High-Performance FPGA-accelerated Chiplet Modeling

Xingyu Li

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-145

http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-145.html

May 19, 2022

Copyright © 2022, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I want to especially thank Sagar Karandikar for mentoring me on different projects during both my undergraduate and graduate research experiences at ADEPT/SLICE Lab. He guided me through various tools used in the lab and provided abundant academic and career development suggestions for me. I also want to thank Jerry for helping throughout my Master program, especially on the NoC system and FireSim simulation topics. Additionally, I would like to thank Tushar Sondhi for working with me on this project, as well as a previous NoC Compression project. Finally, I want to thank Professor Krste Asanovic for advising me throughout this year. Thanks to all those listed here and others for supporting me.

High Performance FPGA-accelerated Chiplet Modeling

by Xingyu Li

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Krste Asanovic
Research Advisor

5/19/2022

(Date)

* * * * * * *

Professor Yakun Sophia Shao
Second Reader

5/17/2022

(Date)

High-Performance FPGA-accelerated Chiplet Modeling

by

Xingyu Li

A technical report submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Krste Asanović, Chair
Yakun Sophia Shao

Spring 2022

The technical report of Xingyu Li, titled High-Performance FPGA-accelerated Chiplet Modeling, is approved:

Chair ______________________________ Date ____________

______________________________ Date ____________

______________________________ Date ____________

University of California, Berkeley

High-Performance FPGA-accelerated Chiplet Modeling

Copyright 2022
by
Xingyu Li

Abstract

High-Performance FPGA-accelerated Chiplet Modeling

by

Xingyu Li

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Krste Asanović, Chair

With the advent of 2.5D and 3D packaging, there has been increasing interest in chiplet architectures, which provide a cost-effective solution for large-scale systems. Chiplets reduce fabrication cost via yield improvement and also provide an opportunity to conveniently incorporate accelerators to existing systems. Currently, there are existing performance models for the RoCC near-core interface in FireSim, and PCIe is an old standard that is very well explored. However, the latency of chiplet interfaces is between that of the two aforementioned technologies, and is not well studied yet. As such, it has become increasingly important to productively and accurately model performance and latency of chiplet interconnects. Hence, this project aims to support high-performance chiplet connection and system modeling in FireSim, an FPGA-accelerated hardware simulation system, which will enable further studies on both hardware and software systems management for chiplet systems.

Contents

Contents **i**

List of Figures **iii**

List of Tables **iv**

1 Introduction **1**

2 Background and Related Works **3**
2.1 Chiplet . . . 3
2.2 Network-on-Chip . . . 5
2.3 FireSim and Other Tools . . . 5
2.4 Related Works . . . 6

3 Design **8**
3.1 TileLink Protocol . . . 8
3.2 FireSim Target-to-Host Bridge . . . 10
3.3 Overview of the Design . . . 10
3.4 Hardware Bridge Module . . . 11
3.5 Software Bridge Driver . . . 14
3.6 Software Switch . . . 15

4 Testing and Evaluation **17**
4.1 Experiment Environment Options . . . 17
4.2 Designs for Tests . . . 18
4.3 Testing . . . 19
4.4 Prototype for Profiling and Evaluation . . . 21
4.5 Performance Evaluation . . . 22

5 Ongoing and Future Works **25**
5.1 Full Integration in FireSim with Real Systems . . . 25
5.2 Other Future Works . . . 27

6 Conclusion **28**

Bibliography **29**

List of Figures

2.1 This is an comparison between a SoC on a monolithic chip and the same design with the chiplet architecture. The green block represents the scope of the chip and components on the same white tile are on the same die. 4

3.1 The channel architecture of a TileLink Link between a pair of agents.[14] 9
3.2 Simulating a 4-chiplet system with 4 F1 instances and a Xeon Host CPU in AWS 11
3.3 The architecture of the hardware bridge modules. 13
3.4 This figures demonstrates the data flow through the architecture of software bridge drivers and the switch on the host CPU for a single direction of a link. (The output buffers of the input side driver and the input buffers of the output side driver exist but are not shown in this diagram to save space.) 15

4.1 The TileLink Bundle Fuzzer designed for generating `TLBundleIO` inputs when testing and prototyping the `ChipletNICBridge`. 18
4.2 The architecture of one testing design used in Hardware Meta-Simulation 20
4.3 The diagram of the software driver loop-back test. 20
4.4 The prototype system used for profiling and evaluation 21
4.5 The first token received time in cycles in both the simulation clock and the actual FPGA platform clock. 23
4.6 Runtime of transferring 10,000 tokens via the bridge in both simulation clock and the actual FPGA platform clock. 23
4.7 Simulation frequency of different chiplet latency vs the platform frequency. . . . 24

5.1 The original simple CPU-DRAM system on the left, and the chiplet connection version of the same system. 25

List of Tables

4.1 A brief summary of three different kinds of simulation utilized in this project . . 17
4.2 Calculated simulation frequency of different chiplet latency. 24

Acknowledgments

I want to especially thank Sagar Karandikar for mentoring me on different projects during both my undergraduate and graduate research experiences at ADEPT/SLICE Lab. He guided me through various tools used in the lab and provided abundant academic and career development suggestions for me. I also want to thank Jerry for helping throughout my Master program, especially on the NoC system and FireSim simulation topics. Additionally, I would like to thank Tushar Sondhi for working with me on this project, as well as a previous NoC Compression project. Finally, I want to thank Professor Krste Asanovic for advising me throughout this year. Thanks to all those listed here and others for supporting me.

Chapter 1

Introduction

A chiplet is an integrated circuit block that is designed to work with other chiplets to form a larger and more complex chip. Instead of manufacturing an entire system on a single piece of silicon, chiplets allows the semiconductor fabrication plants to to use multiple smaller chips to make up a larger integrated circuit with fewer costs. On the other hand, chiplets also ease the process of incorporating extra accelerators to existing systems, such as adding new workload-specific accelerators in the data-centers.

However, chiplet modeling has not been fully studied in the academia and there is not sufficient open-source support in modelling large chiplet systems. FireSim is an easy-to use, FPGA-accelerated cycle-accurate hardware Simulation in the Cloud and it is also widely used in many projects in the SLICE Lab. In FireSim, there is no direct models for chiplet interconnect latency as well. Previously, there were infrastructure for modeling RoCC-standard accelerators attaching to a system, which has the latency across components on the same chip, and PCIe connection with Ethernet protocols for simulating networks of multiple chips. This project aims to build a bridge across FPGAs with relatively small latency to support chiplet modeling in FireSim, with a parameterized latency of the connection to reflect different physical placement of the chiplet systems. This long-term goal of this project is to create a general interface on which one can model the chiplet latency to gain knowledge not only on hardware, but also on programming language and OS management paradigms that are effective under chiplet conditions.

This projects prioritizes on first building a chiplet-modeling bridge for the Constellation Network-on-Chip(NoC) system using TileLink protocol. There are mainly two reasons why I choose it as the beginning step: first, NoC systems can be so large and complicated that they would benefit from dividing the entire systems into chiplets; secondly, large NoC may not be able to be compiled and fit into a single F1 instance used in FireSim and, therefore, building the bridge will also be important for simulating large NoC systems.

The remainder of the paper is organized as follows. In Chapter 2, it provides a brief overview on the background of chiplet, FireSim and other tools, as well as the related works. The report then discusses some directly related design contexts, and illustrates the design of the chiplet modeling bridge in Chapter 3, including both hardware and software implemen-

tations. The experimental environments, testing, and evaluation of the design is presented in Chapter 4. Finally, there are some discussions on the ongoing and future works in Chapter 5, as well as the conclusion of the paper in Chapter 6.

Chapter 2

Background and Related Works

2.1 Chiplet

A chiplet is a small integrated circuit that contains a specific subset of functionalities, and is designed to be assembled with other chiplets to construct a more complicate chip system. The chiplet design has been a highly discussed topics in the semiconductor fabrication industry, due to its advantages over the traditional single system-on-chip design. At Hot Chips 33, 2021, AMD Ryzen[25] was announced with new chiplet technologies and 3D stacking, Intel[21] also talked about its products with 2.5D and 3D packaging and focused more on the Heterogeneous Integration brought by its chiplet packaging, TSMC[28] and other companies also discussed their distinctive technologies and the prospective for the future development[26]. There are also Compute Express Link(CXL)[24] as an open standard for high-speed cache-coherent interconnects and Universal Chiplet Interconnect Express(UCIe)[1] for a universal interconnect protocol at the package-level. In general, there are mainly two reasons why chiplet technologies are a better option:

Lower Cost

As Moore's Law is slowing down, the cost to manufacture a increasingly large monolithic chip system on a single die has been steadily increasing, especially due to the yield limitation of complicate design on a single large die. When a single chiplet is defective, the manufacturer can replace it with another, instead of discarding the entire chip or downgrading the chip to a lower model. With the chiplet approach, there is less waste and a higher yield compared to the monolithic design[7][8].

