Acknowledgement

First and foremost, I would like to thank Professor Borivoje Nikolic for being my advisor over the past year. His guidance and mentorship have helped me to become a better scholar. Additionally, I would like to thank Professor Krste Asanovic for serving as my second reader. Next, I would like to thank the staff and students in BWRC/ADEPT who have helped me over the years. In particular, John Wright for bringing me into the lab, Paul Rigge for guiding me early on, and Vighnesh Iyer for serving as a mentor during my graduate work.

I would also like to thank my research partner, Anson Tsai, who has been a good friend ever since we met on the first day of freshman year. Finally, an enormous thanks goes out to my parents, who have always been there for me during my educational journey.

Design and Application of a Co-Simulation Framework for Chisel

Ryan Arvind Lund

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:

[illegible]

Professor Borivoje Nikolić
Research Advisor

5/14/2021

(Date)

* * * * * * *

[illegible]

Professor Krste Asanović
Second Reader

5/14/2021

(Date)

Abstract

As the cost to design chips increases, an ever-growing portion of the design cycle is spent in pre-silicon verification. When performed in industry, this verification work is backed by closed-source tools that require licenses for use. However, in the open-source domain, developers often lack access to similar verification resources.

This work aims to increase access to verification tools in the open-source space through the introduction of an instruction-accurate co-simulation framework for Chisel (CFC). The framework is designed to accelerate the verification of tightly coupled accelerators by pairing a functional model of a Rocket Chip-based core with an RTL simulation of an accelerator under test.

CFC implements a series of tools and utilities to elaborate and simulate RoCC accelerators removed from full-SoC context. Additionally, it contains utilities to convert hardware bundles to and from protocol buffers. These elements are used along with verification IP from the Chisel Verification Repository to create tools that connect a modified version of the Spike ISA simulator to an RTL simulation of a RoCC accelerator. When this connection is orchestrated by CFC's manager object, the two simulations form a coherent model of a full SoC. This allows SoC-level workloads to be run at RTL fidelity on an accelerator without the overhead associated with simulating the rest of the SoC at RTL accuracy.

Beyond detailing the design of CFC, this thesis demonstrates its impact by testing CFC on Gemmini, an open-source matrix multiplication network accelerator generator. With the additional optimization of ChiselTest binary caching, these tests show the potential for up to 9.72x speedup in test run time when compared to a full-SoC RTL simulation.

Contents

1 Introduction

In industry, product launch delays due to critical bugs found in post-silicon testing can cost hundreds of millions of dollars in lost revenue and engineering expenses. Accordingly, in contemporary design groups, approximately 60-80% of development time is spent in pre-silicon verification [1]. These verification efforts are often aided by a collection of closed-source tools, locked behind licenses. Unfortunately, these licenses limit tool portability and more generally reduce access to verification flows.

In the open-source world, access to verification infrastructure lags well behind industry. Publicly available simulators often lack support for the SystemVerilog constructs needed to write testbenches for Verilog modules. For newer hardware description languages (HDLs), such as Chisel, verification resources are still in early stages of development, which makes a traditional verification flow almost impossible to implement.

To increase access to and improve upon verification resources in the open-source space, this paper introduces a Chisel-centric co-simulation framework built upon the newly developed Chisel Verification repository [2]. This co-simulation framework for Chisel (CFC) is developed to enhance design space exploration by allowing for high-speed and low-overhead integration testing of hardware accelerators.

1.1 Co-Simulation Overview

Co-simulation is a verification technique wherein distinct subsystem models are coupled and jointly simulated to create a larger coherent simulation. Given a defined communication interface, each model is designed relative only to the interface and a specific sub-problem. This allows each model to be built around optimally solving its given task, and without any constraints imposed by the larger simulation.

An important distinction should be made between co-simulation, wherein different components of a larger system are simultaneously modeled and connected, and tandem simulation, wherein two models of the same device are compared against each other. Tandem simulation is often used to compare register transfer level (RTL) functionality to that of a functional model, which serves a different verification purpose than co-simulation. Confusion can arise around this distinction because tools labeled as co-simulation frameworks often actually perform tandem simulation. However, throughout the scope of this thesis, co-simulation will exclusively refer to the coupling of component models into a larger coherent simulation.

There are three main benefits to co-simulation. First, breaking a simulation apart into several different models allows the simulation to be run in a distributed manner. This distribution can take place across several different compute resources, or a mix of compute resources and physical emulation devices. Second, co-simulation can greatly simplify model creation for large designs by allowing different components to be modeled in different environments. The ability to model in different languages is particularly beneficial when building models for mixed-signal designs. Finally, co-simulation allows different models to run at different speeds. This property has the most impact when models have a producer-consumer relationship, as the producer can generate all of its output data without waiting for the consumer.

Within the digital design space, co-simulation can used for a wide range of applications. These applications can be broken down into three main categories. The first category is mixed-signal verification. In this type of application, an analog model is paired with a digital model to simulate or emulate a larger mixed-signal device. The next category is mixed-medium verification, wherein a software RTL simulation is paired with a field programmable gate array (FPGA) based emulation. As emulation is faster than simulation, this approach is typically used to emulate a slower piece of hardware on the FPGA communicating with a faster piece of hardware in software simulation. The final category is mixed-abstraction verification. In this approach, a lower complexity functional simulation is paired with a higher complexity RTL simulation. This approach is often applied to compare RTL behavior to a functional model, or when the RTL behavior of only one component in a larger system is of interest.

For a given application, a tool designer creates a co-simulation framework. This framework defines the communication interface between simulations as well as the protocol for interacting with that interface. Additionally, most frameworks contain a tool that stages and coordinates the execution of simulations for each sub-model. A key characteristic of this tool is its synchronization resolution, or the frequency at which the simulations it is orchestrating are brought into alignment [3]. Typically, as synchronization frequency decreases, simulation speed increases.

The highest frequency of synchronization is employed by phase-accurate co-simulation frameworks. Under this regime, all models are synchronized for every unit time step, which is most similar to real-world behavior. Cycle-accurate frameworks employ a slightly lower frequency of synchronization. For these frameworks, models are synchronized on every clock edge (either positive, negative, or both). The lowest frequency of synchronization occurs in instruction-accurate frameworks. Under instruction-accurate synchronization, models are brought into alignment either at the commit point for instructions or on instruction set architecture (ISA) specific synchronization instructions. Due to its high resolution, phase-accurate co-simulation is most often used for mixed-signal applications while cycle-accurate and instruction-accurate co-simulation are most often used for all-digital applications.

1.2 State-of-the-Art Solutions

The following section describes several notable co-simulation frameworks available for use. For each framework, a brief overview is provided, and any potential shortcomings are discussed.

1.2.1 Commercial Frameworks

Commercial frameworks are produced by leading companies in the simulation and digital tool space. These products are often fully featured, but require a license for use.

HDL Verifier [4] is a co-simulation framework produced by Mathworks which pairs a Matlab-based software model with an RTL simulation. In this pairing, the software model generates stimulus to input into the RTL simulation, and performs checks on the output from the RTL simulation. The framework also supports reading to and writing from registers, probing internal RTL state, and signal visualisation.

A Matlab-based software model makes HDL Verifier ideal for verifying specific types of IP, such as digital signal pipelines or compression accelerators. However, this also limits the tools when attempting to verify processors with tightly coupled accelerators. Software models for processors are primarily written in C or C variants. As porting those models into Matlab is an unrealistic task for most design teams, the use of HDL Verifier for processors with tightly coupled accelerators is unlikely.

Xilinx also produces two tools for co-simulation, Vitis HLS [5] and ISim [6]. Vitis HLS is most similar to HDL Verifier, pairing a C-based software model with an RTL simulation. The co-simulation is executed in three phases. First, the software model generates stimulus vectors to be fed into the RTL simulation. Then, the RTL simulation is run on the provided stimulus, and its output is captured. Finally, the captured output is passed to the software model for post-test correctness checking. This architecture is well suited for confirming that a digital design and a C-based functional model produce identical results on the same stimulus.

The largest drawback of Vitis HLS is that the RTL designs used in co-simulation must be purely combinational. This makes it impossible to test any complex design with state elements. Moreover, since Vitis HLS generates stimulus entirely in the first phase of execution, it is also impossible to test any design that makes requests back to the model. With these constraints in mind, Vitis HLS is best suited for testing simple and purely combinational designs at a unit level.

The second co-simulation tool produced by Xilinx is ISim, which targets mixed-medium verification. ISim takes RTL for a design, and based on user specifications splits it into a software simulation and FPGA emulation. The main benefit of ISim is that the entire design is tested with RTL fidelity. Traditional software-only simulations at RTL accuracy are typically very slow. However, because ISim places some components onto a FPGA, large designs can be accurately and quickly tested. Xilinx specifically denotes that this tool is designed to accelerate the testing of large DSP IPs, which can be incredibly slow to simulate in pure RTL simulations.

While ISim provides the ability to simulate more complex designs than the two previously discussed tools, it too has a set of downsides. The largest downside is that using ISim requires access to a Xilinx FPGA. While this may not be a problem for industry teams, it can limit those without similar resources. The second, smaller, downside is that ISim performs RTL simulation for all components of the design. If the components simulated in software are not of interest, it would be significantly faster to replace them with purely functional C-based simulations that communicate with the FPGA emulated RTL.

1.2.2 Open Source Frameworks

In contrast to the commercial frameworks, open-source frameworks are produced by a wide variety of entities in the commercial and research space. These frameworks often lack features found in their commercial counterparts, but are free for public use. The following section discusses actively developed open-source co-simulation frameworks. Due to their open-source nature, these frameworks vary in their maturity and level of development.

FireSim [7] is an open source tool that allows for a Rocket Chip [8] or BOOM [9] based system on a chip (SoC) to be simulated on AWS EC2 F1 FPGA instances. One use of the tool is to simulate a single core and attached periphery across a cluster of compute nodes. This results in simulations that can run in the hundreds of MHz, a substantial speedup over pure software RTL simulation.

Moreover, FireSim allows components to be modeled purely in software, which creates the ability for it to be used as a co-simulation framework. However, this has not been done in practice. Instead, software models are used to model far-away periphery such as disk memory, where RTL does not exist.

The limitations of FireSim with regards to co-simulation are twofold. First, it requires access to cloud computing resources. While these resources greatly enhance simulation speeds, they may be unnecessarily powerful depending on the size of the component targeted for simulation. Second, the design of FireSim prevents a core from being modeled as a software component. As was highlighted when discussing ISim, in cases where the core is not the component of interest, it would be preferable to replace it with a C-based functional model rather than spending compute cycles emulating it at RTL fidelity.

Another open source framework is Dromajo [10], produced by Esperanto Technologies. Dromajo is a C-based RISC-V RV64GC emulator designed for RTL co-simulation. The emulator executes programs in software simulation and generates checkpoints at user defined stopping points. These checkpoints can then be passed to a compatible RTL simulation. At this point, both Dromajo and the RTL simulation can resume execution from the checkpoint simultaneously, and their traces can be compared to validate RTL correctness. Currently, Dromajo has only been integrated for use with BOOM.

Dromajo ultimately is limited to checking an RTL emulation against a software golden model. As previously noted, this is a form of tandem simulation rather than true co-simulation. Accordingly, Dromajo does does not allow for coherent and simultaneous modeling of distinct components to build a larger system. This makes Dromajo effective for core validation, but limited when attempting to validate periphery.

The last open-source co-simulation framework of interest is proposed by Muñoz-Quijada et al. in their paper, *SW-VHDL Co-Verification Environment Using Open Source Tools* [11]. In this framework, a pure software model is paired with an FPGA based RTL emulation to create a coherent system. This pairing is accomplished by transmitting commands between software and hardware over Unix named pipes. The major outcome of the proposed framework is that workloads can run in co-simulation without modification, enabling a faster testing flow. However, as with previous described mixed-medium and FPGA based co-simulation environments, requiring FPGA based emulation limits the applications of this framework. Specifically, it creates a requirement that a designer have FPGA access to engage in co-simulation. For those with access, co-simulation can only be applied to designs that fit on a single board.

1.3 What CFC Brings to the Table

CFC contributes to the space of open-source co-simulation frameworks by targeting SoC level accelerator integration testing. In the Chipyard [12] environment that this framework targets, this testing is performed with a full-SoC RTL simulation running compiled C-based test binaries. This process is often slow and a barrier to rapid design space exploration. Depending on the size and complexity of the SoC, elaborating a simulator can take anywhere from 15 minutes to several hours.

After the simulator is built, running a simulation binary adds further delay on the order of tens of minutes. This delay is exacerbated if verifying correctness for randomly generated test vectors requires computationally complex golden model calculations to be performed on the simulated SoC. Finally, if a single design change is made, the entire SoC simulation must be rebuilt, which is needlessly wasteful given that a large majority of the design has remained constant.

