer. The packetization layer constructs the TLP on the TX side and handles routing on the RX side.

3.1.1. Transaction Layer The transaction layer is responsible for staging packets for transmission as well as receiving packets and buffering them until the device is ready to receive them. The transaction layer contains a parameterizable number of virtual channels. To be compliant with the specification, flow control is implemented in the transaction layer. Within each virtual channel, there is a TX buffer for staging packets (waiting for corresponding packet limits) as well as TX counters for tracking the current credit limits for each packet type. There are also three header and three payload RX buffers for receiving packets of different types (posted, nonposted, and completion). The RX counters keep track of how many packet chunks have been dequeued from the RX buffers in order to send credit limit updates in the form of flow control packets.

3.1.2. Data Link Layer Our model supports two types of data link layer packets (DLLPs). The first type is acknowledgement packets, which must be sent whenever a full packet is received at the other end of the P2P connection. The second type is flow control packets, which send updated credit limits when packets are consumed. The data link layer and transaction layer together arbitrate between DLLPs and TLPs to ensure that entire packets are transmitted before starting to transmit a packet of a different type.

3.1.3. Physical Layer The physical layer is responsible for adding start and end bytes, as well as serializing the message to transmit it across the physical lanes. It consists of a transmitter module (which serializes the incoming

PCIe packet chunks) and a receiver module which deserializes the incoming bits. Each physical lane is composed of multiple logical lanes. This is to ensure we maintain the same bandwidth as actual PCIe bus while avoiding using a separate clock domain, since this would require slowing down the rest of our system in FPGA simulations.

The physical layer communicates across our timing model which consists of hardware queues. To model latency, a group of bits that are sent simultaneously over the physical lanes are given a timing token when enqueued, which represents the cycle in which they can be dequeued. Hence, a group of bits is dequeued only when the token is greater than or equal to the current cycle. Validation of the PCIe latency and bandwidth is presented in Section 6.1.

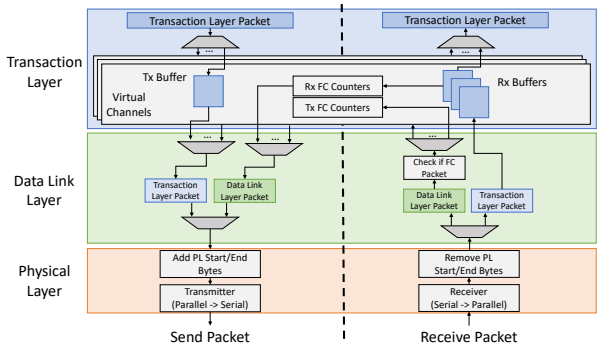

Figure 4: ***Diagram of the PCIe Protocol Stack (excluding the packetization layer).***

3.1.4. Packetization Layer The packetization layer sits on top of the PCIe protocol stack and is responsible for translating messages coming in from Tilelink to PCIe packets and vice versa. It also handles routing for incoming PCIe packets, which is performed based on packet type since different message types will go to the CPU and to memory; however, this could be extended to support address-based routing (if we wanted to be able to model a switch). Note that the endpoint shim also contains a packetization layer to interpret incoming PCIe messages and perform routing.

3.2. Root Complex

The root complex includes an instantiation of the protocol stack. Furthermore, there is a configuration space to provide a compliant interface for Linux drivers. There are four separate connections between the root complex and the system bus; one to configure the root complex (MMIO Node), one to handle CPU communications with the PCIe endpoint (Endpoint Manager Node), and two that allow the root complex to initiate DMA requests (to read from or write to memory on behalf of the endpoint).

3.2.1. MMIO Node The MMIO node is responsible for controlling the configuration registers of the root complex. Mixing in the **HasRegMap** trait with the **PCIeRootComplexMMIORegs** module allows us to instantiate memory mapped registers which are connected to the Tilelink **a** and **d** channels. Additionally, by making the **PCIeRootComplexMMIO** module extend the **TLRegisterRouter** module and also contain a **SimpleDevice** object, we can add the MMIO register addresses into the device tree source (DTS) so that the driver can recognize it.