Heterogeneous Integration and IP Reuse

As long as it follows the chiplet protocols accordingly, each intellectual property(IP) can be independently designed into a small chiplet, and later incorporated into the entire system. This enables designers to conveniently reuse heterogeneous chiplet blocks which already exist

and integrate them into a larger system. It significantly reduces design time and difficulty, via re-utilization of IP blocks and the ease in integration of heterogeneous IP blocks with interfaces of the same standards[19].

One motivating use case would be integrating workload-specific accelerators into datacenter servers with chiplet interconnection. For example, Protobuf accelerator[16] accelerates the execution of Protocol Buffer (Protobuf), which is a common datacenter workload in Google datacenters consuming 9% of CPU time and a significant portion of the "datacenter tax"[15]. Protobuf accelerator provides an average 3.8x improvement vs. a Xeon-based server. However, it is not so feasible for the datacenter operators to make their own single-die chip with everything they want. Then, chiplets would be an option that allows for a more cost-effective and flexible solution to introduce specialized accelerators, such as Protobuf, to the system[20].

Challenges

There are still a lot of challenges in the development of chiplet technology. On the physical design side, there are still issues such as power dissipation, thermal analysis, and better 2.5D and 3D packaging[4]. Besides, there are challenges in mitigating the longer latency caused by chiplet interconnection and memory coherency in the perspective of both hardware and operating system implementation, as well as the challenges in the new business model in the industry[12]. Hence, there's essential need for further study and better simulation model for the chiplet architecture.

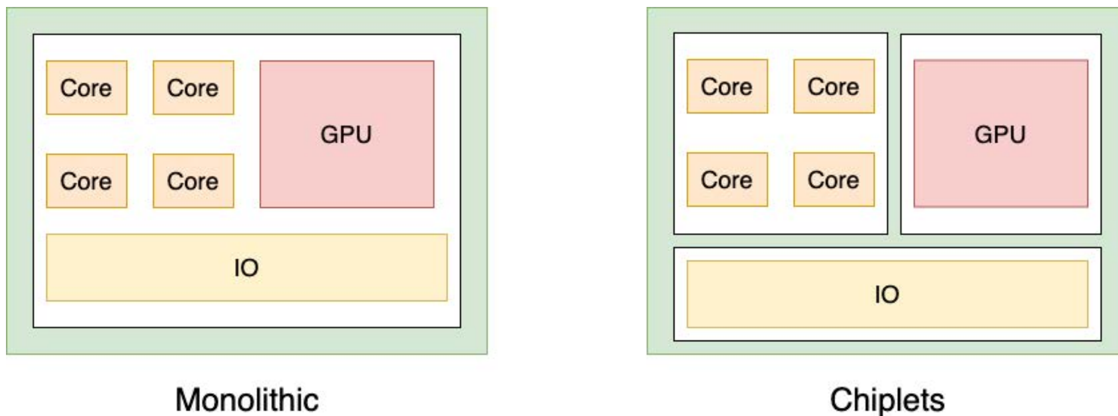

Figure 2.1: This is an comparison between a SoC on a monolithic chip and the same design with the chiplet architecture. The green block represents the scope of the chip and components on the same white tile are on the same die.

2.2 Network-on-Chip

When talking about Chiplet, which raises the need for the increasingly scalable technologies to support the greedy need of more cores and larger caches, Network-on-Chip(NoC) is also mentioned. NoC is a router-based packet switching network that enables an efficient on-chip interconnect. NoC architectures provide a unique intersection of scalability and performance that makes them an ideal target for such systems. Also, a well-designed NoC makes the process of modularization of the entire system easier for a chiplet design and provides robust communication interface and protocols between the modules.

2.3 FireSim and Other Tools

This project builds the chiplet modeling infrastructure under the FireSim framework [18]. FireSim is an open-source simulation platform that supports cycle-exact micro-architectural simulation. It runs on Amazon EC2 F1, a public FPGA platform, providing wide usability, elasticity, and low-cost large-scale FPGA-based experiments. It supports different levels of simulation: Target-level simulation on pure target hardware RTL design, Meta-simulation containing both the target RTL and an abstract model of host, and the FPGA-accelerated simulation. FireSim Meta-simulations are only slightly slower than target-level ones while provide modeling of communication between target and hosts, which is extremely useful for fast prototyping and debugging generations of the chiplet interconnection bridges.

Currently, Supernode[18] in FireSim demonstrates its ability of simulation large scale-out clusters by combining FPGA-accelerated simulation of silicon-proven RTL designs with a scalable, distributed network simulation. However, the communication deploys Ethernet Protocol, which needs the coordination of Operating Systems and has a much longer latency than the desired chiplet interconnections. FireSim also support the Chipyard[2] RoCC-standard interface for adding accelerator, though that's an interface for monolithic chip design. Hence, there is necessity to build new features in FireSim to support chiplet-like latency modeling. Fortunately, there is an existing framework in FireSim for adding new bridges between FPGA targets and CPU hosts, rendering an opportunity to build a set of new bridges for chiplet connection modelling.

To align with the FireSim norm, Chisel[3] is used as the hardware design language. Chisel is a hardware construction language that supports advanced hardware design using highly parameterized generators and layered domain-specific languages. It can generate high-speed C++ based cycle-accurate software simulator, which was convenient and useful in the early stage of hardware unit design and test in this project, and also low-level Verilog for later integration in the later system-level simulations in FireSim.

Chipyard[2], as mentioned above, is a framework for designing and evaluating full-system hardware using agile teams. It provides a lot of open-source tools and pre-built hardware modules, which can be easily reused in this project and alleviates the workload of the in-

tegrating and prototyping a system with the customized communication bridge designed in this project.

2.4 Related Works

There are previous works that model chiplets in different perspectives, such as SIAM[20] and KITE[6], but they primarily focus on tangential goals to this project. [20] proposes a SPICE-based and behavioral software model for chiplet-based architectures for DNNs, and focuses specifically on in-memory architectures. Although it allows for more complex architectures that use multiple chiplet accelerators, the model is entirely software and mainly focuses on In-memory computing (IMC)-based architectures and modeling DNN accelerators. Our solution, on the other hand, would be able to target any generic chiplet blocks that uses the same interconnection standard.

On the other end of the spectrum, [6] is a simulator that focuses on simulating the interconnects between chiplets, and evaluating different Network-on-Interposer (NoI) topologies and how best to form these interconnects within a package across chiplets. It provides some aspiration for this project, though it has a different Network-on-Chip settings from this project.

In addition, [8] approaches the problem from a cost perspective, via building a quantitative cost model and an analytical method for multi-chip systems in terms of yield improvement, chiplet and package reuse, and heterogeneity. It re-examines the actual cost of chiplet systems from various perspectives and suggests how to reduce the total cost of the VLSI system through appropriate multi-chiplet architecture.

In terms of using multiple FPGAs for simulation of large systems, there are several related works, though in various different fields. As mentioned before, [18] use multiple F1 FPGA instances in Amazon Web Services (AWS) to simulate the performance of a large scale of networks. [27] uses multiple FPGAs to combine the speed of dedicated hardware with the programmability of software in simulating neural networks. [9] proposes an automatic synthesis of an area-efficient, high-performance networks for routing inter-FPGA links. It gains some significance in design feasibility, compilation time, and wall-clock performance. However, our case has a different environment of FPGAs and host CPUs set-up, making it not so suitable in our case.

There are several chiplet-based accelerators, which is similar to the base chiplet system for modeling, a core and an accelerator, and may be used as validation for our simulation system. [23] is one such chiplet-based accelerator developed by NVIDIA. However, they seem to be using chiplets to take advantage of the tile-based structure of DNN accelerators, using chiplets to improve yield and achieve larger tile sizes rather than placing the accelerator closer to the CPU. Another paper is [13], which utilizes an ASIC/FPGA-based ML accelerator that uses a package-integrated CPU/FPGA design. While there are some key differences with this design, the target is a single CPU to single accelerator system that is easy to reproduce. This

potentially opens up a validation pathway using Gemmini[11], which is a similar tile-based GEMM accelerator that can be configured similarly to the one used in this paper.