The critical insight that motivates the development of CFC is as follows. While the entire SoC is built into an RTL simulation during accelerator integration testing, the only component that actually requires a simulation with RTL fidelity is the accelerator itself. This means that the simulation method used for the core within the SoC is irrelevant so long as the proper signals can be passed to the accelerator's RTL simulation. Accordingly, CFC creates a set of rules and tools that allow a software simulation of a core — which is magnitudes faster at golden model computations — to be paired with an RTL simulation of an accelerator. These coupled components form a coherent SoC co-simulation. This framework is most similar to that proposed by Muñoz-Quijada et al., but is Chisel oriented and does not require access to FPGAs.

CFC is further differentiated from existing SoC-level verification methods due to the introduction of simulator binary caching. In short, the optimization allows an existing simulation binary to be reused if the underlying Chisel code has not changed between test runs. This is contrasted with the existing ChiselTest paradigm wherein a simulation binary is re-built for every test run. As a result, CFC uses the same number of simulator builds as a full-SoC simulation approach, while offering significant speedups in both build and run times.

An additional benefit of CFC is its high level of integration with Chisel-based designs. The components of CFC that interact with HDL devices are designed entirely in Scala and Chisel. Due to this, co-simulation can be run from within `sbt` [13]. In contrast, using an existing tool would require the elaboration of a Chisel design into Verilog and the porting of that design into a co-simulation framework. This process would need to be repeated every time a design change was made.

Beyond accelerating the exploration of RTL changes internal to an accelerator, CFC components can also be used to alter the diplomatic parameters of accelerators. This simulates the effect of elaborating an accelerator in different SoC environments without needing to have the core RTL for that environment.

Finally, CFC is expandable. The design elements and application of CFC discussed in this thesis are primarily targeted towards Rocket Custom Co-processor (RoCC) [8] accelerators and the Spike RISC-V ISA simulator. However, any devices or models that implement CFC's communication interface and rules could be orchestrated in co-simulation. For example, the TileLink (TL) components of CFC could be used to develop a co-simulation testbench for a MMIO based accelerator. This expandability and CFC's open-source nature creates a wide range of potential future applications for the framework.

It should be noted that FireSim has also been used to test accelerators in development. In Maas et al.'s work, *A Hardware Accelerator for Tracing Garbage Collection* [14], FireSim was used to emulate a SoC containing a core, memory system, and garbage collection accelerator. That emulation was then used to explore the performance impact of altering the accelerator's internal queue sizes and compression techniques.

Similarly, in Qijing Huang et al.'s work, *Centrifuge: Evaluating full-system HLS-generated heterogenous-accelerator SoCs using FPGA-Acceleration* [15], FireSim was used as part of a flow for the rapid generation and evaluation of SoCs. The Centrifuge flow generated accelerators to replace specific software functions using high-level synthesis and modeled their attachment to a core at one of three different locations using FireSim. This flow was then used to find the optimal SoC for a given software workload via a design space sweep that determined which software elements to replace with accelerators and where to attach those accelerators.

CFC is not designed to supplant either of these characterizations. Rather, it is intended to provide an intermediate testing space that does not require access to cloud computing resources. Once a flow such as Centrifuge has been used to determine an optimal SoC configuration, CFC can be used to rapidly design and test accelerators with SoC-level workloads. When this testing is complete, there is still the need for a FireSim-type cycle-accurate device simulation in order to comprehensively characterize performance. In short, CFC is designed to speed up accelerator design and SoC-level debugging. It is not designed to determine which workloads to accelerate or to act as a performance model.

2 Background

Designing a co-simulation framework for Chisel requires not only the creation of new infrastructure, but also the combination of many existing tools and components. This section provides an overview of such tools and components that are used within CFC. A similar overview is also provided for the model and accelerator used in Section 4 of this thesis.

2.1 Chisel

Chisel [16] is an HDL embedded into the Scala programming language. The key advantage of using Chisel for co-simulation is that Scala-based software elements can interact with Chisel-based hardware elements without the need for a DPI-type connection. For example, a Scala object can be created that reads data from an input stream, performs a software function on that data, and then pokes the result into the IO on a Chisel-based hardware module.

2.2 ChiselTest

ChiselTest [17] is a library that supports the creation of test harness capabilities for Chisel-based hardware designs. These harnesses add the ability to step a design's clock while poking data into wires or peek at the values on wires. CFC is designed to work within a ChiselTest testing environment, as it relies on ChiselTest threading and testbench utilities. Additionally, ChiselTest forms the backbone of drivers and monitors from the chisel verification repository that are used within CFC.

2.3 Chipyard

Although CFC can be operated independently from Rocket Chip, the components used for testing in this thesis are from the Rocket Chip ecosystem. To make testing these components a more streamlined process, CFC is designed to be used as a tool within Chipyard, a chip development framework that unifies Rocket Core and related accelerators into a single environment. In Chipyard, CFC is integrated as a sub-project of the verification repository.

2.4 Gemmini

Gemmini [18] is an open-source, Chisel-based hardware generator for matrix multiplication accelerators developed at UC Berkeley. For a given configuration, the generator creates a RoCC based accelerator that is controlled using a non-standard RISC-V custom extension. The accelerator — simply referred to as Gemmini — receives commands from the RoCC interface of a Rocket Chip-based tile and interfaces with memory through a direct TL connection to the L2 cache.

2.5 Spike

Spike [19] is a C-base RISC-V ISA simulator that implements a functional model of RISC-V cores. Being a functional model, Spike provides an instruction-accurate representation of RISC-V executing on a core. This behavior is not clocked, so each instruction seen by the simulator is fully executed before another instruction is processed. To aid in SoC simulation, Spike supports the addition of functional coprocessor models, which emulate RoCC based accelerators. For example, there is a Spike extension for Gemmini that is able to process custom instructions from the Gemmini ISA.

Spike runs standard RISC-V binaries without modification. This includes binaries that contain custom instructions as long as the proper extension is added using the `--extension` command line argument.

Several modifications were made to Spike and the Gemmini Spike extension as part of CFC in order to enable co-simulation. These changes are discussed further in Section 3 and Section 4 of this thesis.

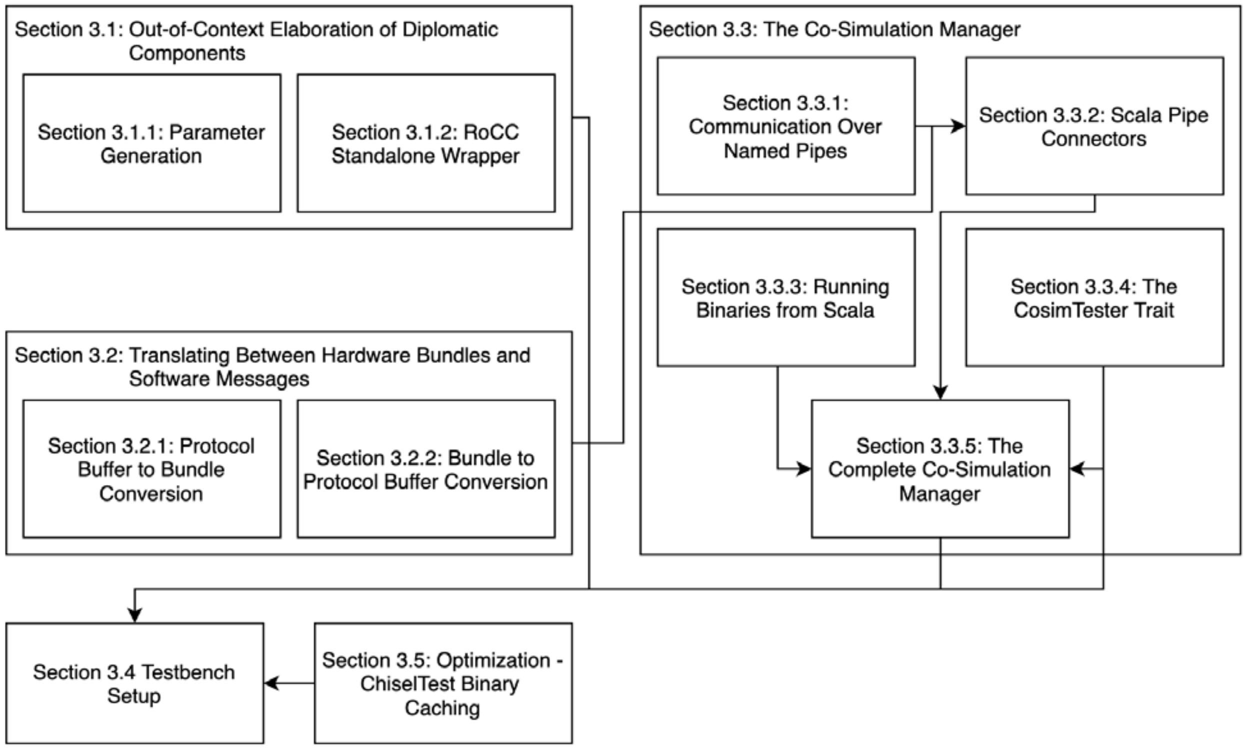
2.6 Protocol Buffers

The backbone of CFC's inter-simulation communication is Google's protocol buffers (protobufs) [20], a language-neutral mechanism for quickly serializing structured data. After writing a structural description, the protoc compiler can be used to generate code for a variety of supported languages. This generated code provides key communication features such as setting or getting message fields, serializing messages to bits, and creating new message objects from input stream data. Since the code for each language is generated from a common structure, every system that uses the generated code is able to communicate in a common language.

3 Design

Several new pieces of infrastructure were created during the development of CFC. These include test components to enable out-of-context elaboration for diplomatic components, a communication agreement to synchronize inter-model communication over named pipes, utilities to convert between information on hardware wires and protocol buffers, and a manager object that stages the entire co-simulation flow.

Figure 1 shows the components produced for CFC organized by their corresponding section in this thesis. It also shows how these components come together to create larger objects via the arrows linking boxes. For example, the arrow from Section 3.1 to Section 3.4 indicates that the utilities developed to enable out-of-context elaboration will be used during testbench setup.



Section 3.1: Out-of-Context Elaboration of Diplomatic Components
Section 3.1.1: Parameter Generation
Section 3.1.2: RoCC Standalone Wrapper
Section 3.2: Translating Between Hardware Bundles and Software Messages
Section 3.2.1: Protocol Buffer to Bundle Conversion
Section 3.2.2: Bundle to Protocol Buffer Conversion
Section 3.3: The Co-Simulation Manager
Section 3.3.1: Communication Over Named Pipes
Section 3.3.2: Scala Pipe Connectors
Section 3.3.3: Running Binaries from Scala
Section 3.3.4: The CosimTester Trait
Section 3.3.5: The Complete Co-Simulation Manager
Section 3.4 Testbench Setup
Section 3.5: Optimization - ChiselTest Binary Caching

Figure 1: CFC components discussed in each section and where they are used.

3.1 Out-of-Context Elaboration of Diplomatic Components

As previously mentioned, a key design aspect of CFC is that devices under test (DUTs) can be tested without the need to elaborate an entire SoC (out-of-context). For the accelerators tested in this thesis, this meant that CFC had to be designed to elaborate tightly coupled RoCC components without an attached Rocket Chip tile. This independence required extra engineering work to achieve. Specifically, any Rocket Chip peripheral elaborated independently from a Rocket Chip core must address both Diplomacy [21] and implicit parameters.

Diplomacy is a framework that allows a set of interconnected devices to negotiate the parameterization of their connection protocol. For each set of devices and their shared bus, the framework cross-checks requirements, negotiates free parameters, and supplies final parameter bindings for each device to use in the hardware generation process. When elaborating out-of-context, there is no set of devices to reference for the negotiation of parameters. Despite this, the generated hardware must elaborate such that it is compatible with the bus interface of the core. To achieve this result, a dummy set of nodes must be created to mimic the endpoints found in the overall system. Connecting these artificial endpoints to the out-of-context device allows for proper negotiation to occur.

The other key to correct out-of-context elaboration is ensuring that the implicit parameters needed by a device are both present and set to the correct values. In Rocket Chip-based designs, parameters are typically passed in a `Parameters` object. This object acts as a key-value mapping between specific parameters and their values. Harnessing a language-specific feature of Scala, the parameters object is most often passed as an implicit argument (typically `p`). This means that if there is some `Parameters` named `p` declared in the scope of a function or generator call, it is automatically supplied as an argument to that call.