3.2.2. Endpoint Manager Node The Endpoint Manager Node is a memory mapped region representing access to the endpoint. Requests at the Endpoint Manager send out root complex internal messages that are translated into configuration read or write PCIe packets in the packetization layer. When the packet arrives at the endpoint, it causes the endpoint to perform certain actions by writing to the configuration registers.

3.2.3. DMA Node The DMA node supports memory read and write requests on behalf of the endpoint. When the root complex receives a memory read or write packet from the endpoint, it routes the corresponding internal message to the **PCIeRootComplexDMA** module which forwards the message to the **RootComplexDMAReaderControl** module or the **RootComplexDMAWriterControl** module according to the type of the transaction. The control modules send the base address and length of the DMA request to either the **MemLoader** module or the **MemWriter** module, both of which chunk up the DMA requests into transaction sizes that are compatible with Tilelink. They both send memory requests through the **RootComplexDMAHelper** module which makes sure that the responses from Tilelink are reordered such that they match the request order.

3.3. Endpoint Compatibility

An instantiation of the protocol stack follows the timing model to ease compatibility concerns. Devices can connect directly to this stack, or they can have a shim to convert the data into an expected format. To support a wide range of existing accelerators, an optional MMIO-shim was created to make the model compatible with existing MMIO accelerators. The shim latency is assumed to be negligible relative to the PCIe transaction latency. The shim converts PCIe packets into TileLink messages so that MMIO devices in Chipyard can be plugged directly into our model without changes. Figure 3 shows a conceptual diagram of the shim which can be used as the basis for developing a shim for another protocol (eg. RoCC).

4. Chipyard Integration

Ease of use was a design goal, so the model was designed to be integrated into Chipyard. It is easy to instantiate our model in different systems and to connect different MMIO-based peripherals to our model without modifying the interface. The root complex is connected to the system bus through several TileLink nodes following the methodology of Chipyard (as outlined in Figure 3). The MMIO node provides access to the internal root complex configuration registers, as well as the ability to send interrupts to the RocketChip CPU. The Endpoint manager node allows for configuration reads and writes to be sent to the endpoint (through our PCIe model). The DMA client node facilitates DMA on behalf of the endpoint (which sends memory read and write packets through our PCIe model).

We developed two protocol adapters; one for a PCIe client and TileLink manager (called **PCIeToTL**), and one for a TileLink client and PCIe manager (called **TLToPCIe**). The **PCIeToTL** adapter converted PCIe configuration reads and writes into corresponding MMIO reads and writes. The **TLToPCIe** adapter converted TileLink read and write requests into PCIe packets. Both adapters had to handle both request and response messages. The **PCIeToTL** adapter created TileLink requests and handled responses, whereas the **TLToPCIe** adapter handled TileLink requests and created corresponding response messages. The **TLToPCIe** adapter had to be configured to support multi-cycle memory writes and read completions to handle requests that were sent through our PCIe model to the DMA module. The **TLToPCIe** adapter also had to track source IDs to match completions when they were returned to the adapter (as it allowed for multiple in-flight read and write requests).

The high-level structure of our protocol adapters and endpoint shim is outlined in Figure 5. The endpoint shim arbitrates between the manager and client node (as well as an interrupt line) and provides routing back to these modules. When interrupts are asserted, the shim translates them into corresponding MSI packets and sends them through our PCIe model to the MMIO module.

One complication with translating requests between the TileLink and PCIe protocols is that unlike our PCIe model (which enforces strict ordering), TileLink allows for out-of-order message delivery. This complicated the design of our DMA module, which needed to be able to support message reordering. Note that this doesn't impact the multi-cycle requests between the client node on the MMIO peripheral and the manager node on our protocol adapter, as these are connected using a simple crossbar without internal arbitration (which ends up being instantiated as a wire).

To manage the top-level design, we used Chipyard configurations in order to manage the different SoC components that were added. These include CPU configuration, attaching our PCIe model to the main system bus, as well as additional configuration for address spaces and memory. Additionally, during our case study in Section 6.2, we had to develop configurations that also managed the placement of the Network Interface Controller (NIC) (since it was attached to the TileLink crossbars for our protocol adapter) as well as configuring the IO for the NIC to punch out to connect to a loopback device. To be able to run our design on AWS with FireSim, we had to develop separate configurations to include additional commands required for launching FPGA-based simulations.