There are also previous works add latency in some existing systems to inspect guidelines for the future development, which is similar to the goal of this project that inserting proper chiplet interconnection latency paves way for further research in the chiplet field. In [10], it adds network latency between the communication of existing clustered datacenters to model the performance of the disaggregated version. The main focus of this work is not on representing an accurate model but more on what are the potential effects on the performance under different conditions. It sweeps through a spectrum of network latency, and explores what are the network requirements for designing such a disaggregated data storage system. It also provides some suggestions on what should be improved to make a successful product. Similarly, this project simulates the chiplet architecture in RTL level, sweeping through a range of different chiplet latencies, and pay attention to the implications of the simulation results as well as the simulator itself.

Chapter 3

Design

This project aims to build a bridge between FPGAs and host CPUs with chiplet-like interconnection latency to support simulation of chiplet simulation across multiple F1 instances in AWS. Before building a fully generic chiplet bridge supporting all kinds of communication protocols, this project chooses TileLink Protocol as its first step. The reason is that TileLink Protocol is conveniently supported in Chipyard and FireSim, and several other projects in the SLICE Lab, such as the Constellation NoC project, deploy it in communication. After the TileLink Protocol bridge is implemented, architectural researchers using FireSim would be able to replace the interface of this bridge with the desired interfaces for their specific communication relatively easily.

In this Chapter, after a brief overview on the TileLink Protocol, it will explain the process of building a target-to-host bridge in FireSim, with an existing Ethernet-protocol bridge as an example. Then, the design of the bridge will be illustrated in details, including the hardware bridge module on the FPGAs, and the software driver and switch run on the host CPUs.

3.1 TileLink Protocol

TileLink[14] is a free and open interconnect standard providing multiple masters with coherent memory-mapped access to memory and other slave devices. TileLink was originally designed for connecting general-purpose multiprocessors, accelerators, DMA engines, and other simple or complex devices on System-on-Chip (SoC), as a fast scalable interconnect of low-latency and high-throughput transfers. It has many other features, such as cache-coherent shared memory with MOESI-equivalent protocol and verifiable deadlock freedom for any conforming SoC.

Within the complicated architecture design of TileLink Protocol, the network link and its channels are the most interesting parts for this project, since the chiplet modelling bridge needs to accommodate the interface while keeping other perspective the same. A network link in TileLink Protocol is unidirectional, connecting a master interface on one end and

a slave interface on the other end. In the TileLink Cached (TL-C) version, there are five channels:

- **Channel A** transmits a request from the master that an operation should be performed on the targeted address range, accessing or caching the data.
- **Channel B** transmits a request from the slave that an operation should be performed at an address cached by a master agent, accessing or writing back that cached data.
- **Channel C** transmits from the master a data or acknowledgement message for a Channel B request.
- **Channel D** transmits back a data response or acknowledgement message to the original requester (the master).
- **Channel E** transmits a final acknowledgement of a cache block transfer from the original requester, used for serialization.

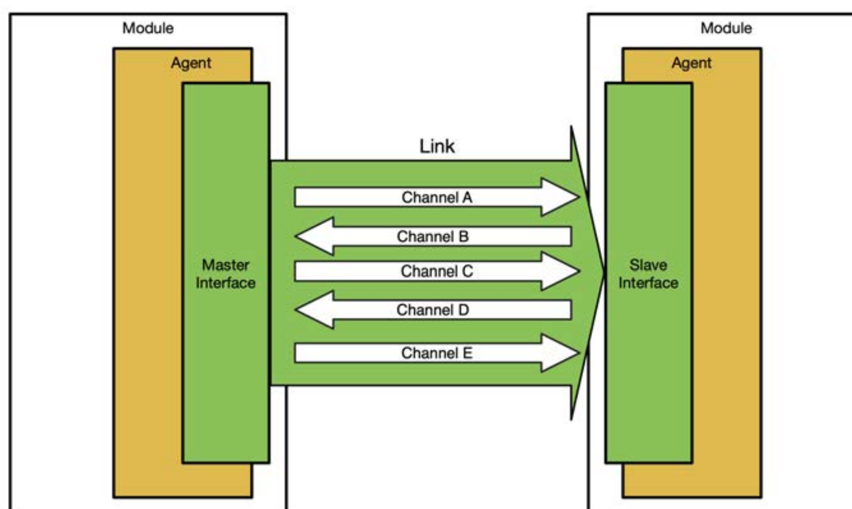

Figure 3.1: The channel architecture of a TileLink Link between a pair of agents.[14]

The signals in each TileLink Channel are different, since they serve different purposes. Among all the signal fields, several are more important than the other during developing and testing. For example, the source and sink field contain the unique, per-link master and slave source identifiers, which is useful in our project to build a system simultaneously supporting multiple chiplet connections.

3.2 FireSim Target-to-Host Bridge

In the documentation of FireSim[17], `Target-to-Host Bridge`, or `Bridge` for short can be deployed as a custom model to build specialized I/O models or distributed simulations. The `Bridge` framework render formalized process to inject hardware and software models which generate and consume customized token streams.

A `Bridge` has a target side, consisting of a Module connecting to the target hardware device simulated on FPGA, and host side, consisting of an FPGA-hosted `Bridge` module and CPU-hosted Bridge drivers. The `SimpleNICBridge` is an Ethernet-protocol bridge used in `IceNet` library[22] for multi-node network simulation in FireSim. It acts as a great example and provides guidance for this project, in building a custom communication bridge F1 FPGAs in the AWS environment. Therefore, it will be used as an example to illustrate the architecture of an `Bridge`.

First of all, `NICTargetIO` is defined to capture the target-side interface of the Bridge, and it then defined a Black-Box module of `NICBridge`, which can be instantiated at the top-level of the chip system and connected to the Ethernet port from the target hardware. On the host side, `SimpleNICBridgeModule` is constructed, containing the actual logic to process the tokens from the targets and send it to the software driver, and expose a memory-mapped interface for the driver. In `SimpleNICBridgeModule`, it packages seven tokens streamed from the target hardware with additional control signals into a 512-bit packet towards the PCIe ports, and disassemble the received PCIe packets to the token receivers in the target hardware. It also sets up some Memory Mapped I/O (MMIO) registers to take in parameters from the software drivers, such as rate limit settings and MAC address upper and lower bounds. In the software side, there is a `simplenic_t` driver that parses the large tokens taken from the PCIe ports into Ethernet tokens, a switch for routing and scheduling, and a Linux driver that actually process the Ethernet Protocol.

3.3 Overview of the Design

The `ChipletNICBridge` of this project contains three major parts:

- `ChipletNICBridgeModule` is the hardware bridge module, converting the TileLink Bundles into big 512-bit tokens for PCIe on the transmitter side, and the reverse on the receiver side.
- The software bridge driver `chipletnic_t` reads and writes big tokens from and to the PCIe and communicate to the FPGAs, as well as process the data for the `switch` when necessary.
- The `switch` handles the routing of network and directs the tokens to their destination FPGAs.

Some may doubt why this project doesn't build a bridge directly between two FPGAs. As mentioned before, due to the infrastructure of the AWS that this project uses, it is more feasible and compatible to have a CPU handling the communication between FPGAs. The diagram below presents an example of simulating a large chiplet systems of multiple chiplets using `ChipletNICBridge`, with multiple FPGAs in the AWS infrastructure.