3.1.1 Parameter Generation

In order to address the needs of Diplomacy and provide implicit parameters, CFC contains a parameter generation utility function, `getVerifParameters`. It should be noted that because this thesis focuses on co-simulation for RoCC devices, the parameter generation utilities are targeted towards RoCC devices. Future work would be needed to augment these utilities to allow for the elaboration a non-RoCC but Rocket-related device out-of-context. Furthermore, `getVerifParameters` needs to take in arguments to guarantee that the parameters it returns match what would be seen during an in-context elaboration. As these parameters are critical for proper device elaboration, future work could be done to allow for the caching and reuse of these parameters from in-context elaboration runs.

The `getVerifParameters` function works by taking the parameters from a minimal Rocket configuration and augmenting them with key parameters typically set in a full system during elaboration. Some of these parameters, such as `XLen`, `beatBytes`, and `blockBytes`, must be specified by the user. Altering these arguments alters the system that the DUT elaborates for.

Other parameters are set with non-configurable values. For example, the `TileKey` parameter is mapped to a minimal set of `TileParams` that define a fake "verif_tile" with a `hartId` of 0, as well as default `RocketCoreParams`, `DCacheParams`, and `ICacheParams`. Moreover, since the `hartId` of this tile is 0 and there will be no other defined tiles, the `MaxHartIdBits` parameter is set to 1.

The section of code from `getVerifParameters` where all the mentioned parameters are set is shown in Listing 1.

The second objective of `getVerifParameters` is to ensure that the environment has the necessary information for Diplomacy to successfully negotiate the bus parameters for an out-of-context device. During in-context elaboration, this negotiation is performed using each

```
    val origParams = new VerifBaseRocketConfig

    // Augment the parameters
    implicit val p = origParams.alterPartial {
      case MonitorsEnabled => false
      case TileKey => VerifTileParams
      case XLen => xLen
      case PgLevels => if (xLen == 64) 3 else 2
      case MaxHartIdBits => 1
      case SystemBusKey => SystemBusParams(
        beatBytes = beatBytes,
        blockBytes = blockBytes
      )
    }

```
CosimTestUtils.scala

Listing 1: Section of getVerifParameters used to set key parameters.

tile's TL ephemeral node, which is mapped to by the TileVisibilityNodeKey parameter. This TL ephemeral node attaches to the master TL crossbar, which in turn is attached to the core and other peripheral devices. Due to this attachment structure, Diplomacy negotiates bus parameters by looking for a parameterization that allows the TL ephemeral node to communicate to all other devices on the crossbar. The connection structure for an in-context elaboration run is shown in Figure 2.

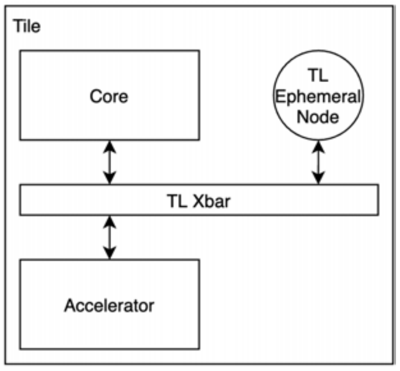


Figure 2: Connections established during typical elaboration.

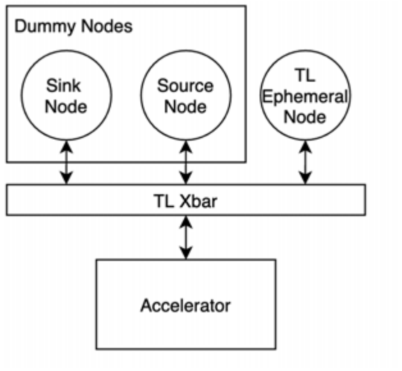


Figure 3: Connections established during out-of-context elaboration.

When elaborating a RoCC device out-of-context, there is no core to attach to the TL crossbar. This means that the TL ephemeral node is not constrained by the core's TL requirements, which could lead to a different set of negotiated parameters. To address these missing constraints and to set the TL parameters for an out-of-context device, getVerifParameters connects a set of dummy nodes to the TL crossbar based on user-supplied arguments. Ideally, these arguments will be set to match the TL parameters negotiated during an in-context elaboration.

Given a set of arguments, getVerifParameters creates a set of TLBundleParameters. These are in turn used to create source and sink BundleBridges, which are converted into dummy TL nodes using the BundleBridgeToTL method. During this process the pAddrBits parameter — which sets the bitwidth of memory addresses — can be modified based on one of the user-supplied arguments. This is accomplished by adjusting the number of bits in the address-set mask for the dummy source node connection.

Since there is no tile present during an out-of-context elaboration, getVerifParameters creates a new TL ephemeral node and connects it and the dummy nodes to a new TL crossbar. Finally, to ensure that the device being elaborated sees these connections, the created TL ephemeral node is set as the TileVisibilityNodeKey in the parameters object. During the elaboration flow, this will result in the connection structure shown in Figure 3.

The section of getVerifParameters used to set diplomatic parameters is shown in Listing 2.

```scala
  def verifTLUBundleParams: TLBundleParameters = TLBundleParameters(addressBits = 64, dataBits = 64,
↪  sourceBits = 1,
    sinkBits = 1, sizeBits = 6,
    echoFields = Seq(), requestFields = Seq(), responseFields = Seq(),
    hasBCE = false)

  val dummyInNode = BundleBridgeSource(() => TLBundle(verifTLUBundleParams))
  val dummyOutNode = BundleBridgeSink[TLBundle]()

  val tlMasterXbar = LazyModule(new TLXbar)
  val visibilityNode = TLEphemeralNode()(ValName("tile_master"))

  visibilityNode :=* tlMasterXbar.node
  tlMasterXbar.node :=
    BundleBridgeToTL(getVerifTLMasterPortParameters) :=
    dummyInNode

  dummyOutNode :=
    TLToBundleBridge(getVerifTLSlavePortParameters(beatBytes, pAddrBits, transferSize)) :=
    visibilityNode

  val outParams = p.alterPartial {
    case TileVisibilityNodeKey => visibilityNode
  }

```
CosimTestUtils.scala

Listing 2: Section of getVerifParameters used to set diplomatic parameters.

3.1.2 RoCC Standalone Wrapper

With a set of dummy parameters generated, the final component needed for out-of-context elaboration is a wrapper module for elaborated DUTs. This module takes in a function that returns a LazyRoCC module and exposes its ports to the testbench. The need for this wrapper module is twofold. First, the TL node in RoCC modules is an identity node. This means that the node has a cardinality constraint on its sinks and sources. This can create a problem during out-of-context elaboration, as different DUTs may have a different number of internal connections. The standalone wrapper addresses this by allowing the number of sources and sinks to be specified by the user. For each specified source, a new BundleBridgeSource is created and connected to the LazyRoCC's TL node. The same pattern holds for each specified sink, except a BundleBridgeSink is used for each connection. When the module is elaborated, the IOs from those new sources and sinks are constructed and then exposed to the testbench via the wrapper.

The second reason for needing a wrapper is that the elaboration process can automatically optimize away wires that have no connections. In the case of the `RoCCIO` found in `LazyRoCC` modules, if there is a wire inside the IO that does not have a connection path to the test harness, there is no guarantee that that wire will exist after elaboration. The optimization away of wires can be avoided by wrapping an IO with a `dontTouch` directive. However, this would require modification to the internal HDL of a DUT to support co-simulation. To address this, the entire `RoCCIO` of the `LazyRoCC` module inside the wrapper is tied to a `dontTouch` IO port in the wrapper, exposing it to the testbench and ensuring that it does not get optimized away. The full implementation of `VerifRoCCStandaloneWrapper` is shown in Listing 3.

```scala
class VerifRoCCStandaloneWrapper(dut: () => LazyRoCC, beatBytes: Int = 8, addSinks: Int = 0, addSources: Int
↪  = 0)(implicit p: Parameters) extends LazyModule {
  def verifTLUBundleParams: TLBundleParameters = TLBundleParameters(addressBits = 64, dataBits = 64,
  ↪  sourceBits = 1,
    sinkBits = 1, sizeBits = 6,
    echoFields = Seq(), requestFields = Seq(), responseFields = Seq(),
    hasBCE = false)

    lazy val ioOutNodes = new MutableList[BundleBridgeSink[TLBundle]]
    lazy val ioInNodes = new MutableList[BundleBridgeSource[TLBundle]]
    val dutInside = LazyModule(dut())

  for (i <- 0 until addSinks) {
    ioOutNodes += BundleBridgeSink[TLBundle]()
    ioOutNodes(i) :=
      TLToBundleBridge(TLManagerPortParameters(Seq(TLManagerParameters(address = Seq(AddressSet(0x0, 0xfff)),
        supportsGet = TransferSizes(1, 64), supportsPutFull = TransferSizes(1,64), supportsPutPartial =
        ↪  TransferSizes(1,64))), beatBytes)) :=
      dutInside.tlNode
  }

  for (i <- 0 until addSources) {
    ioInNodes += BundleBridgeSource(() => TLBundle(verifTLUBundleParams))
    dutInside.tlNode :=
      BundleBridgeToTL(TLClientPortParameters(Seq(TLClientParameters("bundleBridgeToTL")))) :=
      ioInNodes(i)
  }

  lazy val module = new VerifRoCCStandaloneWrapperModule(this)
}

class VerifRoCCStandaloneWrapperModule(outer: VerifRoCCStandaloneWrapper) extends LazyModuleImp(outer) {
  import outer.dutInside
  import outer.ioInNodes
  import outer.ioOutNodes

  val io = dontTouch(IO(new RoCCIO(dutInside.nPTWPorts)))
  io <> dutInside.module.io

  val tlOut = ioOutNodes.map{ (outNode) => outNode.makeIO()}
  val tlIn = ioInNodes.map{ (inNode) => inNode.makeIO()}

}
```
VerifRoCCStandalone.scala

Listing 3: Implementation of `VerifRoCCStandaloneWrapper`.

3.2 Translating Between Hardware Bundles and Software Messages

In CFC protobufs are used to carry information between the Scala-based test harness for a device and the software model representing the rest of the system. While this communication system is effective when transferring information between two pieces of software, additional translation is needed to convert the software-based key-value pairs of a protobuf into a packet of hardware information that can be driven into a DUT. Additionally, many DUTs require bi-directional communication with the software model, so the conversion needs to occur in both directions: from software to hardware for testbench drivers and from hardware to software for testbench monitors. To aid in this process, CFC introduces a set of utility functions that allow for the mapping of protobufs to Chisel bundle literals, and vice-versa.

3.2.1 Protocol Buffer to Bundle Conversion

The first step towards creating a protobuf to bundle conversion is to define a library of helper functions that allow for the creation of bundle literals with specific values. For a given bundle type, there should exist one such helper that turns specified values into a Chisel bundle literal. To aid down the line in the translation process, all such helpers for a given category of bundle type — such as TL bundles — should exist within the same object. An example of such a helper function for the `RoCCCommand` bundle is shown in Listing 4. This helper is contained within the `VerifRoCCUtils` object, which also contains a helper for the `RoCCInstruction` bundle.

```scala
  def RoCCCommandHelper(inst: RoCCInstruction = new RoCCInstruction, rs1: UInt = 0.U, rs2: UInt =
  ↪  0.U)(implicit p: Parameters): RoCCCommand = {
    new RoCCCommand().Lit(_.inst -> inst, _.rs1 -> rs1, _.rs2 -> rs2, _.status -> MStatusHelper(dprv = 3.U,
    ↪  prv = 3.U))
  }
```
CosimTestUtils.scala

Listing 4: Implementation of `RoCCCommandHelper`.

Using these bundle literal helpers, the naive solution for protobuf to bundle literal conversion is to define a new conversion function for each bundle type. This converter would accept a protobuf message matching the target type and get all the values contained in that protobuf, or a default if the value does not exist. Those values could then be converted to the corresponding Chisel literals and used as arguments to the bundle literal creation helper.

While this approach is effective, there are a few key downsides. The primary drawback is that the process is time-consuming and repetitive. For each bundle conversion, a user has to create a new conversion function that maps values to literals and then feeds those literals into a helper function. In the extreme case where a bundle has hundreds of fields, this could correspond to a thousand or more of lines of code. Additionally, the process is not robust to changes in the bundle. Ideally, if a bundle's definition changes, a user should only have to change the bundle literal helper and the matching protobuf without having to make changes in the converter. However, under the naive solution, adding a new field into a bundle requires changes in three places: the bundle literal creation helper, the protobuf declaration, and the converter.

Taking note of the repetitive key to key mapping employed by the naive solution, CFC harnesses the power of Scala and introduces a generic protobuf to bundle conversion function: `ProtoToBundle`. Currently, this function is only compatible with the `RoCCCommand`, `RoCCInstruction`, `TLBundleA`, and `TLBundleD` bundle types. However, increasing compatibility is as simple as defining a bundle helper and corresponding protobuf for a chosen bundle type.