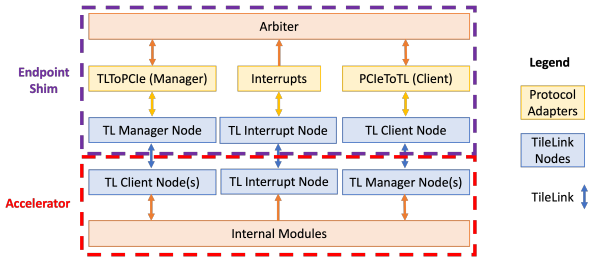

Figure 5: ***Architecture of the endpoint shim and protocol adapters for converting TileLink requests and responses from the MMIO endpoint into PCIe packets. The endpoint shim contains a protocol adapter with a client node to connect to the endpoint's manager node(for handling control requests from the CPU), as well as a protocol adapter with a manager node to handle DMA requests from the client node. The shim also contains interrupt connections and is able to convert TileLink interrupts into MSI packets.***

5. Driver

Bare-metal applications are sufficient for simple, but limited, tests. As previously mentioned, devices connected across PCIe are typically complex devices which are not conducive to bare metal applications. To enable hardware/software co-optimization and identify bottlenecks in software, we have been developing a custom Linux driver to interface with our PCIe model.

As this is a model, the driver avoids conventional configuration space checks and operations related to enumeration. However, the driver utilizes the Linux PCI subsystem and adheres to the Linux device driver model. The advantage of such a driver is that existing code can run with our model without adaption. This reduces the effort on the software side for existing devices or novel devices compatible with existing device software. Development for the driver was performed in a combination of Spike (RISC-V ISA-SIM), FireMarshal and QEMU.

In addition, the IceNIC device driver was adapted to communicate over our PCIe MMIO interface. The modified driver allowed for control over the NIC without major revision to the design. Since the PCIe driver exposes a

simple memory read and write interface to the endpoint driver, adapting the endpoint device driver to work directly with the PCIe root complex has neglible impact on the software overheads. Being able to reuse existing MMIO drivers has clear advantages as it allows for software reuse. However, completing the development of the subsystem-integrated PCIe driver will still be important for being able to precisely estimate software overheads for PCIe devices.

6. Evaluation

Our evaluation consisted of two phases. The first phase aimed to verify the functional correctness of our model and validate its performance characteristics. Additionally, in this phase we assessed the area of the proposed module when synthesized on an FPGA. In the second phase, we pursued a case study to show both the ease of integration with existing MMIO-based peripherals in Chipyard, as well as to show how our platform can be used for full-stack analysis.

6.1. Model Validation

In order to verify our model and fully exercise different features during testing, we used a fake PCIe endpoint that connected directly to our PCIe model (in place of the endpoint shim). This model was able to exercise different functionality (for example, triggering different DMA read and write access patterns for bandwidth analysis). We maintained separate testbenches for exercising different functionalities (for example, DMA reads and writes, MMIO configuration, and interrupt signaling).

To validate the behavior of our model, we first set the timing parameters of our model by matching the delay parameters for our physical layer with the measured round-trip latency for a one-dataword packet (which was roughly one microsecond for the 8-lane PCIe 3.0 configuration) [15]. After matching the latency for a single dataword for PCIe 3.0, we then explored both how the number of lanes affects the model latency, as well as how the end-to-end latency is affected by the packet size. Figure 6 shows the measured round-trip read latency for different numbers of PCIe lanes and different packet sizes. Compared to the measured results for the 8 lane configuration, our model exhibited similar scaling, although our latency scaled slightly faster with packet size compared to the measured results, which went from roughly 1 to 1.25 microseconds as the payload increased from 1 to 512 bytes (whereas our 8-lane configuration exhibited a latency of 1 to 1.35 microseconds). This could be the result of differences in root complex design between our model and the device used in the paper (as the root complex design is not standardized).