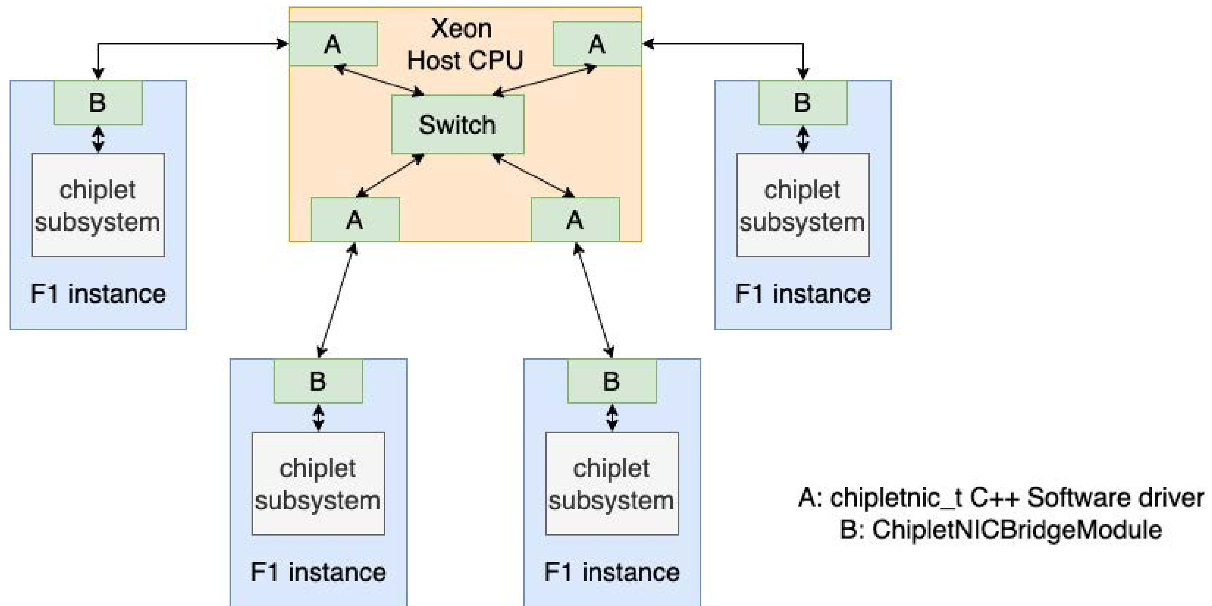

Figure 3.2: Simulating a 4-chiplet system with 4 F1 instances and a Xeon Host CPU in AWS

3.4 Hardware Bridge Module

As discussed in Section 3.1, there are 5 channels in a single link in TileLink Protocol, with Channel A, C, and E from master to slave, and Channel B and D in the reverse direction. Theoretically, there should be two sets of transmitter-receiver hardware modules to handle with both sides of communication in a single link, and two links for an entire bidirectional connection. For each link, the master side of the module consists of ACE-channel transmitter and the BD-channel receiver, and the slave side of the module is made up of the BD-channel transmitter and the ACE-channel receiver. Since both sides of the link is mostly symmetric, except for the direction of the channels, this section mainly explains the hardware design of the master side of the `ChipletNICBridgeModule`.

IO Bundles

Bundle in Chisel is a base class for data types defined as a "bundle" of other data types, which is kind of an analogy of `struct` in C or C++. There are three kinds of IO Bundles used in this project:

- `TLBundleIO` consists of the `Decoupled` (with the ready and valid bits) TileLink Bundles for channel A, B, C, D, and E, with ACE as the output and the BD for the input on the master side, and the reverse on the slave side.

- `ACEBigToken` selects the necessary fields in the `TLBundleIO` and makes it easier for the both receiver hardware and software to decode and process the token. As shown below, it reserves the last bit to tell the decoder in both hardware and software whether it is a token for the ACE-link or the BD-link, the next last five bits for storing the valid and ready bits for the five bundles.

```
class ACEBigToken(params: TLBundleParameters) extends Bundle {
    val A = new TLBundleA(params)
    val C = new TLBundleC(params)
    val E = new TLBundleE(params)
    val Avalid = Bool()
    val Bready = Bool()
    val Cvalid = Bool()
    val Dready = Bool()
    val Evalid = Bool()
    val isACE = UInt(1.W)
}
```

- There's also the `BDBigToken`, which is similar to the `ACEBigToken` but serves the BD link.

Transmitter

On the master side of the bridge, there are mainly two components in the ACE-channel transmitter. The first one is the `ACEBigToken Generator`, which takes in the output signals in the `TLBundleIO`, packs it into the `ACEBigToken` format, and finally convert it into a 512-bit token fed into the PCIe port. The second part is the MMIO registers, which are used to store metadata for the link. There is an already set-up `Widget` bridge in FireSim to handle all these MMIO register. Though the latency of this communication is quite long, usually in several micro-seconds, it provides a convenient interface for data transfer between the FPGA and host CPU. In the transmitter side, the hardware stores the parameters of the deployed TileLink bundles, which are essential for the software to know how to parse the incoming tokens.

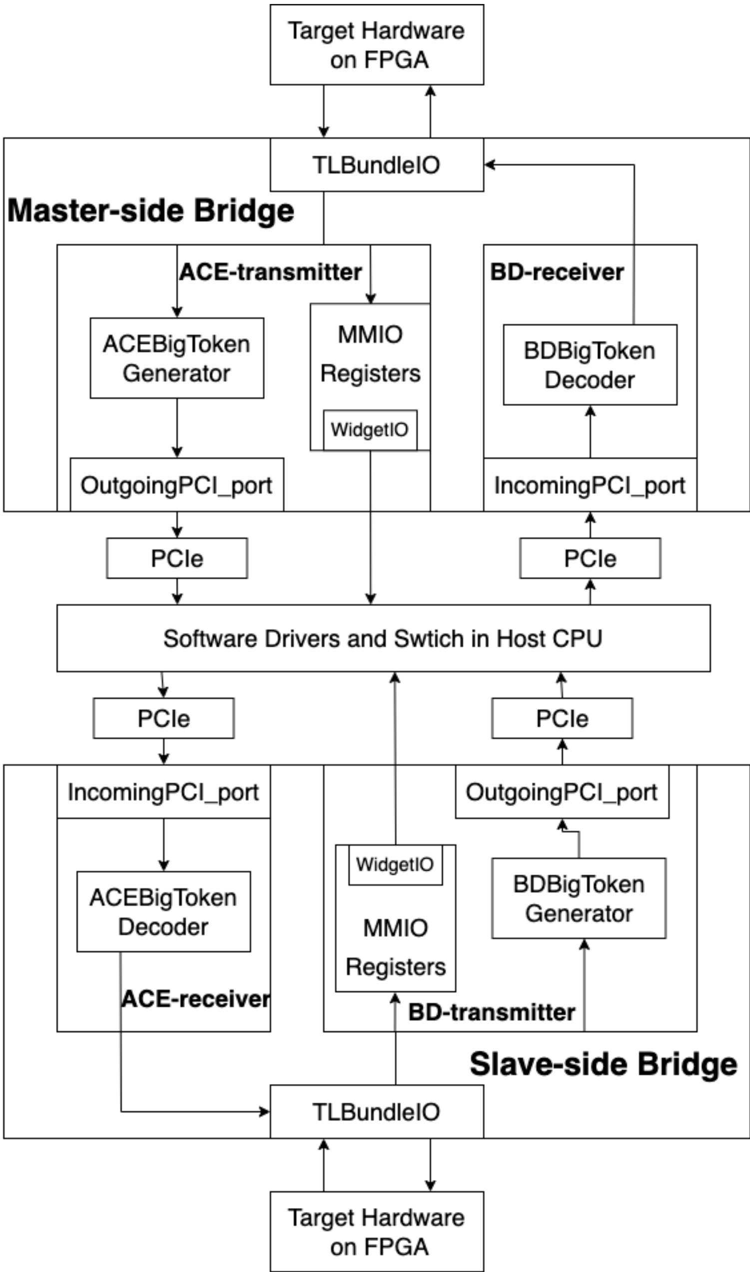

Figure 3.3: The architecture of the hardware bridge modules.

Receiver

Compared to the transmitter side, the receiver in the bridge is much less complicated. Basically, it takes in the `BDBigToken Generator` or `ACEBigToken Generator` from the PCIe

port, decode the token, and outputs the TileLink bundles back to the proper channels connecting to the targets on the FPGA.

3.5 Software Bridge Driver

Similar to the hardware bridge module, each software driver also has two side, the input side that takes in and processes the PCIe big tokens, and the output side that produces the PCIe tokens. During initialization, the driver on the input side reads in the parameters of the TileLink Channels built in this simulation to configure its inner data structure. The `Widget` framework in FireSim automatically generates a `struct` that contains all these MMIO registers, rendering great convenience in accessing these variables.

The communication between the software driver and the switch is double buffered. For each driver, it has two `Input Buffer File`s for data flowing from the network into the switch and two `Output Buffer File`s for data flowing from the switch towards the network. On each tick of the software, it switches the buffer to be used. This enables a relatively larger overall throughput of the software via hiding part of the memory copying latency, providing a larger bandwidth for this custom bridge.

The software driver also takes the advantage of two MMIO registers pre-built in the PCIe pipes, `outgoing_count` and `incoming_count`. They records the number of valid output tokens from the FPGA side and the number of available slots in the queue towards the FPGA. It allows the software side to verify if it gets the correct number of tokens and process a burst of tokens together once when there are sufficient tokens, instead of fetching small number of packets frequently. It increases the bandwidth via reducing the average processing time for each token in the software.