`ProtoToBundle` takes in three arguments: 1) a protobuf message 2) an object wrapping a set of bundle literal creation helper functions (the helper library), and 3) the target return type. The requirements of `ProtoToBundle` with regards to these arguments are as follows. First, the helper library must contain all the bundle literal helpers needed to create a bundle literal for the message that is passed in. In the case of a message the contains sub-messages, bundle literal helpers matching the those sub-message types must also be contained within that same helper library. Second, there must be a one-to-one mapping from the name of a protobuf message or sub-message to the name of a bundle helper. This mapping is performed by checking if a bundle helper name contains the name of a protobuf message. For example the helper `RoCCCommandHelper` is a valid name for the a protobuf named `RoCCCommand`. Finally, the protobuf message and bundle literal must have the same fields, and those fields must have identical names and be of compatible types.

Assuming all these requirements are met, `ProtoToBundle` creates a new bundle literal from a protobuf message using the following algorithm. For readability, the algorithm is broken down into numbered steps accompanied by relevant code segments.

1. `ProtoToBundle` gets the name of the passed in protobuf (e.g. `RoCCCommand`) and searches the helper library for the matching bundle literal creation function. This bundle literal creation function is obtained using Scala reflections, so it is of the generic type `MethodSymbol`.

```scala
    val protoName = proto.getDescriptorForType.getName
    val helper = getHelper(helpers, protoName)
```
CosimTestUtils.scala

Listing 5: Search for matching bundle literal creation function.

2. The public `ProtoToBundle` calls a private variant of `ProtoToBundle` that takes takes in five arguments: 1) a protobuf message 2) a `MethodSymbol`, 3) a `TypeTag`, 4) a `ClassTag`, and 5) a return type. The message and return type arguments are passed through from the public function arguments, the `TypeTag` and `ClassTag` are derived from the helper library object, and the `MethodSymbol` is the found bundle literal creation function.

```scala
    ProtoToBundle(proto, helper, helpers, helpers, returnType)
```
CosimTestUtils.scala

Listing 6: Call to private `ProtoToBundle`.

3. In the private `ProtoToBundle`, the fields defined in the protobuf message are converted into key-values pairs. Those pairs are then iterated through and used to create a mapping

of arguments to Chisel literal types. How that Chisel literal is created depends on the type of the field within the protobuf. Bools are created directly using the corresponding type to literal function. Numerical values larger than 64-bits are stored as hex strings, so they must be converted into BigInts and which in turn are converted into literals. Numerical values under 64 bits require more care in the conversion process to BigInt, as protobufs encode numerical values using Java Integers or Doubles, both of which are signed. To avoid the field values being interpreted as signed and becoming negative, the values are shifted left to knock off the sign bit, and then have the sign bit reintroduced after conversion. Finally, if the field is a sub-message, a recursive call is needed to create a bundle literal for this sub-message. In this case, the name of the sub-message is used to find a corresponding bundle literal helper for the sub-message type, and the private ProtoToBundle is called again recursively. This recursive structure is the motivation for including all related bundle literal helpers inside the same library.

```scala
    // Extract fields from the protobuf
    val protoArgs = collection.mutable.Map[String, Any]()
    proto.getAllFields.foreach { case (field, value) =>
      val fieldName = field.getName
      field.getType match {
        case MESSAGE =>
          // Recursive case needs some extra spice
          val subProto = value.asInstanceOf[com.google.protobuf.Message]
          val subProtoName = subProto.getDescriptorForType.getName
          val subHelper = getHelper(helperType, subProtoName)
          protoArgs += (fieldName -> ProtoToBundle(subProto, subHelper, helperType, helperClass,
          ↪  subHelper.returnType))
        case UINT32 => protoArgs += (fieldName -> fromBigIntToLiteral((BigInt(value.asInstanceOf[Integer] >>>
        ↪  1) << 1) + (value.asInstanceOf[Integer] & 1)).asUInt)
        case UINT64 => protoArgs += (fieldName -> fromBigIntToLiteral((BigInt(value.asInstanceOf[Long] >>> 1)
        ↪  << 1) + (value.asInstanceOf[Long] & 1)).asUInt)
        case BOOL => protoArgs += (fieldName -> fromBooleanToLiteral(value.asInstanceOf[Boolean]).asBool)
        case STRING => protoArgs += (fieldName -> fromBigIntToLiteral(BigInt(value.asInstanceOf[String],
        ↪  16)).asUInt)
      }
    }
```

CosimTestUtils.scala

Listing 7: Conversion of protobuf fields into key-value pairs.

5. Once conversion for every field in the protobuf message is completed, the resulting values need to be mapped to the bundle literal converter arguments. To accomplish this, the private helper once again uses Scala reflections to gather a list of the bundle literal converter arguments. These arguments are then used to generate a zip of argument names and types. A zip is used instead of a map due to the fact that zips preserve ordering. This is important because the order in which arguments appear in the initial list is the same order in which they need to be applied in the bundle literal helper. To produce the final argument list, the zip goes through a mapping process wherein the value in the protobuf's argument map is used if the argument name is present, else a default value is used. This name-based mapping motivates the requirement for an exact match between the names of the protobuf message field and the bundle literal fields. What is left after this process is an in-order list of the arguments to the bundle literal helper method where those arguments all have the type expected by the helper.

```
    // Get a zip of (argument -> type) for our helper function.
    // Fill undeclared values with 0s and set the implicit parameters to p
    val args = getArgsZip(getArgs(helper))
      .map{ case (arg, typ) =>
        protoArgs.getOrElse(arg, // Match arguments list to the values from the proto if present, else get a
        ↪  default
          typ match {
            case t if t =:= typeOf[chisel3.UInt] => 0.U
            case t if t =:= typeOf[chisel3.Bool] => false.B
            case t if t =:= typeOf[freechips.rocketchip.config.Parameters] => p
          })
      }
```
CosimTestUtils.scala

Listing 8: Production of the final arguments list.

6. The final step in the conversion process is to produce the bundle literal. Because the bundle converter is of type `MethodSymbol`, its apply function accepts arguments as a variable length list of `Symbol` objects. The types of these arguments and the length of the supplied list are validated during the application process. Thus, the previously generated list of in-order arguments can be decomposed into a variable length sequence and applied. The resulting value is `Any` type, so it is cast to the target return type before being returned.

```
    currentMirror
      .reflect(helperClass)
      .reflectMethod(helper)
      .apply(args.toList: _*)
      .asInstanceOf[R]
```
CosimTestUtils.scala

Listing 9: Cast to target return type.

Through this lengthy process, the generic protobuf message to bundle converter addresses the concerns raised by the naive conversion solution. A user can convert between equivalent software and hardware messages without having to manually define a conversion function. Moreover, the helper is agnostic to bundle structure changes as the supporting bundle literal helpers are updated.

It is important to note that this function currently only supports conversion for messages containing `UINT32`, `UINT64`, `BOOL`, `STRING`, and `MESSAGE` fields. This should not be a limiting factor, as including more types is as simple as adding more case statements in the helper function.

3.2.2 Bundle to Protocol Buffer Conversion

While the process of converting protobufs to bundle literals is highly complex, the inverse process, converting bundle literals to protobufs, is significantly simpler. This conversion is done using the Json parser built into protobuf. Given a protobuf message builder, the parser takes a Json formatted string and automatically converts it into the builder's target message type. Effectively, this reduces the bundle to protobuf conversion to a bundle to Json conversion.

The bundle to Json conversion is done by iterating recursively over a given bundle. For each (name, value) pair in the bundle's elements, a string is generated by pairing the name with a string generated based on the value type. Bundle types undergo a recursive call to generate their value strings. Other types are converted to their Java literal values, which are then converted to strings using the default `toString` method. A special case exits for numerical values wider than 64 bits, which are converted into hex strings.

With a bundle to Json converter implemented, conversion from bundle to protobuf is as simple as first converting the bundle to Json, and then calling the protobuf Json parser on the output. The implementations for both converters are shown in Listing 10.

```scala
  def BundleToProto[B <: Bundle](bundle: B, builder: com.google.protobuf.Message.Builder): com.google.protobuf.Message = {
    com.google.protobuf.util.JsonFormat.parser().ignoringUnknownFields().merge("{" + BundleToJson(bundle) + "}", builder)
    builder.build()
  }

  def BundleToJson[B <: Bundle](bundle: B): String = {
    bundle.elements.map { case (name, value) =>
      value match {
        case _: Bundle => s""" "${name}": { ${BundleToJson(value.asInstanceOf[Bundle])} }"""
        case _: Bool => s""""${name}": ${value.asInstanceOf[Bool].litToBoolean}"""
        case _: UInt => value.asInstanceOf[UInt].getWidth match { // Values wider than 64.W are stored as hex strings
          case x if x > 64 => s""""${name}": "${String.format(s"%${math.ceil(x.toDouble/4)}s", value.litValue.toString(16).toUpperCase).replace(" ", "0")}""""
          case _ => s""""${name}": ${value.litValue}"""
        }
        case _ => s""""${name}": ${value.litValue}"""
      }
    }.mkString(",\n")
  }
}
```
CosimTestUtils.scala

Listing 10: Implementation of `BundleToProto` and `BundleToJson`.

3.3 The Co-Simulation Manager

The co-simulation manager is the tool within CFC that controls the various simulations and orchestrates their communication. The following section details how the co-simulation manager and its sub-components are built.

3.3.1 Communication Over Named Pipes

CFC uses named pipes for interprocess communication (IPC). Although this type of communication is often done using sockets, Scala-based socket servers do not reliably terminate when a test fails within `ScalaTest`. When these failures happen, a complete restart of `sbt` is needed to ensure that the server instance is deleted. In contrast, named pipes act as system files. This means that named pipes can be created, accessed, and deleted in both C and Scala by using standard APIs.

However using named pipes introduces a new set of challenges when trying to manage coherency in IPC. The largest challenge is that named pipes block until a connection is made on both sides. This means that if one process connects to some pipe as a reader, that process will stall until another process connects to the same pipe as a writer. The obvious hazard that arises because of this is deadlock – if two processes try to communicate over named pipes but do not form connections in the same order, they will both stall indefinitely.

In order to avoid this problem, CFC defines a standard order for creating, connecting to, and deleting named pipes. By adhering to this order, it is ensured that neither process involved in co-simulation will stall on file operations. A summary of this standard is shown in Table 1. In this table "C Action" refers to the SoC simulator and "Scala Action" refers to the ChiselTesters-based co-simulation testbench.

Phase	C Action	Scala Action
1	Open Named Pipes	None
2	Open Input Streams	Open Output Streams
3	Open Output Streams	Open Input Streams
4	None	Delete Named Pipes

Table 1: CFC named pipe management phases.

The ordering of pipe operations can be broken down into four phases. In the first phase the SoC simulator is responsible for opening every pipe at the start of simulation execution. In C, this is done using the `mkfifo` command. In the second phase, the SoC simulation will open its input streams. At the same time, the co-simulation testbench is responsible for opening its output streams. Once this step is completed the next phase starts and the SoC simulation will open its output streams while the co-simulation testbench opens the corresponding input streams. In both of these phases, if multiple pipes are interacted with sequentially, then the interactions must be done in the same order on both the SoC simulation and in the co-simulation testbench. Finally, once the testbench has finished running, it is responsible for deleting all of the created named pipes. A graphical representation of these phases is shown in Figure 4.

The other challenge introduced by named pipes is internal buffering and its impact on short messages. In both C and Scala, stream writers contain an internal buffer that only writes out to the target file on closure or when a certain internal capacity is reached. For typical applications, where writers are writing long messages to static files, this buffering can increase software performance. However, in the context of CFC, where the files being written to are intended for real-time information transfer and the messages being sent are often very small, this buffering behavior can result in messages not being written into a pipe during the entirety of a co-simulation run.