Figure 7 showed the effective and total bandwidth for our model. For this evaluation, we sent repeated DMA writes from our fake endpoint to fully saturate the PCIe lanes. We modelled the PCIe 3.0 configuration to mimic the evaluation in [15]. Note that we used 16 lanes for our bandwidth experiments, and so we observed twice the bandwidth observed in [15] when they used an 8-lane configuration (note that the plot in [15] uses bits rather than bytes transferred). The effective bandwidth is the bandwidth from only the payload for each packet, and the total bandwidth includes the additional information such as headers and ECRC bytes for each transaction layer packet, as well as the required DLLPs for flow control and ACK/NACK.

The model exhibits a saw-tooth pattern as observed in [15] since at each 256-byte boundary, an additional transmission is required for one additional byte (with a separate header and separate ACK/NACK response). Note that the maximum per-lane bandwidth observed in our model was slightly higher than in [15]. This is likely because we used large RX buffers (which limited the number of flow control transmissions that needed to be sent); future investigation into RX buffer sizing could help us model different root complex and endpoint implementations.

We limited our endpoint to 256-byte payloads to mimic the evaluation in [15], which used a device with a maximum payload size of 256 bytes. The maximum acheivable PCIe payload size is 4096 bytes so another device could transmit more per payload, but it would still exhibit this same pattern (just with the sawtooth pattern occurring at 4096-byte boundaries).

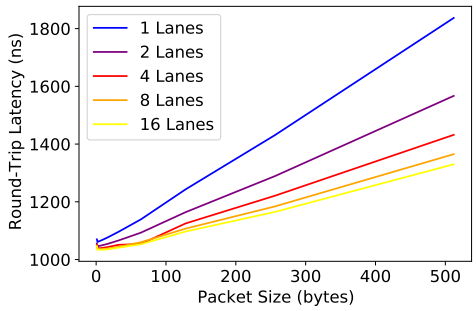

Figure 6: ***Plot of our model latency versus number of PCIe lanes.***

Table 2 shows the post-opt utilization of our design incorporating the IceNIC attached over PCIe. We found that the area of our PCIe model was using **3.39**% of the total available LUTs on an AWS F1 4x Large instance while the in-order Rocket Core was using **0.82**%. This highlights the complexity of our PCIe model.

6.2. Case Study - Network Interface Controller

We pursued a case study to demonstrate the benefits of our platform for analyzing the performance characteristics

	Total LUTs	Logic LUTs	LUTRAMs	FFs	RAMB36	DSP Blocks
Digital Top	81505(9.10%)	65451(7.31%)	16054(3.56%)	28508(1.59%)	122(7.26%)	15(0.27%)
Rocket	7351(0.82%)	7271(0.81%)	80(0.02%)	2111(0.12%)	0(0.00%)	4(0.07%)
PCIe	30376(3.39%)	16774(1.87%)	13602(3.02%)	5414(0.30%)	0(0.00%)	0(0.00%)

Table 2: ***Resource utilization on a Xilinx Virtex Ultrascale+ (xcvu9p) for a system with an SoC and our model.***

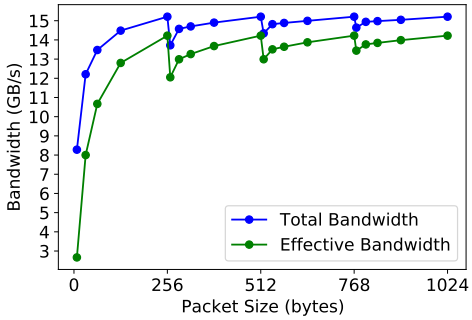

Figure 7: ***Plot of the effective and realized bandwidth of our model (for the 16-lane PCIe 3.0 configuration).***

of a device when placed over PCIe versus close-core integration. We tested the performance of a Network Interface Controller (NIC) when connected over MMIO versus when connected over PCIe. NICs are commonly connected over PCIe, but can also be integrated close-core for latency-sensitive applications [12]. The NIC we used was IceNIC, which is a Chipyard peripheral for networking [12]. The integration of IceNIC over PCIe and over MMIO is outlined in Figure 8. IceNIC uses separate TileLink client nodes for the read and receive paths, as well as a manager node for receiving control signals and sending interrupt signals. After receiving a send signal from the CPU, the NIC's send path reads data from the desired memory address and then sends it out on the network. When receiving a packet, the NIC will use an address sent by the CPU to store the received data in memory.