On each tick of driver, the input side of the driver check if there is enough number of tokens in the queue. Then, it reads in a burst of tokens from the PCIe port, parses the tokens, and store all the tokens into the buffers shared with the switch program. With the information gained during initialization, the driver can easily convert the sequences of bits into actual TileLink bundles of different channels, getting prepared for the further processing in the switch program. Conversely, the output side of the driver reads from the Output Buffers and sends to the PCIe port.

The driver also supports insert an parameterized latency for the tokens, aiming to support various latencies of chiplet connections. The current way to deal with the latency is by adding tokens containing all zeros before the actual token in the Input Buffer Files, according to the desired latency. In such way, the switch routes these dummy tokens to the destination output drivers, and the output driver needs to "generate" some invalid tokens that didn't goes into the PCIe, before it outputs the actual token. This is an easy way to generate the latency, though it has a disadvantage of creating unnecessary data movement inside the CPU and may cause a bandwidth downgrade. Another option would be that the input side calculate a delayed timestamp and append this to each token, and the output side will only

output the token when the expected timestamp arrives. This option is considered, but still in the developing phase, and, hence, will not be explained in details in this paper.

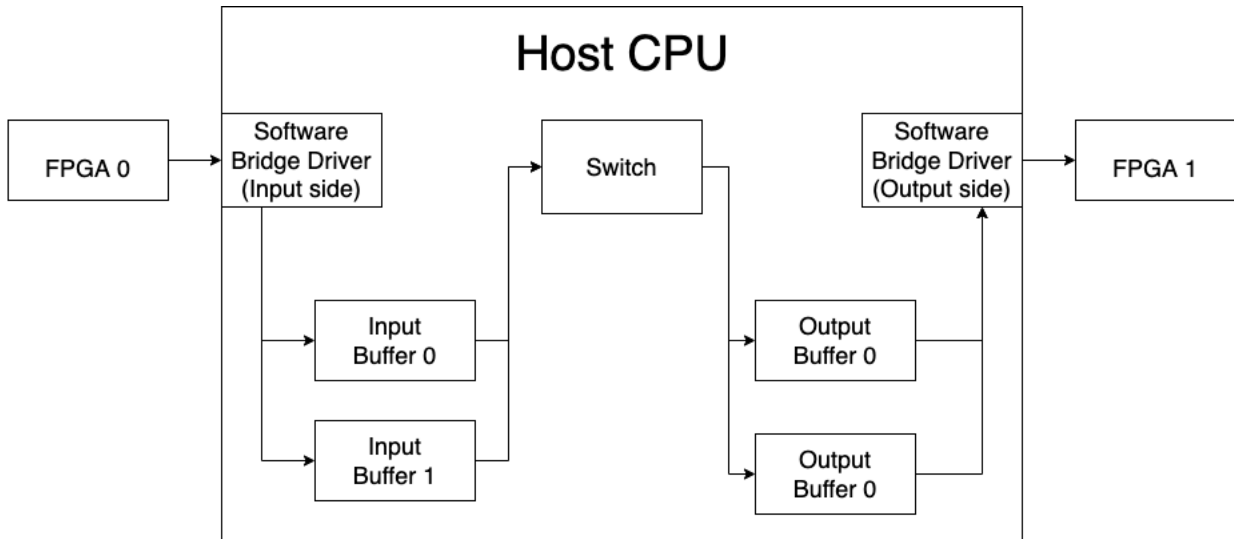

Figure 3.4: This figures demonstrates the data flow through the architecture of software bridge drivers and the switch on the host CPU for a single direction of a link. (The output buffers of the input side driver and the input buffers of the output side driver exist but are not shown in this diagram to save space.)

3.6 Software Switch

On the high level, the software switch reads in the data stored in the input buffers files, and writes the read-in data to the accordingly output buffer files. It also take advantage of multi-threading speed-up via using the `OpenMP` library. The number of threads in the parallel region equals to the number of agents in the network, which is the number of software drivers in this case. There are several tasks that are independent and can be parallelized:

- Setting control variables of a single `ShmemPort`, which is the interface towards the input and output buffer files of a software bridge driver;
- Reading tokens from the input buffer of a driver and process the tokens;
- Writing back the tokens from the respective output queue towards the actual output buffer of a driver.

Since all theses tasks are independent across `ShmemPort`s, each `ShmemPort` can be assigned to a thread. However, further multi-threading inside a `ShmemPort` will mess up the order of the tokens, and hence not implemented.

In the naive version with only two FPGAs in the system, the mapping in the switch is hard-coded, since all communication can only goes to the other direction. This speeds up the process of building a simple model for running integration simulation for two FPGA nodes to test correctness and profile basic features.

In order to support mode FPGAs, a chiplet-FPGA mapped should be created. It records the all identifiers of sinks and sources existing in the system and matches them to the respective FPGAs. This requires a lot of knowledge of the simulated TileLink protocol systems to build the map appropriately. Some potential future works would be finding a way to automatically generate the map in the infrastructure setup or simulation time instead of hard-coding it in the developing phase.

Chapter 4

Testing and Evaluation

There are several phases of testing and profiling on the `ChipleNICtBridge` system, with different levels of integration and in approaches of simulation. This chapter will first explain the experiment set-ups and some special designs for the testings. Then, it will enumerate all the tests made for this project. After that, it will discuss some performance benchmarks of this bridge, including latency and bandwidth, gained by profiling a prototype of a simple system annotated with this bridge. Finally, some further discussion will focus on the system integration of this bridge.

4.1 Experiment Environment Options

As the table below shows, there are mainly three different environments for testing or simulating the system, each serves a different purpose:

Type	Description	Resources used
Chisel Test	Tests for pure hardware targets.	local machine with Chisel installed
FireSim Meta-Simulation	Software simulation of both the hardware and software driver and switch	z1d.3xlarge or z1d.6xlarge instance in AWS
FireSim FPGA-accelerated Simulation	FireSim simulation with hardware target simulated in FPGAs with software driver and switch run on Host CPU	f1.2xlarge instance in AWS

Table 4.1: A brief summary of three different kinds of simulation utilized in this project

- **Chisel Test** is basically used for the early stage tests for verifying the hardware units until the phase that the entire hardware bridge is implemented and checked. It can be executed on any local machine as long as `Chisel` is properly installed. In addition, the `Chisel Bootcamp`[5], which can be run in `JupyterLab` also provide a convenient

interactive approach to generate the hardware unit RTLs and make small unit tests for the target design. However, some modification would need to be made to make sure the `chisel3` instead of `chisel2` is used.

- **FireSim Meta-Simulation** utilizes the `manager-metasimulation-support` branch in the FireSim and runs the simulation of both hardware and software components in the traditional FireSim simulations in a single `z1d` instance in AWS. For small design with relatively light workload, it takes the advantage of using `verilator` software for hardware simulation, which decreases time cost in developing different versions of hardware and software codes, instead of waiting for hours on `firesim buildafi` commands to be finished.

- **FireSim FPGA-accelerated simulation** is the most common way of simulating a system in FireSim platform. It requires a `f1.2xlarge` instance in AWS and is able to execute larger workload on the simulated system. This is the set-up for the final integration simulation in this project.

4.2 Designs for Tests

In order to build a simple prototype to test the functionality of the bridge, several small hardware gadgets are built. `TileLink Bundle Fuzzer` is designed for generating random or pre-programmed TileLink bundles that follow certain patterns. It starts off as a random number generator and gradually involves into a more complicated version displayed below to meet the increasing needs from testing.

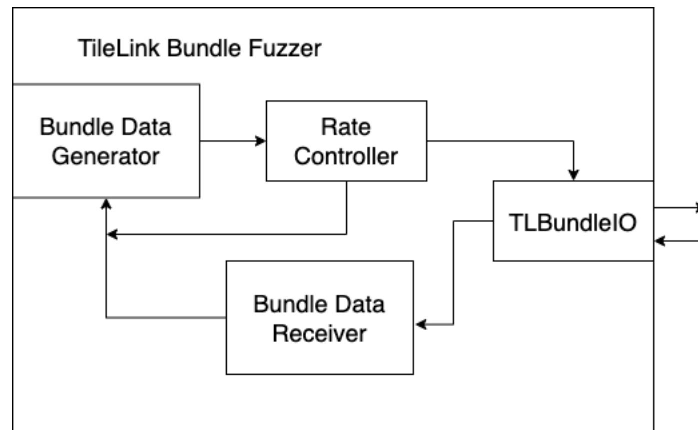

Figure 4.1: The TileLink Bundle Fuzzer designed for generating `TLBundleIO` inputs when testing and prototyping the `ChipletNICBridge`.