In C, this issue is avoided by disabling buffering on output streams. This is done by calling `ostream.rdbuf()->pubsetbuf(0, 0)`, where the `ostream` is of type `std::fstream`. In Scala, the issue is less easily remedied, as there is no direct function call to disable buffering on `io.outputstream` objects. Moreover, calling `flush()` on an output stream object flushes

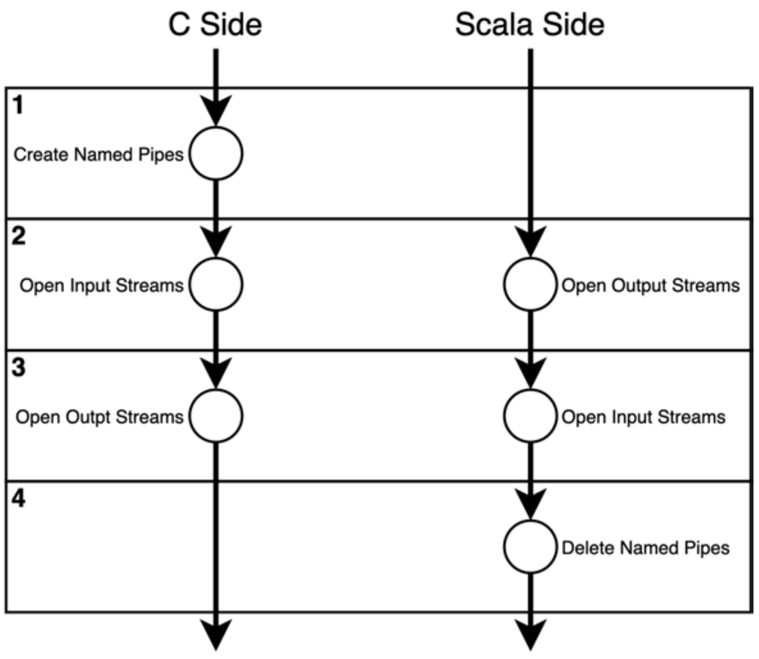


Figure 4: Timeline of named pipe phases.

the object's buffer, but does not guarantee that the underlying OS buffer will be flushed. In short, this means that calling `close()` is the only way to guarantee that a Scala based `io.outputstream` will flush its contents to the target in real time.

Due to this limitation, CFC introduces one last communication agreement between the SoC simulation and the co-simulation testbench, which is summarized in Table 2. In order to ensure that messages from Scala are written into the pipe in real time, CFC requires that all Scala output streams be opened just before a write and closed immediately after that write. To avoid any deadlock, the C side must open connections before a read is expected and close that connection after a message is received. This agreement means that neither side is left with a broken pipe. However, since both sides will stall waiting for the matching connection to be made, these Scala output writes become defacto synchronization barriers. Future work could be done to find another workaround for Scala output stream buffering, which would remove the unintended synchronization.

Transmit Phase	Sender	C Action	Scala Action
Pre	C	None	None
Pre	Scala	Open Input Stream	Open Output Stream
Post	C	None	None
Post	Scala	Close Input Stream	Close Output Stream

Table 2: Post communication action agreements.

3.3.2 Scala Pipe Connectors

CFC ties together protobuf IPC over named pipes and hardware-to-software converters into two object types. One reads a protobuf from an input stream, converts that message into a bundle literal, and pushes the bundle literal into a specified port on a DUT. This object is called a `PipeDriver`, as it takes input from a named pipe and drives it into a DUT. The other object monitors a specified port on a DUT, converts the seen bundle literals into protobuf messages, and writes those messages to an output stream. This type of object is called a `PipeMonitor`, as it monitors signals from a DUT and outputs them to a named pipe. Both of these objects types extend the empty `AbstractCosimPipe` trait for ease of staging in the co-simulation manager.

On an abstract level, the method by which each object works can be described as follows. A `PipeDriver` or `PipeMonitor` exists as an infinite loop inside a ChiselTest fork. The loop continually checks the input source for a new message or transaction. When new data is seen, the appropriate converter is applied to result in the output type, and then that output is either pushed into a driver or written to a pipe. The objects are placed in forked regions so that they can be run in parallel, which more realistically simulates a hardware only environment by allowing for transactions to be driven or monitored on multiple interfaces within the same clock cycle.

An example of the `DecoupledPipeDriver` is shown in Listing 11. This connector is an abstract base implementation that can be used to create a `PipeDriver` for any bundle with a `Decoupled` wrapper. More specific implementation details on and examples of pipe connectors are provided in Section 4.2.

It should be noted that there are alternative methods for transmitting bundle information between simulations other than the protobuf-based pipe connectors employed by CFC. One significant example is the bridge protocol employed by FireSim, which performs inter-simulation bundle transmissions using tokens. These tokens represent the per-cycle inputs and outputs of each simulation, and their use allows for a cycle-accurate distributed simulation to be maintained. However, CFC is unable to use this bridge protocol because it targets mixed-abstraction co-simulation with a purely functional core model that cannot generate cycle-accurate tokens.

3.3.3 Running Binaries From Scala

Co-simulation requires that the Scala-based testbench is able to invoke an external simulator on a specified test binary with any necessary arguments. Additionally, this process needs to capture the standard out, standard error, and exit code for correctness checking. To accomplish this, a `runCommand` utility function is defined within the `VerifCosimTestUtils` object. This function takes in a command as a sequence of `String`s, representing the space separated arguments to be executed in the command line, and returns the captured output values as a tuple. The full implementation of `runCommand` is shown in Listing 12.

3.3.4 The CosimTester Trait

The last component needed for the full cosim manager is a utility to determine the path in which a testbench is running. This allows the manager to create named pipes in

```scala
abstract class DecoupledPipeDriver[T <: Data, D](pipeName: String, clock: Clock)(implicit p: Parameters,
↪  cosimTestDetails: CosimTestDetails) extends AbstractCosimPipe {

  val driver: DecoupledDriverMaster[T]
  val inputStreamToProto: (java.io.InputStream) => D

  def pushIntoDriver(message: D): Unit

  val pipe = s"${cosimTestDetails.testPath.get}/cosim_run_dir/${pipeName}"

  fork {
    while (!Files.exists(Paths.get(pipe))) {
      Thread.sleep(250)
    }

    val in = new FileInputStream(pipe)

    clock.step()

    while (true) {
     if (in.available != 0) {
       val message = inputStreamToProto(in)
        if (message != null) {
          pushIntoDriver(message)
        }
      }
      clock.step()
    }
  }
}
```
CosimPipeConnectors.scala

Listing 11: Implementation of DecoupledPipeDriver.

```scala
  def runCommand(cmd: Seq[String]): (Int, String, String) = {
    val stdoutStream = new ByteArrayOutputStream
    val stderrStream = new ByteArrayOutputStream
    val stdoutWriter = new PrintWriter(stdoutStream)
    val stderrWriter = new PrintWriter(stderrStream)
    val exitValue = cmd.!(ProcessLogger(stdoutWriter.println, stderrWriter.println))
    stdoutWriter.close()
    stderrWriter.close()
    (exitValue, stdoutStream.toString, stderrStream.toString)
  }
```
CosimTestUtils.scala

Listing 12: Implementation of runCommand.

the appropriate sub-directories of test_run_dir. In order to accomplish this, a new trait CosimTester is defined that must be added to any co-simulation test class. For any given test case of the test class, the trait creates an object CosimTestDetails that contains the sbt root and test run directory. This object is implicitly passed to the CosimManager. The implementations of both CosimTester and CosimTestDetails are shown in Listing 13.

In future work, this trait could be expanded to capture thread information for identical tests running multiple times in parallel. Currently, running such tests will not work in co-simulation, as the pipes created are per test rather than per thread and test combination. However, if CosimTestDetails could capture thread information, then the pipe paths could be prefixed with thread information, solving this problem.

```scala
class CosimTestDetails {
  var sbtRoot: Option[String] = None
  var testName: Option[String] = None
  def testPath: Option[String] = if (sbtRoot.nonEmpty && testName.nonEmpty) Some(sbtRoot.get +
  ↪  "/test_run_dir/" + sanitizeFileName(testName.get)) else None
}

trait CosimTester extends TestSuiteMixin { this: TestSuite =>
  implicit val cosimTestDetails = new CosimTestDetails

  abstract override def withFixture(test: NoArgTest): Outcome = {
    cosimTestDetails.sbtRoot = Some(s"${File(".").toAbsolute}")
    cosimTestDetails.testName = Some(test.name)
    super.withFixture(test)
  }
}
```
CosimTestUtils.scala

Listing 13: Implementation of `CosimTester` and `CosimTestDetails`.

3.3.5 The Complete Co-Simulation Manager

Now, with all of the necessary building blocks defined, the complete `CosimManager` object can be built. The constructor takes in three arguments: 1) a simulator, 2) a sequence of no argument functions that create `AbstractPipeConnectors`, and 3) an implicit `CosimTestDetails`. These are considered to be constants to the manager object. Note that the simulator is searched for using the standard system `PATH` variable. This means that any installed simulator can be invoked from its name without needing a full path to the install location.

To run a simulation, the `CosimManager` has a `run` function, which takes in three arguments: 1) simulation arguments as a sequence of `Strings`, 2) the path to a test binary to execute as a `String`, and 3) a correctness check function that maps the exit code of running the simulation to a Boolean value. It should be noted that the manager is set up to search for targets relative to the `sbt` root. In the Chipyard context this is the top level `chipyard` directory.

When called, the run method invokes the supplied simulator on the given target. The supplied simulation arguments are also passed to the sim, as well as an automatically generated `--cosim-path` argument. This last argument is a special addition made to the standard Spike simulator that both enables co-simulation mode and informs the Spike where to create and connect to named pipes.

Next, the `CosimManager` starts each of the `AbstractPipeConnector` objects. By running the connectors in simultaneous forked regions, concurrent traffic to and from multiple ports can be maintained.

At this point, the co-simulation process is in full swing. Given proper synchronization barriers, it is assured that the software simulation thread cannot terminate until the hardware simulation is no longer working. Thus, the `CosimManager` steps the clock until the simulation thread terminates. A discussion of how synchronization barriers are constructed using Fence commands occurs in Section 4.

To conclude a co-simulation run, the manager asserts that the output of the correctness check lambda applied to the exit code from the simulation thread is true. This is the only assertion made during the co-simulation check. All other correctness checks should be performed inside the C-based simulator binary.

The generalized testing environment orchestrated by a `CosimManager` object during the duration of its `run` method is shown in Figure 5.

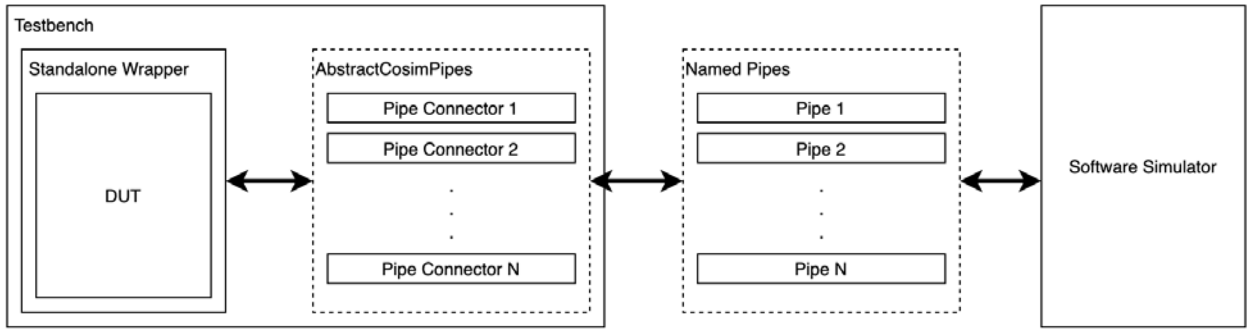


Figure 5: Test environment orchestrated by a `CosimManager`.

3.4 Testbench Setup

Creating a co-simulation testbench is done in a similar manner to any other ChiselTesters based testbench, with a few key differences to add in the components of CFC. First, `with CosimTester` must be added to the outer level test class. This enables the implicit passing of test directory information to internal CFC components.

Next, an implicit parameter set should be created to allow for the proper out-of-context elaboration of any diplomatic module. This is done using the `getVerifParameters` utility. Additionally, the test module should be wrapped in an appropriate standalone wrapper, configured to match device specifics. This wrapped module becomes the DUT for a given testbench. Because the DUT is a `LazyModule`, a given test case is formed around the module element of the DUT, which represents its elaborated hardware form.

Within a given test case, all the necessary pipe connectors should be created for the various ports of the DUT as defined by their constructors. Then, a new `CosimManager` should be created for a software simulator and a sequence containing all of the defined pipe connectors. Finally, to run a co-simulation workload in the test case, the manager's `run` function should be called on a sequence of desired simulation args, a target binary, and a lambda for exit code checking.

A testbench skeleton implementing the above mentioned components is shown in Listing 14. This skeleton is expanded upon in Section 4.2 to create a full testbench for Gemmini.