To connect IceNIC to our PCIe model, we used the endpoint shim with two protocol adapters instantiated (a manager adapter for handling DMA requests from both the send and receive paths and a client adapter to handle control messages sent to the NIC). The client adapter translated PCIe read and write packets to TileLink read and write requests and converted responses to read completion PCIe packets to be sent back to the CPU. The manager adapter converted TileLink requests to read and write PCIe packets and converted PCIe completion packets to TileLink responses.

Figure 9 shows the performance of the NIC when placed over PCIe with different physical layer latency parameters for a bare-metal latency test with a loopback NIC attached as a endpoint. PCIe-X represents that it takes X-cycles to cross the physical layer. As expected, the NIC experiences a much greater latency penalty when

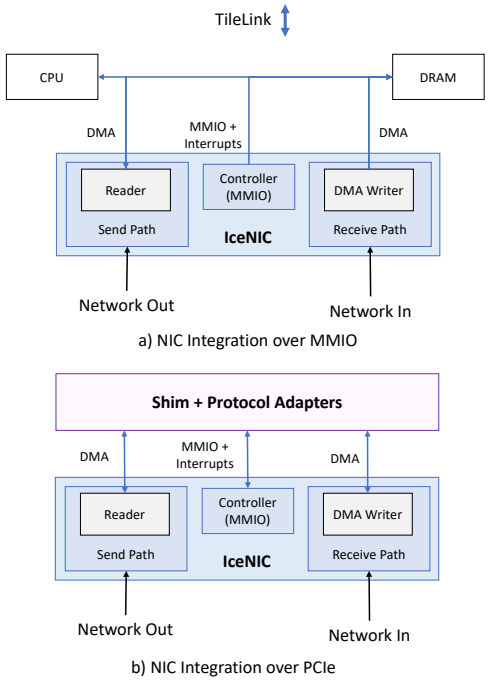

Figure 8: ***Diagram of the IceNIC network interface controller, outlining how it is integrated over MMIO and over PCIe.***

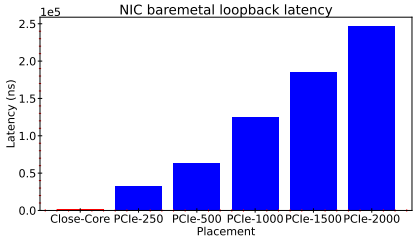

Figure 9: ***Plot of the NIC latency on a bare-metal benchmark when attached over MMIO and when attached over PCIe for different PCIe bus latencies.***

attached over PCIe. Figure 10 demonstrates the performance of the loopback NIC on a bare-metal latency test with different numbers of PCIe lanes. These results show that increasing the number of lanes has no impact on performance (since the NIC is latency-bound for this benchmark), and also that the NIC is idle waiting for control signals almost 50% of the time.

A significant amount of the slowdown when the NIC is attached over PCIe is due to control overhead. This over-

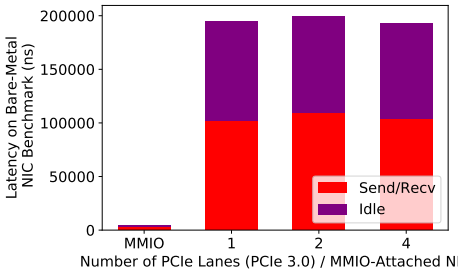

Figure 10: ***Plot of the NIC latency on a bare-metal benchmark when attached over MMIO and when attached over PCIe for different lane configurations (using 500-cycle physical layer latency). The breakdown highlights the amount of time that the NIC has outbound memory requests for the Send and Receive paths, and the amount of time that it is idle waiting for control signals.***

head is because the NIC is configured to have the CPU transmit send and receive addresses for each packet and to then wait for the CPU to accept send and receive completions for each packet. Even for the time that the send or receive path is active, it is still underutilized since requests aren't batched. In [15], they outline several potential optimizations that can be implemented to avoid having the NIC control signals become the bottleneck when attaching a NIC over PCIe. Given additional time, one optimization we could have pursued and tested would have been to enqueue and dequeue send and read addresses and completions in batches to show how our platform allows for system-level optimizations to be explored. A second optimization is to increase the maximum number of outbound requests that the NIC can support at one time in order to saturate the PCIe model bandwidth.