- **Bundle Data Generator** has two modes of granularity. One is to treat each channel bundle as an entity, and generate a random number or a value from a sequence of pre-defined patterns. In the finer granularity mode, only the data is randomly generated, the other fields, such as sink and source are determined by the integrated system configuration, serving the tests on more functionalities in the software driver and switch.
- **Pattern Controller** controls the rate or intervals between each valid token and can set the maximum number of tokens to be generated, in order to not overflow the communication network and test the bandwidth and other performance of `ChipletNICBridge` system. It sets the valid bits for the bundles, and is able to zero out selected channels or fields when needed.
- **Bundle Data Receiver** consumes the received bundles and makes some sanity checks. Optionally, it can check if the received bundle contains certain value of data, and notifies the **Bundle Data Generator** to generate a specific value in the next cycle.

In addition to the TileLink Bundle fuzzer on the transmitter side, a RAM is also created to consume all the incoming 512-bit tokens from the PCIe port on the receiver side. The bundles are first stored in the RAM and later read out for further verification.

4.3 Testing

Testing is an essential part for verifying the functionality of the design in this project. There are mainly three stages of testing:

Unit Tests in Chisel

In the early stage of this project, the main focus is to review the Chisel programming language and understand related hardware designs and libraries. Various hardware component blocks are built and tested in the Chisel Test set-up mentioned above.

Hardware Loop-back Tests in Meta-Simulation

When the initial version of the hardware bridge module is implemented, we start to move to the FireSim Meta-simulation, which instantiates a single bridge module in the `FireSim.scala` as the top of the design and paves way for later work of integrating the software components. Instead of connecting the output of the bridge towards the PCIe DMA ports, it directs its output towards the receiver and checks that it receives the same data as sent, as it is displayed in the diagram below. It uses the no-nic topology `topology=no_net_config` and use a `z1d.3xlarge` instance with `metasimulation_enabled=yes`.

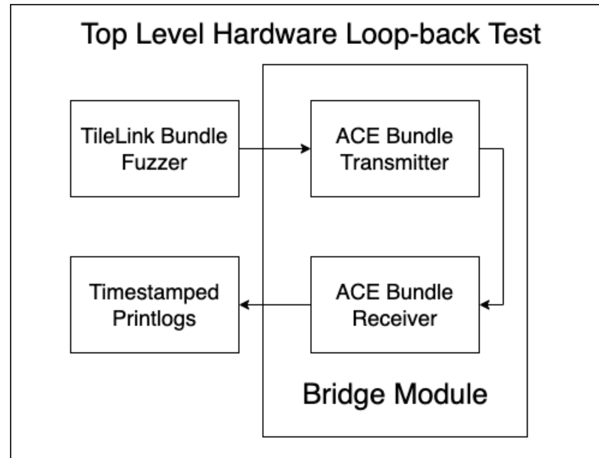

Figure 4.2: The architecture of one testing design used in Hardware Meta-Simulation

Software-Hardware Integrated Tests in Meta-Simulation

After verifying the hardware alone, tests have been made to verify the hardware alone, two integrated tests are made for testing the software drivers and switch. First, we executed a Loop-back test with a single bridge and the newly-designed software driver. The abstract design of this test is shown below. The software driver takes in the input from the FPGA and writes back the same data to the same hardware via PCIe. The hardware used only here for the test purpose is a little different from the final version, and it has the ACE-transmitter and ACE-receiver for the ease of loop-back tests. On one side, it checks if the data at each point of IO/port is the expected value; on the other side, it timestamps all the data transaction and validates if the software driver is inserting the proper latency.

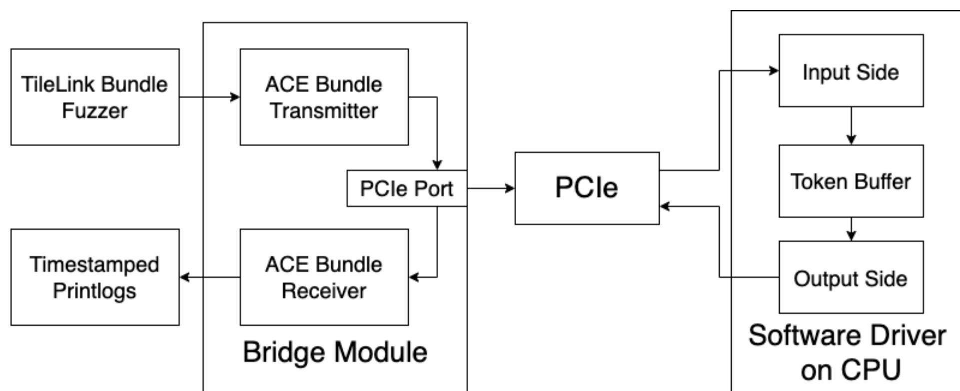

Figure 4.3: The diagram of the software driver loop-back test.

Having verified the software drivers, the switch is also integrated in the next stage. The layout of this test is similar to the set-up in the profiling and will be illustrated in the section below. The newly implemented switch is quite different from the Ethernet switch in the way they read in, parse and writes back tokens. The proper data-flow is checked in this stage to make sure the switch shares the same buffering standard as the software drivers.

4.4 Prototype for Profiling and Evaluation

After testing, a prototype system is built for evaluating the basic performance of the bridge, including the ACE- and BD-Chiplet hardware bridge, their respective software drivers, and the switch. It requires a `z1d.6xlarge` to run it in meta-simulation and a customized network topology for the prototype design displayed in the diagram below. It sets the frequency setting `PLATFORM_CONFIG=F90MHz_BaseF1Config`, which is a typical frequency configuration for simulating rocket mutli-core systems.

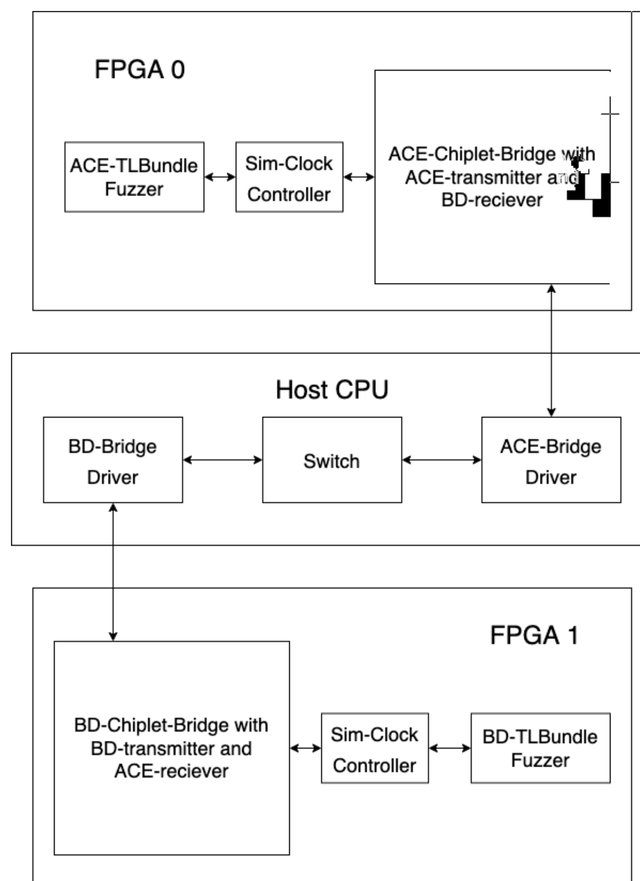

Figure 4.4: The prototype system used for profiling and evaluation

Due to a compatibility issue that will be explained in the next chapter of future works, there are some compiling problems when using the existing `HostPort` API. Hence, an additional `Sim-Clock Controller` is implemented between the bridge and the rest of the design, for synchronizing the hardware clock and the software driver. The Sim-Clock controller generates another clock for simulation that only advances when there's a token received from the network in this physical cycle, to match up with the software speed. It also has two counters recording the "simulation" and the "physical" cycle counts, which would be useful when later analyzing the performance of the bridge.

4.5 Performance Evaluation

The benchmark used in the evaluation are sending a long sequence of TileLink Bundles through the bridge. The master side of the bridge takes in randomly generated ACE channel data and the other side takes in BD channel data. The receiver on each sides decodes and logs the bundle values and compared log to the log generated by the transmitter side, for checking correctness and analyzing timing.

Software Throughput Smaller than FPGA

Initially, when running the benchmark without the `Sim-Clock Controller`, an increasing latency occurred as more tokens are sent and there were some token drops. After some investigation, we found out that the PCIe and software drivers operate in a speed that is slower than the hardware simulated in the FPGA. This also motivates the design of the `Sim-Clock Controller` in the bridge.