3.5 Optimization - ChiselTest Binary Caching

When built upon a standard ChiselTest backbone, a co-simulation test will elaborate a given DUT for every single test that is run. While this is highly advantageous in periods of

```scala
class CosimTestSkeleton extends AnyFlatSpec with CosimTester with ChiselScalatestTester {
  // Create implicit parameter set
  implicit val p: Parameters = VerifTestUtils.getVerifParameters()

  // Create device under test
  val dut = LazyModule(
    new VerifRoCCStandaloneWrapper(
      () => // Create a new Lazy RoCC based DUT
  ))

  val simPath = // Select simulator
  val simArgs = // Set simulator arguments

  it should "Run some test binary" in {
    val simTarget = // Path to test binary relative to Chipyard root

    // Test using Verilator, simulation caching, and waveform generation
    test(dut.module).withAnnotations(Seq(VerilatorBackendAnnotation, CachingAnnotation, WriteVcdAnnotation))
    ↪  { c =>
      // Create pipe connectors
      val commandPipe = () => new RoCCCommandPipeDriver("RoCCCommandPipe", c.clock, c.io.cmd)
      val tlPipe = () => new TLPipeConnector("TLAPipe", "TLDPipe", c.clock, c.tlOut(0))
      // Create other pipe connectors here

      // Create CosimManager object using simulator path and pipes connectors
      val manager = new CosimManager(simPath, Seq(commandPipe, tlPipe), c.clock)

      // Run the manager on the target test binary and assert that the exit code is 0
      manager.run(simArgs, simTarget, x => x == 0)
    }
  }
}
```
CosimTestSkeleton.scala

Listing 14: Co-simulation testbench skeleton.

rapid design change, a downside emerges when a large number of tests are run on a single stable design.

Consider the extreme case where elaborating a full SoC and accelerator simulator takes 20 minutes and each subsequent test run on the generated simulator takes one minute. At the same time, assume that running a co-simulation test takes a total of two minutes to elaborate the accelerator and run a test. If the quantity of tests run exceeds 15, it takes more time to run in co-simulation than compared to elaborating the monolithic SoC simulation. This is the exact opposite of what co-simulation hopes to achieve.

In order to address this shortcoming, an optimization was added to ChiselTest to allow the caching and re-use of Verilator [22] simulation binaries between test runs. In brief, the optimization specifies a new annotation `CachingAnnotation` that can be used to enable caching for a given test case. When this is the case, two hashes are generated, one for the high FIRRTL of the DUT and one for the annotations supplied to the test. If these hashes match the hashes of the last test run, then the existing Verilator binary in the test's run directory is used. Otherwise, the new hashes are saved into the test's run directory and a new Verilator binary is generated. The net result of this annotation is that tests run on identical DUTs do not waste time re-elaborating identical simulators.

4 Application to Gemmini

A key goal in the development of CFC was to improve the debugging environment for Gemmini, the open-source matrix-multiply accelerator, in order to allow for rapid feature development. This verification is built upon the CFC base framework, with specific modifications made to support features needed by Gemmini and its corresponding software tests.

Once the changes were implemented, a full testbench was built and used to run a series of tests. The results of these tests provide a basis to compare CFC to a full-SoC simulation.

4.1 Spike Modifications

Most of modifications made to support Gemmini were added to the Gemmini extension for Spike. One such change occurs in the `gemmini_t::custom3` function located in `gemmini.cc`. In the default Gemmini extension, this function processes incoming RoCC Commands and triggers the appropriate software modeling functions. However in the co-simulation enabled extension, `custom3` issues RoCC Commands to the co-simulation testbench as protobufs over named pipes rather than processing them locally. In effect, this disables the software Gemmini model in favor of the RTL Gemmini model.

A similar modification must be made to the `customX` instruction for any RoCC accelerator with an existing Spike extension that wishes to use Spike as the software simulation in a co-simulation testbench. Non-Spike software simulators should be modified in a similar matter, with the details dictated by specific simulation architecture.

Beyond adding the ability to issue RoCC commands to a co-simulation testbench, other Spike modifications were made to provide continual processing of TileLink memory traffic, to ensure hardware-software synchronization on `fence` instructions, and to allow for co-simulation mode to be enabled via a command-line argument. The specifics of these modifications are discussed in detail below.

4.1.1 Threaded TileLink Memory Model

The first attempt to create a software-based memory model for RTL Gemmini simulation to interact with revolved around using `fence` instructions as servicing points. When a `fence` was seen by the software sim, it would check for and service any incoming memory requests until the RTL simulation indicated that it was no longer busy.

While this approach would have successfully allowed for TL requests to be processed, it introduced a major issue. In a real Rocket-based SoC, TL requests are made directly to a memory device or peripheral. The servicing of these requests occurs on a per-device basis, independently of the core. However, in the process-during-`fence` model, the core is acting as if it is responsible for servicing memory requests. This eliminates the possibility of observing behavior wherein the core performs an operation at the same time as a memory request is serviced. Additionally, the bunched and all-at-once processing of memory requests creates an unrealistic memory environment for Gemmini in co-simulation.

To address these shortcomings, the threaded TL memory model was created. This model is built around a memory processing object, the `tl_memory_t`. The `tl_memory_t` object's `run` method starts an infinite loop that scans for memory requests on a named pipe, processes those requests, and responds on a different named pipe. In the Gemmini extension, a new `tl_memory_t` object is created. That object's run method is then run in a new thread through the use of the `pthread` library. As a benefit of using `pthread`, the memory thread can be cleanly terminated in spite of running in an infinite loop by using the `pthread_cancel` method in the Gemmini extension's destructor.

Since the processor thread and the memory thread operate on the same underlying `mmu_t` object, which represents the SoC's memory, special precautions had to be taken to make memory operations thread safe. To accomplish this, a mutex was defined in each `mmu_t` object. Then, the macro that creates load and stores operations was modified to include a `std::lock_guard` on the `mmu_t`'s mutex. When a load or store function is called, the `lock_guard` stalls until a lock can be acquired on the mutex. Once the lock is acquired, the function can execute, performing a load to store on the underlying memory object. When the function returns, the `lock_guard` falls out of scope, and its destructor unlocks the mutex. As a result, only one thread can execute a load or store to the same `mmu_t` at a given time.

All told, the threaded TL memory model allows for a memory processing thread to safely run independently of the core. Requests can be processed as soon as they are seen, and responses are sent back within one hardware cycle thanks to the relative speed of Spike when compared to an RTL simulation. The code for the memory model can be found in `tl_memory.h` and `tl_memory.cc`.

4.1.2 Fence Handling

As mentioned in previous sections, a key component of the co-simulation environment is the ability to synchronize hardware and software at critical points or on key instructions. The existing Spike simulation is a single threaded, single issue, purely functional model. Accordingly, attached accelerator models must complete any given instruction before the processor can move on to the next program instruction. However, RoCC instructions without the `xd` bit set do not block the processor until completion because they do not issue a RoCC response. This means that in the real world, multi-cycle accelerator actions can exist, while in the Spike architecture, they cannot.

This discrepancy creates issues when testing RoCC accelerators such as Gemmini, which have `xd` set to zero in their ISAs, and contain internal command queues.

In order to address this problem, a handler for the `fence` instruction was added to Spike. In normal Spike, `fence` instructions are handled like no-ops, consuming a cycle with no other changes to the processor state. In the CFC specific version of Spike, a `fence()` function is added to the `extension_t` class that all accelerator software models are based on. When a `fence` is seen, Spike checks if it has an enabled extension, and if it does, it calls on that extension's `fence()` function. In order to preserve the default no-op behavior of base Spike for accelerators models without a custom fence implementation, the `fence()` function has an empty default implementation in `extension.h`. In essence, this modifications allows an attached accelerator model to define its own set of rules for handling a fence.

In the Gemmini extension, when a `fence` instruction is seen, the called method sends a `GemminiFenceReq` over a named pipe and then spins in a loop, processing no further instructions. The request contains a single field, denoting the number of RoCC commands sent prior to the request being issued. On the other end of the named pipe is special type of pipe connector named the FencePipe. When a FencePipe sees an incoming GemminiFenceReq from the software sim, it spins until the busy signal in Gemmini's RoCCIO bundle goes low. When busy is low, it indicates that Gemmini has no further pending instructions to process, which would place it in line with the fence. Additionally, the FencePipe will spin until the number of commands pushed into Gemmini matches the number of commands specified in the fence request. This additional check prevents the case where busy is low because a fence request has been issued to Gemmini before it has time to process any commands. Once both checks are satisfied, the FencePipe sends a GemminiFenceResp back to the software sim. When Spike sees a GemminiFenceResp, it exits the infinite loop and continues processing the instruction flow. This ensures that both Spike and Gemmini are brought into sync on fence instructions.

Figure 6 depicts the program flow of Spike with the addition of fence handling as well as the custom3 modifications discussed in Section 4.1. Note that all non-fence and non-custom3 instructions are handled in the same manner as unmodified Spike.

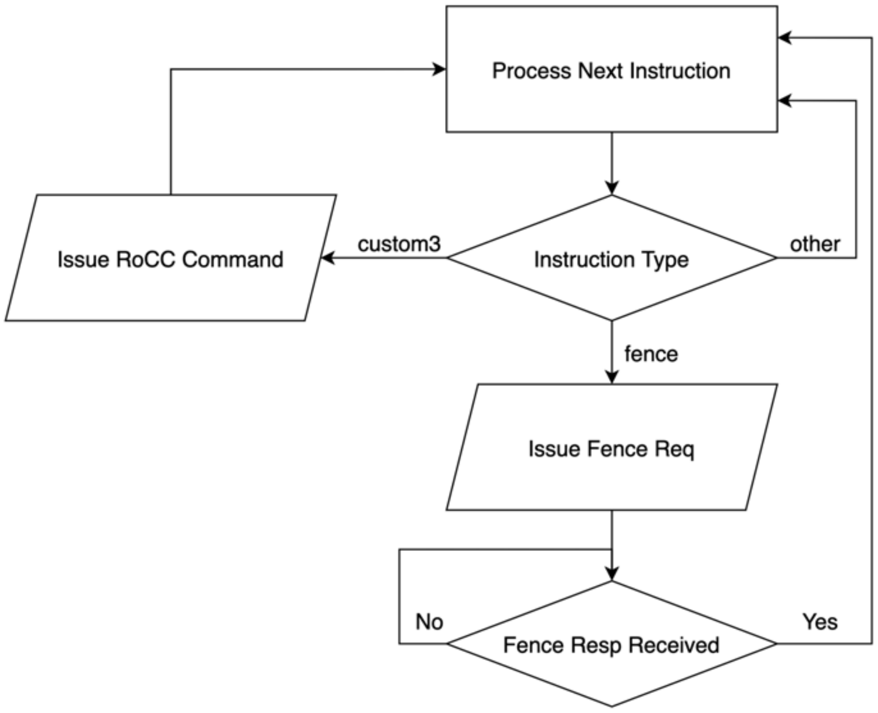


Figure 6: Modified Spike program flow.

4.1.3 Command Line Arguments

Another addition made to Spike in support of Gemmini co-simulation is the --cosim-path command line argument. This argument allows a user to specify the directory in which the simulation will create and connect to named pipes. Moreover, by making the default argument null, the flag can be used as a switch to enable or disable co-simulation features, such as sending out RoCC commands over named pipes instead of processing them in software.

The largest benefit of this addition is that a single Spike installation can be used to run both co-simulation testbenches and normal full software tests.

4.1.4 Gemmini Specific Pipe Connectors

In order to support Gemmini, specific pipe connectors needed to be created to support its RoCC Command and TL ports.

The `RoCCCommandPipeDriver` is a simple example of a `PipeDriver` used to drive new commands into the corresponding port on Gemmini. The connector is built upon the `DecoupledCosimPipeDriver` described in Section 3.3.2, and its implementation is shown in Listing 15.

```scala
class RoCCCommandPipeDriver(pipeName: String, clock: Clock, io: DecoupledIO[RoCCCommand])(implicit p:
↪  Parameters, cosimTestDetails: CosimTestDetails) extends DecoupledPipeDriver[RoCCCommand,
↪  RoCCProtos.RoCCCommand](pipeName, clock) {

  val driver = new DecoupledDriverMaster(clock, io)
  val inputStreamToProto = (in: java.io.InputStream) => RoCCProtos.RoCCCommand.parseDelimitedFrom(in)

  override def pushIntoDriver(message: RoCCProtos.RoCCCommand): Unit = {
    driver.push(new DecoupledTX(new RoCCCommand).tx(VerifProtoBufUtils.ProtoToBundle(message,
    ↪  VerifBundleUtils, new RoCCCommand)))
  }
}
```
CosimPipeConnectors.scala

Listing 15: Implementation of `RoCCCommandPipeDriver`.

In contrast to the `RoCCCommandPipeDriver`, the `TLPipe`, used to interface with the threaded TL memory model, is a more complex example of a pipe connector. The pipe is built around the `TLDriverSlave` from the `tilelink` package in the Chisel Verification Repository library. In high level terms, the driver monitors a TL-A port for an incoming request. When that request is seen, the driver passes that request to a processing function, which models a response to the observed TL-A message. Finally, the driver pushes the TL-D message returned by the processing function into a TL-D port. Due to this functionality, a majority of the `TLPipe` is contained within its processing function.