7. Discussion

7.1. Results

The model validation experiments successfully demonstrated two critical design points motivating the project: (1) a root complex model incorporating both functional and timing models, and (2) validation of our model using published performance metrics. Our model's latency and bandwidth characteristics matched well with measured PCIe characteristics [15]. It is important to note that the exact performance characteristics over PCIe can be root complex-dependent, as the root complex implementation isn't standardized. We were then able to use our tool to both assess NIC placement and then to further analyze bottlenecks in latency when the device is placed over PCIe. In the future, we plan to use this tool to pursue and test further optimizations.

Although the model was extensively tested in VCS and metasim, despite significant effort we were not able to accomplish our goal of full-stack simulation on FPGAs with FireSim. This was in part because the FireSim linux networking simulations required multinode configurations, and the simulation setup was highly sensitive to the exact placement of the NIC in the design. Several time-consuming setbacks with configuring and launching jobs on AWS EC2 instances prevented running the intended benchmarks and analysis prior to the deadline. Running these tests on a standard RTL simulation tool like VCS would have taken on the order of several days and therefore was not a viable option to produce results. The benchmark tests basically provide a stress test for our model by constantly keeping the NIC active. We intended to show two simulations using this benchmark: (1) the NIC close core, and (2) the NIC connected through the PCIe model. Our hypothesis was that the control packets between the IceNIC and CPU added non-negligible overhead when the device was connected over PCIe, which would in turn reduce bandwidth. One optimization to address this would be to alter the IceNIC to buffer non-urgent control signals and send them as one TLP, instead of multiple smaller TLPs. With this optimization, the NIC could be attached as a PCIe device using our model with comparable performance (for non-latency sensitive workloads). We also planned to investigate whether setting the PCIe bandwidth too low can throttle the NIC performance, preventing it from saturating the network connection.

7.2. Linux Driver

The Linux driver was eventually deferred for future work. The default Linux kernel generated by FireMarshall was modified to support the PCI subsystem and our custom driver. Additionally, we created a DTB containing a node referencing our model within our design. Matching occurred and the device appeared in *sysfs*, however our model failed probing during boot. We determined the issue was due to the lack of enumeration once the kernel attempts to scan the root bus. After this realization, the Linux driver became out of scope for this project, but still important for future work.

7.3. Takeaways and Lessons Learned

This project was very ambitious and several lessons were learned in overcoming hurdles. Over the course of the implementation, the project was rescoped from the original proposal of a large framework for DSE with accelerators to focusing on supporting PCIe in academia infrastructure projects. As a result, even though setbacks with AWS prevented us from attaining the benchmark results we ultimately targeted, we have established a validated first implementation of the model to expand upon. A general takeaway from this project is to scope infrastructure-related projects appropriately (accounting not just for design time for our module, but also for integration and setup overheads with existing infrastructure).

One lesson learned was the complexity with configuring the PCI network in Linux. To configure and initialize a PCIe bus on boot requires both configuration software and firmware within the root complex. The configuration software can be largely replaced by functions with the Linux PCI subsystem, but even a reduced version of the root complex firmware is complicated to model as it must support complex functions like enumeration. Given the timeline for the project, efforts dedicated to this were forwent. However, missing such firmware complicated using the PCI subsystem. Ultimately, the general Linux driver for the root complex was relegated to future work, and we opted to pursue the modified IceNIC driver instead.

Another lesson learned was to plan to discard the first implementation of a system. Designing within the Chipyard environment simplified tedious tasks like connecting to the system bus, but complicated other aspects like maintaining several TileLink Nodes per desired function and adhering to Diplomacy. The core Chisel was rewritten twice as conflicts were discovered in the underlying infrastructure and intended design goal.

7.4. Future Work

There are several avenues for expanding on this work. The first avenue is to develop additional protocol adapters to convert between RoCC commands and MMIO commands (which could then be converted into PCIe packets). Note that we could not develop these adapters before the PCIe to MMIO and MMIO to PCIe protocol adapters as they relied on using the MMIO bus (meaning that we would be going from PCIe to MMIO to RoCC or from RoCC to MMIO to PCIe). This would allow for an even greater number of open-sourced accelerators to be supported by our platform.