Latency

We swept through different chiplet connection latencies with the benchmark and recorded the time when the first valid token sent from the transmitter bridge is received in the decoder of the receiver bridge. The received time in the simulation cycle is exactly the same as the pre-set chiplet latency, as expected, and the latency for the following tokens also remains the same as the first one. Meanwhile, the received time in the actual FPGA platform clock is larger but scales linearly with the configured latency. The linear relation is

$$y = 2x + 109$$

where y is the time when the first token received in FPGA clock, x is the modelled latency, and x and y are in cycles.

Due to the implementation of the software driver, the minimum latency that the bridge supports is 2 cycle, which is sufficiently small for modelling any type of chiplet connection available with the current technology. It also supports any larger latency, and the largest latency in this experiment is 200 cycles, which is already overly large for chiplet connections.

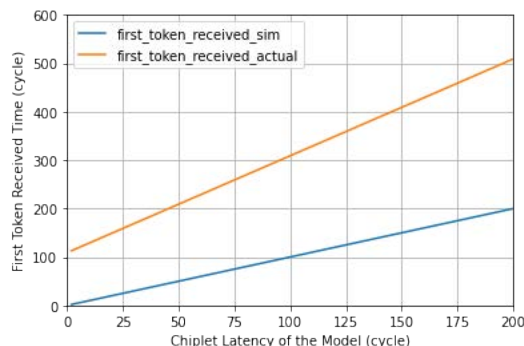

Figure 4.5: The first token received time in cycles in both the simulation clock and the actual FPGA platform clock.

Simulation Speed

In order to better understand the simulation performance in terms of speed, the benchmark with 10,000 cycles of continuous valid TileLink Bundle data is being run on a range of chiplet latency models, and the runtimes are collected. Due to the implementaion of its latency controlling logic, the runtime increases exponentially when the modelled latency approaches 0. The software driver fetches the number of latency of tokens at each tick from the PCIe into its input buffer of the current around, and send the same amount of tokens to the PCIe from the output buffers. The switch also deal with that amount of tokens for each port in each round. In such way, the software makes sure the proper orderings of the tokens and reduces the individual simulation latency for a TileLink bundle. Due to the overhead in each tick of the software, decreasing latency will increase frequency of ticks and thus increase the overall runtime.

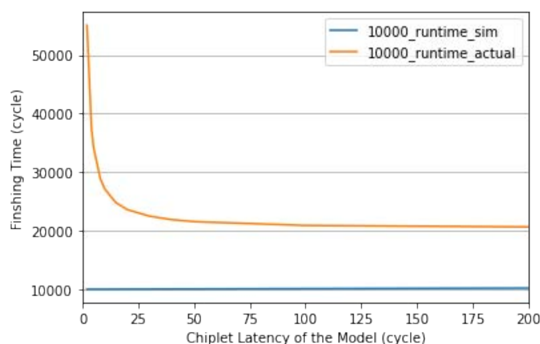

Figure 4.6: Runtime of transferring 10,000 tokens via the bridge in both simulation clock and the actual FPGA platform clock.

Some may argue that combing more cycles of data than the latency could reduce overhead time. However, in the real system, following program execution may depend on the a previous data transaction, and thus the aforementioned suggestion will change the program behavior and should not be taken.

As the platform configuration is 90MHz, the actual simulation frequency at each of the different chiplet latency is calculated and plotted in the diagram below. When the latency is over 8 cycles it can reach the simulation frequency of 30 MHz, it can reach a 40 MHz simulation frequency if the latency is larger than 30 cycles, and gradually converges to approximately 45 MHz as the latency goes further longer. In the current FireSim, the platform configuration for simulating Rocket-Gemmini system is also at 30 MHz, which suggests that the simulation performance of this bridge is not that ideal but acceptable at this stage.

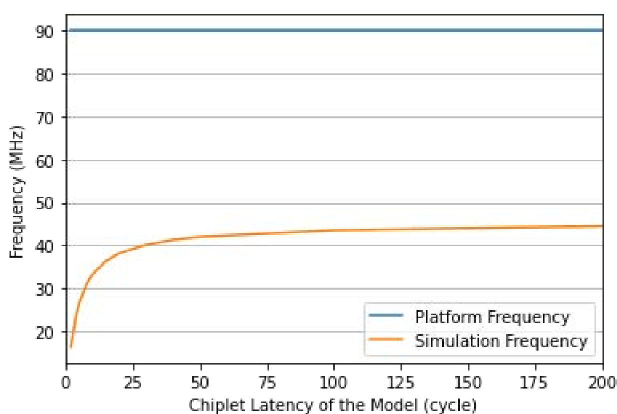

Figure 4.7: Simulation frequency of different chiplet latency vs the platform frequency.

Modeled Chiplet Latency (cycle)	Calculated Simulation Frequency (MHz)
2	16.3
5	26.4
10	33.2
15	36.4
20	38.2
30	40.1
50	42.0

Table 4.2: Calculated simulation frequency of different chiplet latency.

Chapter 5

Ongoing and Future Works

5.1 Full Integration in FireSim with Real Systems

Currently, I'm working on integrating the `ChipletNICBridge` model into the existing FireSim framework and build up a CPU system to evaluate the bridge in the system level. With the help from other colleagues in the lab, the first step is to build a simple system with a CPU core and a DRAM and split it into two parts at a point that utilizes TileLink Protocol for communication. Then, the hardware and software modules in the `ChipletNICBridge` model is instantiated and take charge of the communication between the CPU and the DRAM. Due to time constraints, it is easier to split the system into two parts on the same FPGA instead of splitting the system into two heterogeneous FireSim target designs. In terms of investigating the system performance under the chiplet latency, this design will serve the same purpose.

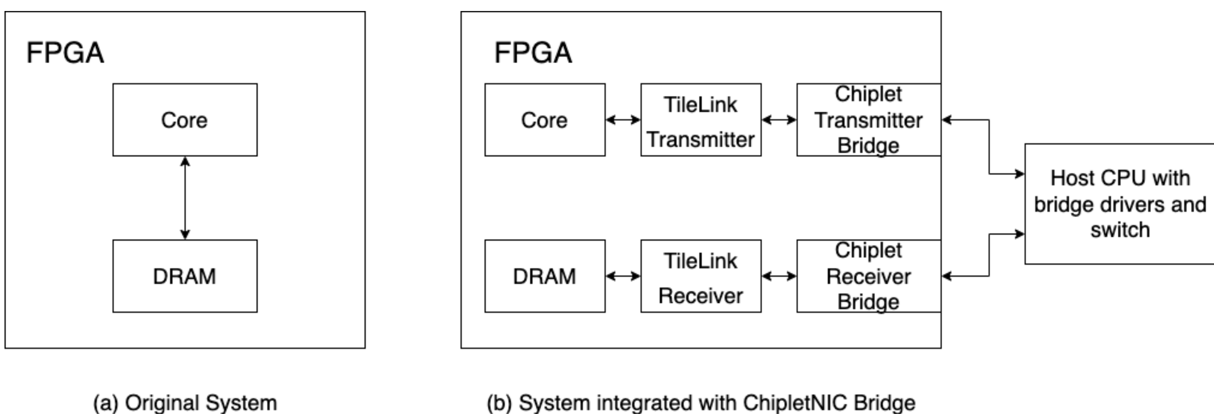

Figure 5.1: The original simple CPU-DRAM system on the left, and the chiplet connection version of the same system.

However, there are several challenges when I want to integrate the entire system with the new designed bridge.

Difficulties in Conveniently Instantiating the Bridge

In the top-level design file `FireSim.scala`, instantiating the bridge directly in the `FireSim` Module will cause a weird compiling error in `HostPortIO`, which is an API that the hardware bridge used to communicate with the rest of the hardware design. The `HostPortIO` always complains about the data-type mismatch, although moving the hardware design inside the bridge and connecting the TileLink bundle ports directly to the transmitter module works properly. Although I spent some decent efforts, the issue exists and forbids me from instantiating the bridge in a convenient way. The formal way of annotating a bridge in FireSim should be invoked when integrating the bridge with the system.