The `TLCosimMemoryInterface` processing function is responsible for sending memory requests to and reading memory responses from the threaded TL memory model. In order to properly handle burst messages, the function is defined on the `TLCosimMemoryBufferState` type. This type contains a sequence of observed TL-A messages. If an observed TL-A message does not form a complete transaction, a new state is formed by appending the observed message onto the state's existing sequence. Once the state's sequence of TL-A messages forms a complete transaction (either a single message or all the messages of a burst request), the `TLCosimMemoryInterface` sends each message over a named pipe to the threaded TL memory model. The processing function then accumulates the TL-D responses it sees into a sequence. Once that sequence forms a complete transaction, it is returned and the TL-D messages it contains are pushed into Gemmini.

If each TL-A from a burst request were independently sent by the `TLCosimMemoryInterface`, the processing function would wait for a TL-D response before sending any of the other TL-A messages from that burst request. At the same time the TL memory model has received a TL-A message but needs the rest of the burst request before it can send any TL-D response. In this case, both sides of the communication chain would be stuck waiting for messages that can never be received. Therefore waiting for a complete transaction to be present before sending any of the observed TL-A avoids deadlock.

The implementation of `TLPipe` is shown in Listing 16.

4.2 Gemmini Testbench Setup

The co-simulation testbench for Gemmini is largely created in the manner discussed in Section 3.4. For simplicity, a `dut` variable, referencing a Gemmini module wrapped in a `VerifRoCCStandaloneWrapper` is declared for the entire `GemminiCosimTest` class. However, because this variable references a `LazyModule`, the Gemmini module will be re-elaborated for each test case independently. In a similar vein, `simPath` and `simArgs` variables are defined for the entire class, as Spike will always be the desired sim, and Gemmini will always be the desired simulation extension.

In an individual test case, a variable for the target co-simulation software workload is defined. Then, pipe connectors are created for Gemmini's RoCC Command port and TL port. Additionally, a `FencePipe` connector is created for synchronization. These connectors are placed into a sequence and passed to the constructor for a `CosimManager` along with the global simulator variable, in this case Spike. Next, the testbench calls the `run` method of the newly created manager with globally defined simulation arguments and the locally defined simulation target. Successful C-based Gemmini tests return an exit code of 0, so the manager's exit code lambda checks for that value. As described in previous sections, the success of the testbench is based on if this lambda check asserts to true.

The complete environment orchestrated by the `CosimManager` during the execution of `run` within each test case is depicted in Figure 7. Note that this is a refinement upon the environment depicted in Figure 2.

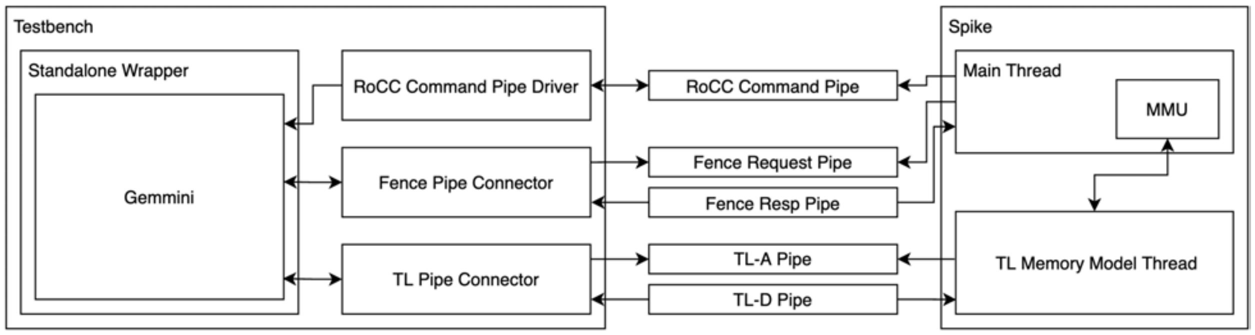


Figure 7: Test environment orchestrated by the `CosimManager` during a Gemmini test.

An example of a test case used to execute the `mvin_mvout-baremetal` test is shown in Listing 17. This can be thought of as a boilerplate example of how to run Gemmini tests in co-simulation.

```scala
case class TLCosimMemoryBufferState(txnBuffer: Seq[TLChannel])

class TLCosimMemoryInterface(tlaPipe: String, tldPipe: String, bundleParams: TLBundleParameters)
                            (implicit p: Parameters, cosimTestDetails: CosimTestDetails) extends
                            ↪  TLSlaveFunction[TLCosimMemoryBufferState] {
  while (!Files.exists(Paths.get(tlaPipe)) || !Files.exists(Paths.get(tldPipe))) {
    Thread.sleep(250)
  }

  // NOTE: Scala convention is to open inputs before outputs in matching pairs
  val tld_pipe = new FileInputStream(tldPipe)

  override def response(tx: TLChannel, state: TLCosimMemoryBufferState): (Seq[TLChannel],
  ↪  TLCosimMemoryBufferState) = {
    tx match {
      case txA: TLBundleA =>
        val tla_buffer = state.txnBuffer :+ txA
        if (TLUtils.isCompleteTLTxn(tla_buffer, 16)) {
          tla_buffer.foreach(tla => {
            val tla_proto = VerifProtoBufUtils.BundleToProto(tla, TLProtos.TLA.newBuilder())

            val tla_pipe = new FileOutputStream(tlaPipe)
            tla_proto.writeTo(tla_pipe)
            tla_pipe.close()
          })

          var tld_buffer = Seq[TLChannel]()
          do {
            var tld_proto = com.verif.TLProtos.TLD.parseDelimitedFrom(tld_pipe)
            while(tld_proto == null) {
              tld_proto = com.verif.TLProtos.TLD.parseDelimitedFrom(tld_pipe)
            }
            tld_buffer = tld_buffer :+ VerifProtoBufUtils.ProtoToBundle(tld_proto, VerifBundleUtils, new
            ↪  TLBundleD(bundleParams))
          } while (!TLUtils.isCompleteTLTxn(tld_buffer, 16))

          (tld_buffer, TLCosimMemoryBufferState(Seq()))
        } else {
          (Seq(), TLCosimMemoryBufferState(tla_buffer))
        }
      case _ => ???
    }
  }
}

class TLPipeConnector(tlaName: String, tldName: String, clock: Clock, io: TLBundle)
           (implicit p: Parameters, cosimTestDetails: CosimTestDetails) extends AbstractCosimPipe {

  val tlaPipe = s"${cosimTestDetails.testPath.get}/cosim_run_dir/${tlaName}"
  val tldPipe = s"${cosimTestDetails.testPath.get}/cosim_run_dir/${tldName}"

  val driver = new TLDriverSlave(clock, io, new TLCosimMemoryInterface(tlaPipe, tldPipe, io.params),
  ↪  TLCosimMemoryBufferState(Seq()))
}
```
CosimPipeConnectors.scala

Listing 16: Implementation of TLPipe.

In order to run a different test, the only line that needs to be changed is the one shown in Listing 18, which defines the simulation target. All other lines can remain the same.

```scala
class CosimTest extends AnyFlatSpec with CosimTester with ChiselScalatestTester {
  implicit val p: Parameters = VerifTestUtils.getVerifParameters()

  val dut = LazyModule(
    new VerifRoCCStandaloneWrapper(
      /*** SET YOUR CONFIGURATION FOR COSIM HERE ***/
      () => new Gemmini(OpcodeSet.custom3, GemminiConfigs.defaultConfig.copy(use_dedicated_tl_port = true,
    meshRows = 4, meshColumns = 4, rob_entries = 4)),
    beatBytes = 16,
    addSinks = 1
  ))

  it should "Run mvin_mvout-baremetal" in {
    val simTarget = "generators/gemmini/software/gemmini-rocc-tests/build/bareMetalC/mvin_mvout-baremetal"

    test(dut.module).withAnnotations(Seq(VerilatorBackendAnnotation, CachingAnnotation, WriteVcdAnnotation))
    ↪  { c =>
      val commandPipe = () => new RoCCCommandPipeDriver("RoCCCommandPipe", c.clock, c.io.cmd)
      val fencePipe = () => new FencePipeConnector("GemminiFenceReqPipe", "GemminiFenceRespPipe", c.clock,
      ↪  c.io)
      val tlPipe = () => new TLPipeConnector("TLAPipe", "TLDPipe", c.clock, c.tlOut(0))

      val manager = new CosimManager(simPath, Seq(commandPipe, fencePipe, tlPipe), c.clock)
      manager.run(simArgs, simTarget, x => x == 0)
    }
  }

}
```
CosimTest.scala

Listing 17: Boilerplate code for Gemmini co-simulation tests.

```scala
    val simTarget = "generators/gemmini/software/gemmini-rocc-tests/build/bareMetalC/matmul-baremetal"
```
GemminiCosimTest.scala

Listing 18: Definition of the simulation target.

4.3 Results

In order to measure the performance difference between a co-simulation testbench and a tradition full-SoC simulation, a series of tests were performed and timed using both testing methodologies. In both methodologies, the simulators were built with settings that produced waveform dumps, as these dumps would be critical to debugging applications. Moreover, each set of tests was performed on both 4x4 Gemmini and 16x16 Gemmini configurations, where the NxN configuration represents a default Gemmini modified to have N rows, N columns, and N ROB entries.

First, a test was performed to time the elaboration and building of a simulation binary (referred to simply as elaboration), without running any tests on the produced simulator. For the co-simulation testbench, this consisted of running an empty test on a Gemmini module. For the full-SoC simulation, this consisted of calling `make debug` on a properly parameterized `GemminiRocketConfig`.

After benchmarking elaboration times, the time to run a test binary using either simulation method was recorded. This test was performed for the `mvin_mvout-baremetal`, `matrix_add-baremetal`, and `matmul-baremetal` test binaries, which represent increasing level of computational complexity.

4.3.1 Collection Methodology

In order to present an accurate picture of run times, all tests were performed five times per configuration. This repetition was used to control for inter-test variation. The results of these repeated tests are displayed in a table within each section.

To ensure a controlled collection environment, all results were gathered using a VMware Fusion-based virtual machine running Ubuntu 64-bit 20.04.1 with 10 GB of RAM and 4 processor cores. Verilator was used as the simulator backend. The raw time values presented correspond to run times with this system and environment configuration. The times were measured using the Unix `time` command for SoC tests, and the `sbt` elapsed time measure for co-simulation tests. Recognizing that different machines may run faster or slower than the testing configuration, relative speedups are also presented where appropriate.

4.4 Pre-Optimization Results

The pre-optimization results are those gathered before the addition of Verilator binary caching to ChiselTest.

4.4.1 Elaboration Time

Table 3 shows the elaboration times recorded for both the 4x4 and 16x16 Gemmini configurations.

Configuration	Run	Full SoC	Cosim.
4x4	1	1197	67
	2	1144	73
	3	1105	70
	4	1062	68
	5	977	69
16x16	1	7156	353
	2	7112	337
	3	7072	357
	4	7085	380
	5	6835	352

Table 3: Pre-Optimization elaboration times (sec).

For the 4x4 configuration, the full-SoC had an average elaboration time of 1097 seconds, while co-simulation had an average elaboration time of 69.4 seconds. This represents an average elaboration time speedup of 15.8x when using co-simulation. For the 16x16 configuration the speedup was slightly larger at 19.8x, with an average of 7052 seconds for a

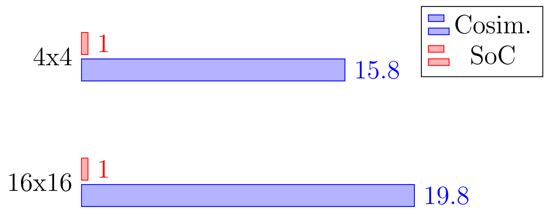


Figure 8: Elaboration speedup relative to full-SoC simulation.

full SoC-elaboration versus an average of 355.8 seconds for a co-simulation elaboration. The relative speedups are shown in Figure 8.

Of note is that the speedup seen does not scale directly with the relative change in Gemmini size. While the difference in size between a 4x4 and 16x16 Gemmini is approximately 16x, the change in elaboration speedup was only about 1.25x. This is likely due to the fact that core elaboration dominates the overall speedup figure. Any speedups seen in Gemmini elaboration between the two configurations only results in a small change in overall speedup when factoring in the large constant speedup from eliminating core elaboration.

4.4.2 Test Time

Table 4 shows the run times for the mvin_mvout, matrix_add, and matmul tests run on both Gemmini configurations using both simulation methods.

For the 4x4 configuration, co-simulation had an average mvin_mvout run time of 74.2 seconds, an average matrix_add run time of 76.2 seconds, and an average matmul run time of 106.4 seconds. In comparison, the full-SoC had average run times of 133.8, 149.2, and 539.6 seconds respectively.