Along with the PCIe to RoCC protocol adapter, we are planning to build IOMMUs so that accelerators attached over PCIe can access the virtual address space. This feature is crucial to enable hardware-software codesign for accelerators.

Additionally, there are multiple design space exploration case studies that we plan to pursue. One accelerator we plan to analyze placements for is a protobuf accelerator. Existing work already details possible software optimizations such that the inevitable increased latency can be cleverly hidden, and our tool provides a platform to validate whether these optimizations can be performed [11]. However, attaching the protobuf accelerator would require protocol adapters for RoCC.

A second case study we plan to pursue is with Gemmini, which is a machine learning accelerator generator [9]. We plan to investigate how the memory hierarchy is affected by changes in placement for different types of models (which have different memory bandwidth requirements in order to be able to fully utilize the compute units). However, this case study is also dependent on first developing protocol adapters for RoCC.

Lastly, the Linux driver for the root complex is a high priority goal. As previously mentioned, an accompanying driver integrates into the Linux driver model, such that device drivers can sit on top of our root complex driver and communicate as if the model was a true root complex. Furthermore, existing benchmark and applications could run without modification. This is seen as a greater benefit for designers wanting to test prototypes of newer generations of devices in industry, or academics leveraging an established code base for hardware/software co-design.

8. Related Work

There have been many attempts to cycle-accurately model PCIe performance in **software** simulators[3]. However, software simulators are slow relative to FPGA-based simulations and do not scale out to many nodes. In contrast, our PCIe model is FPGA synthesizable and can run in a cycle-exact manner on Firesim [12] which can support up to 1024 simulation nodes. Furthermore, as our model is a hardware generator, it maintains the high configurability of software simulators.

Examples of RTL implementations or IP for root complexes exist in both academia and industry, however they have limitations [14, 2]. The paramount drawback is that these implementations are solely functional, and therefore timing analysis for particular scenarios cannot be assessed unless the scenario is replicated. Additionally, these studies only provide a root complex implementation, and other components which compose the actual system will still need to be acquired and integrated. In contrast, our model is inherently integrated in an SoC generation ecosystem providing all requisite infrastructure and additional components to emulate more complex SoCs. Lastly, RTL root complex implementations typically target FPGAs which require additional effort in verifying electrical connections as PCIe devices will be external, introducing another failure mode which complicates debugging novel designs.

Other works presented analytical models of PCIe performance and compared it to the performance of PCIe 3.0 by running networking benchmarks [15]. However, unlike our model that provides insight into PCIe performance throughout the full compute stack, their work only models the PCIe hardware performance.

9. Acknowledgements

Throughout the design and implementation phases of the project, several members of SLICE provided advice about orienting the project and shared their knowledge of the Chipyard infrastructure. We would like to acknowledge these collaborators and the SLICE lab as a whole for

their contribution. Lastly, we would like to thank Kubi for offering this course and for providing direction and feedback for the project; we discovered viable research topics through this class project and are looking forward to building on this work.

References

[1] QEMU Documentation.

[2] UltraScale+ Devices Integrated Block for PCI Express v1.3.

[3] Mohammad Alian, Krishna Parasuram Srinivasan, and Nam Sung Kim. Simulating pci-express interconnect for future system exploration. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 168–178, 2018.

[4] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020.

[5] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[8] Daichi Fujiki, Arun Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. Genax: A genome sequencing accelerator. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 69–82, 2018.

[9] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration, 2019.

[10] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[11] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A hardware accelerator for protocol buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 462–478, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42, 2018.

[13] Albert Magyar, David Biancolin, John Koenig, Sanjit Seshia, Jonathan Bachrach, and Krste Asanović. Golden gate: Bridging the resource-efficiency gap between asics and fpga prototypes. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[14] Rekha B Manjunath. Implementation of pci express architecture, 2022.

[15] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[16] Nathan Pemberton. Firemarshal, Feb 2022.

[17] Andrew Waterman. riscv-isa-sim, Dec 2021.

[18] J. Zhao, A. Agrawal, B. Nikolic, and K. Asanovic. Constellation: An open-source soc-capable noc generator. In *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*, pages 1–7, Los Alamitos, CA, USA, oct 2022. IEEE Computer Society.