Parameterization

Parameterizing of the TileLink protocol used in the bridge is another challenge. Currently, the convention of defining a bridge in FireSim is to first define a bridge class only in charge of connecting the IOs,

```
class ACENICBridge(params: TLBundleParameters)(implicit p: Parameters)
extends BlackBox with
Bridge[HostPortIO[ACENICTargetIO], ACESimpleNICBridgeModule] {
  val io = IO(new ACENICTargetIO(params))
  val bridgeIO = HostPort(io)
  val constructorArg = None
  generateAnnotations()
}
```

and the `ACESimpleNICBridgeModule` is the actual hardware module where the transmitter and receiver are implemented. Both the `ACESimpleNICBridgeModule` and `ACENICTargetIO` need to take in the `TLBundleParameters`, which is the configuration of the TileLink Protocol used by the CPU and DRAMs. However, adding this parameter is not a trivial work, as FireSim does not reserve the option for this due to lack of need previously.

Currently, when instantiating a new bridge, FireSim will use `BridgeAnnotations.scala` in the `midas/widget` directory. In this file, it assumes all the parameters follow the existing convention and have been serialized into the pre-determined format, and it calls an abstract function `constructor.newInstance(key, px)` or `constructor.newInstance(px)` to instantiate the bridge in the

```
private[midas] case class BridgeIOAnnotation.
```

A lot of efforts have been taken into investigating the way to pass the parameters properly into all required IOs and Bridge modules, though not yet succeeded. The temporary solution is to know the `TLBundleParameters` in advance and hard-code the parameters into various location. However, in order to make sure the bridge is fully integrated into FireSim framework, future works need to be done in fully understanding and modifying the low-level parameter serialization process to make passing parameters conveniently.

5.2 Other Future Works

Currently, the hardware bridge is sending a token into the PCIe port each cycle, no matter if the data inside the token is valid or not. This generates too much traffic and exceeds the bandwidth of software driver. A potential optimization is to reduce communication via sending a token only when necessary. In the hardware, this could be achieved via adding a timestamp in the token, and sending a valid token to the PCIe when there's some valid data in TileLink Bundle Fields or the ready bits change. A more complicated design may compact multi-cycle data into a single token. On the software side, it would take some hard work in parsing the tokens and maintaining the sequential order of the communication transaction among different FPGAs.

Chapter 6

Conclusion

This project attempts to build `ChipletNICBridge` for supporting high-performance FPGA-accelerated chiplet modeling in FireSim. The hardware bridge modules on the target FPGA side, and the software bridge drivers and the switch on the host CPU operates together to enable the fast data transaction across the bridge system. Then, tests have been made to verify the functionalities of the bridge and a prototype system is built for performance valuation. Though substantial efforts have been made, the full automatic integration into the FireSim framework has been only partially implemented, due to the difficulties of fully understanding and adding new parameters options in the existing or legacy codes of lower-level FireSim framework. Since I was working on another NoC compression project in the Fall semester and only started this project in the Spring, the time constraint imposes a great challenge on this project. Some more time and future works would be beneficial to further improve the utility and performance of the work.

Bibliography

[1] URL: https://www.uciexpress.org.

[2] Alon Amid et al. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40.4 (2020), pp. 10–21. DOI: 10.1109/MM.2020.2996616.

[3] Jonathan Bachrach et al. "Chisel: Constructing hardware in a Scala embedded language". In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[4] Brian Bailey. *Many Chiplet Challenges Ahead*. 2021. URL: https://semiengineering.com/many-chiplet-challenges-ahead/.

[5] Stevo Bailey et al. *Chisel Bootcamp*. URL: https://github.com/freechipsproject/chisel-bootcamp.

[6] Srikant Bharadwaj et al. "Kite: A Family of Heterogeneous Interposer Topologies Enabled via Accurate Interconnect Modeling". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218539.

[7] Tim Brookes. *What Is a Chiplet*. 2021. URL: https://www.howtogeek.com/740584/what-is-a-chiplet/.

[8] Yinxiao Feng and Kaisheng Ma. *Chiplet Actuary: A Quantitative Cost Model and Multi-Chiplet Architecture Exploration*. 2022. DOI: 10.48550/ARXIV.2203.12268. URL: https://arxiv.org/abs/2203.12268.

[9] Kermin Elliott Fleming et al. "Leveraging Latency-Insensitivity to Ease Multiple FPGA Design". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '12. Monterey, California, USA: Association for Computing Machinery, 2012, pp. 175–184. ISBN: 9781450311557. DOI: 10.1145/2145694.2145725. URL: https://doi.org/10.1145/2145694.2145725.

[10] Peter X. Gao et al. "Network Requirements for Resource Disaggregation". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 249–264. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao.

[11] Hasan Genc et al. *Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration*. 2019. DOI: 10.48550/ARXIV.1911.09925. URL: https://arxiv.org/abs/1911.09925.

[12] Nicole Hemsoth. *AMD ON WHY CHIPLETS—AND WHY NOW*. 2021. URL: https://www.nextplatform.com/2021/06/09/amd-on-why-chiplets-and-why-now/.

[13] Ranggi Hwang et al. "Centaur: A Chiplet-Based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations". In: *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. ISCA '20. Virtual Event: IEEE Press, 2020, pp. 968–981. ISBN: 9781728146614. DOI: 10.1109/ISCA45697.2020.00083. URL: https://doi.org/10.1109/ISCA45697.2020.00083.

[14] SiFive Inc. *SiFive TileLink Specification*. 2017. URL: https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf.

[15] Svilen Kanev et al. "Profiling a Warehouse-Scale Computer". In: *SIGARCH Comput. Archit. News* 43.3S (2015), pp. 158–169. ISSN: 0163-5964. DOI: 10.1145/2872887.2750392. URL: https://doi.org/10.1145/2872887.2750392.

[16] Sagar Karandikar et al. "A Hardware Accelerator for Protocol Buffers". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 462–478. ISBN: 9781450385572. DOI: 10.1145/3466752.3480051. URL: https://doi.org/10.1145/3466752.3480051.

[17] Sagar Karandikar et al. *FireSim Documentation*. 2018. URL: https://docs.fires.im/en/stable/index.html.

[18] Sagar Karandikar et al. "FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA '18. Los Angeles, California: IEEE Press, 2018, pp. 29–42. ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00014. URL: https://doi.org/10.1109/ISCA.2018.00014.

[19] Jinwoo Kim et al. "Architecture, Chip, and Package Co-design Flow for 2.5D IC Design Enabling Heterogeneous IP Reuse". In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019, pp. 1–6.

[20] Gokul Krishnan et al. "SIAM: Chiplet-Based Scalable In-Memory Acceleration with Mesh for Deep Neural Networks". In: *ACM Trans. Embed. Comput. Syst.* 20.5s (2021). ISSN: 1539-9087. DOI: 10.1145/3476999. URL: https://doi.org/10.1145/3476999.

[21] Ravi Mahajan and Sane Sandeep. *Technology Provider: Intel packaging technologies for chiplets and 3D*. 2021. URL: https://hc33.hotchips.org/assets/program/tutorials/Tutorial_Mahajan_Sane_HotChips_2021_Talk_final_Formatted_1.pdf.

[22] Berkeley Architecture Research. *Chipyard Documentation*. 2019. URL: https://chipyard.readthedocs.io/en/latest/index.html.

[23] Yakun Sophia Shao et al. "Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 14–27. ISBN: 9781450369381. DOI: 10.1145/3352460.3358302. URL: https://doi.org/10.1145/3352460.3358302.

[24] Debendra Das Sharma and Siamak Tavallaei. *Compute Express Link 2.0 White Paper*. 2021. URL: https://b373eaf2-67af-4a29-b28c-3aae9e644f30.filesusr.com/ugd/0c1418_14c5283e7f3e40f9b2955c7d0f60bebe.pdf.

[25] Raja Swaminathan. *Case Study: AMD products built with 3D packaging*. 2021. URL: https://hc33.hotchips.org/assets/program/tutorials/2021%20Hot%20Chips%20AMD%20Advanced%20Packaging%20Swaminathan%20Final%20%2020210820.pdf.

[26] Jan Vardaman. *Expert Opinion: An overview of the package technology landscape and industry deployment*. 2021. URL: https://hc33.hotchips.org/assets/program/tutorials/TechSearchInternational_TutorialHotChips%20FINAL.pdf.

[27] Shufan Yang, Qiang Wu, and Li Renfa. "A case for spiking neural network simulation based on configurable multiple-FPGA systems". In: *Cognitive neurodynamics* 5 (Sept. 2011), pp. 301–9. DOI: 10.1007/s11571-011-9170-0.

[28] Dong Yu. *Technology Provider: TSMC packaging technologies for chiplets and 3D*. 2021. URL: https://hc33.hotchips.org/assets/program/tutorials/2021%20HotChips%20TSMC%20Packaging%20Technologies%20for%20Chiplets%20and%203D_0819%20publish_public.pdf.