In both simulation methods, the run times follow the same pattern, with $t_{matmul} > t_{matrix\_add} > t_{mvin\_mvout}$. However, the relative increases in time between different levels of test complexity are less in co-simulation than they are in full-SoC simulation. This is likely due to the dominance of elaboration time (69.4 seconds per test on average) in co-simulation. When the average co-simulation elaboration time is subtracted from the average co-simulation test run time, the relative increases become more similar.

For the 16x16 configuration, co-simulation had an average mvin_mvout run time of 430.6 seconds, an average matrix_add run time of 421.6 seconds, and an average matmul run time of 441.2 seconds. In comparison, the full-SoC had average run times of 269, 291.8, and 897.8 seconds respectively. The full dataset used to compute the averages for both this configuration and the 4x4 configuration are shown in Table 4.

Config.	Run	mvin_mvout		matrix_add		matmul	
		SoC	Cosim.	SoC	Cosim.	SoC	Cosim.
4x4	1	123	89	189	75	552	104
	2	147	77	158	84	541	117
	3	137	68	139	74	568	99
	4	120	69	131	73	472	104
	5	142	68	129	76	565	108
16x16	1	272	391	299	445	931	431
	2	267	433	289	485	896	511
	3	268	411	293	438	890	441
	4	271	446	288	357	882	407
	5	267	472	290	383	890	416

Table 4: Pre-Optimization test run times (sec).

There are a few interesting results that come out of the 16x16 Gemmini configuration. First, while the full-SoC run times still follow the ordering described above, the co-simulation run times do not. Specifically $t_{matrix\_add} \not> t_{mvin\_mvout}$. From this it seems that 16x16 co-simulation run time is so greatly dominated by elaboration time that variations in elaboration time obscure the actual differential in test run time. Additionally, while the run times for co-simulation were strictly less than the run times for full-SoC simulation in the 4x4 configuration, this does not hold for the 16x16 configuration. For the smaller mvin_mvout and matrix_add test cases, the overhead of re-elaboration in co-simulation (355.8 seconds on average) is significant enough to make the full-SoC simulation faster than the co-simulation.

These findings will be revisited Section: 4.5, when elaboration is cached for repeated co-simulation runs.

Figure 9 displays the absolute test run times of co-simulation and full-SoC simulation for both configurations. These run times are converted into speedups relative to the full-SoC simulation in Figure 10.

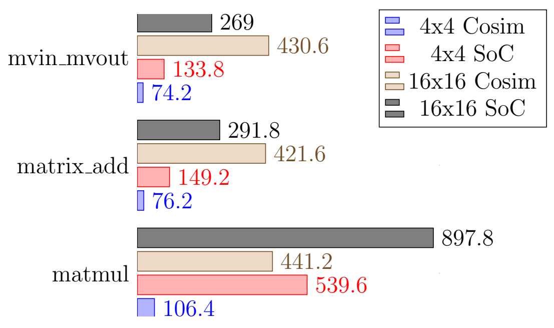


Figure 9: Avg. test run time (sec).

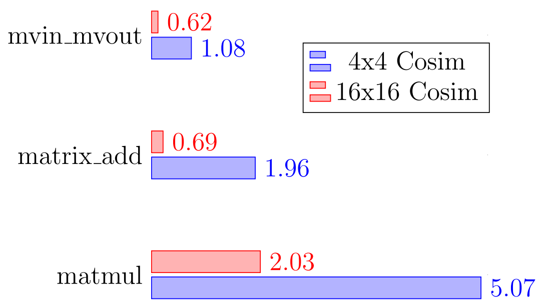


Figure 10: Avg. test speedup relative to full-SoC simulation.

While co-simulation elaborates Gemmini in each test run, the full-SoC simulation does not. If the average elaboration time for a SoC with configuration c is γ_c, the approximate total time consumed to run n tests on a full-SoC simulation can be approximated to $t_{SoC,c} = \gamma_c + \tau_{SoC,c} * n$, where $\tau_{SoC,c}$ is the average run time for the target test on a full-SoC simulation of configuration c. In contrast, the total time consumed to run n tests in co-simulation can be approximated to $t_{Cosim,c} = \tau_{Cosim,c} * n$ where $\tau_{Cosim,c}$ is the average run time for the target test on configuration c in co-simulation.

The ratio $\frac{t_{Cosim,c}}{t_{SoC,c}}$ provides insight into the speedup relationship between co-simulation and full-SoC simulation. The ratio allows the calculation of an asymptotic minimum possible speedup in all cases. Moreover, in the case where $t_{SoC,c} > t_{Cosim,c}$ finding the point where the ratio equals one determines the number of tests needed to make co-simulation a less time-efficient simulation method. Analysis of $\frac{t_{Cosim,c}}{t_{SoC,c}}$ for all tests is shown in Table 5. In brief, un-optimized co-simulation is a faster simulation method for all configurations and tests except for when 16x16 mvin_mvout is run over 52 times or 16x16 matrix_add is run over 54 times.

Config.	Test	$\frac{t_{Cosim,c}}{t_{SoC,c}}$	Asymptotic Speedup	Convergence
4x4	mvin_mvout	$1.80 + \frac{14.78}{n}$	1.80x	N/A
	matrix_add	$1.96 + \frac{14.40}{n}$	1.96x	N/A
	matmul	$5.07 + \frac{10.31}{n}$	5.07x	N/A
16x16	mvin_mvout	$0.67 + \frac{17.47}{n}$	0.67x	52.39
	matrix_add	$0.69 + \frac{16.73}{n}$	0.69x	54.33
	matmul	$2.03 + \frac{15.98}{n}$	2.03x	N/A

Table 5: $\frac{t_{Cosim,c}}{t_{SoC,c}}$ analysis for pre-optimized co-simulation.

4.5 Post-Optimization Results

The post-optimization results are those gathered after addition of Verilator binary caching to ChiselTest.

After the addition of Verilator binary caching, the co-simulation run times for both Gemmini configurations significantly decreased. For the 4x4 configuration, mvin_mvout had an average run time of 20.6 seconds, matrix_add had an average run time of 22.8 seconds, and matmul had an average run time of 56.6 seconds. The 16x16 configuration has average run times of 53.2 seconds, 58.6 seconds, and 92.4 seconds respectively. The full dataset used to compute these new averages are shown in Table 6.

Figure 11 displays these run times relative to the full-SoC, while Figure 12 converts the run times into relative speedups. The largest takeaway from these figures is that with optimized ChiselTest there is no longer a test for which co-simulation has a slower run time that

Config.	Run	mvin_mvout	matrix_add	matmul
4x4	1	21	23	43
	2	20	22	48
	3	21	23	65
	4	22	23	62
	5	19	23	65
16x16	1	50	67	81
	2	52	54	84
	3	55	57	105
	4	53	54	91
	5	56	61	101

Table 6: Test run times for optimized co-simulation (sec).

a full-SoC simulation. In fact, co-simulation has around or over 5x speedup for all test binaries across all configurations. Even more compelling is that for the most complex workload, matmul, co-simulation exhibits a speedup of over 9.5x across both Gemmini configurations.

An additional comparison can be made between the test run times for co-simulation with and without ChiselTest optimizations. These results are shown as relative speedups in Figure 13. This figure shows that even in the most elaboration-dominated testbench, 4x4 Gemmini matmul, optimized ChiselTest still contributes a significant 1.88x speedup in test run time.

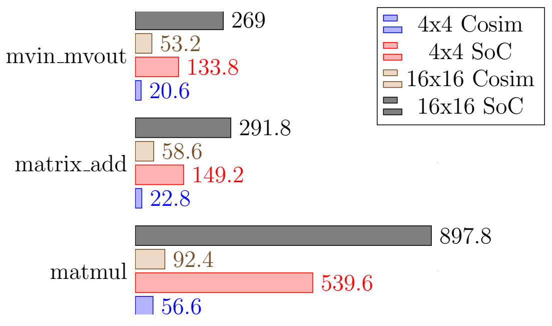


Figure 11: Avg. test run times for optimized co-simulation (sec).

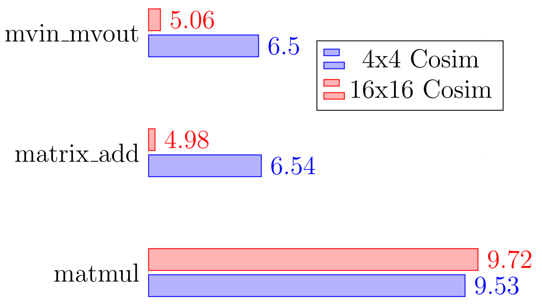


Figure 12: Avg. test speedup relative to full-SoC simulation.

Finally, the ratio $\frac{t_{Cosim,c}}{t_{SoC,c}}$ can be revisited for the post-optimization results. With the addition of caching, the formula used to approximate the total time to run n tests in co-simulation changes to $t_{Cosim,c} = \tau_{Cosim,c} + \lambda_{Cosim,c} * (n-1)$ where $\tau_{Cosim,c}$ is the average uncached run time for the target test on configuration c in co-simulation, and $\lambda_{Cosim,c}$ is the average cached run time. Since there is now no longer any test case for which which full-SoC simulation run time is less than co-simulation run time, the ratio now only provides information about an asymptotic minimum possible speedup. Analysis of $\frac{t_{Cosim,c}}{t_{SoC,c}}$ for post-

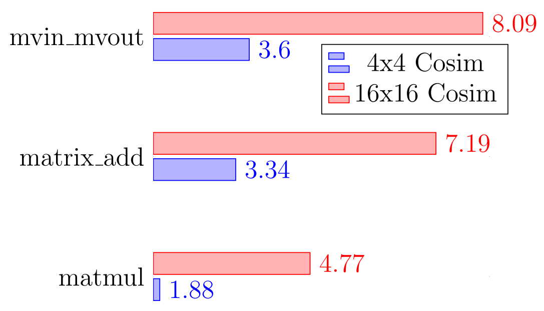


Figure 13: Avg. test speedup relative to pre-optimization co-simulation.

optimization results are shown in Table 7. Note that the asymptotic speedups shown match the relative speedups shown in Figure 12 due to the relative dominance of test run time versus elaboration time as n approaches infinity.

Config.	Test	$\frac{t_{Cosim,c}}{t_{SoC,c}}$	Asymptotic Speedup
4x4	mvin_mvout	$\frac{1097+133.8n}{74.2+20.6(n-1)}$	6.50x
	matrix_add	$\frac{1097+149.2n}{67.2+22.8(n-1)}$	6.54x
	matmul	$\frac{1097+539.6n}{106.4+56.6(n-1)}$	9.53x
16x16	mvin_mvout	$\frac{7052+133.8n}{430.6+53.2(n-1)}$	5.06x
	matrix_add	$\frac{7052+133.8n}{421.6+58.6(n-1)}$	4.98x
	matmul	$\frac{7052+133.8n}{441.2+92.4(n-1)}$	9.72x

Table 7: $\frac{t_{Cosim,c}}{t_{SoC,c}}$ analysis for optimized co-simulation.

5 Conclusion

5.1 Summary of Produced Tools and Results

This paper introduced CFC, an instruction-accurate co-simulation framework for Chisel that enables rapid accelerator design space exploration by pairing a software model of a Rocket core with an RTL simulation of a RoCC accelerator. Building this framework led to the development of wrappers to enable diplomatic devices to elaborate out-of-context. It also required the creation of utilities to enable signals on hardware wires to be translated to and from protocol buffers. Applying CFC to Gemmini led to custom modifications to Spike, enabling a single simulator to be used for both full functional simulation and co-simulation. Finally, developing CFC led to implementation of Verilator binary caching within

ChiselTest to optimize performance. This development has impact beyond CFC, speeding up ChiselTest-based testbenches for all tests that reuse a DUT.

Testing CFC on Gemmini showed significant improvements in both elaboration and simulation time. In elaboration, CFC was an average of 15.8x faster than elaborating a full-SoC simulation for a 4x4 Gemmini configuration and 19.8x faster for a 16x16 Gemmini configuration. With ChiselTest binary caching enabled, CFC was able to execute the matmul test binary an average of 9.72x faster than a full-SoC simulation for a 4x4 Gemmini and 9.53x faster for a 16x16 Gemmini configuration. Similar results were found for the mvin_mvout and matrix_add benchmarks as well, with 4x4 configuration speedups of 6.50x and 6.54x and 16x16 configuration speedups of 5.06x and 4.98x respectively.

These significant increases in simulation speed, as well as the ability to quickly test an accelerator against different processor environments motivates the continued use and development of CFC as a verification tool.

5.2 Potential For Future Use

While this thesis focused on using CFC to test Gemmini, there are a wide range of other potential use cases. The first use case is testing other accelerators. CFC can be expanded to include support for other existing RoCC accelerators such as Hwacha, which will dramatically increase the speed at which new changes to these devices can be tested at the SoC-integration level. Moreover, CFC can be used to aid in the development of new accelerators, allowing for faster bring-up of new IP.

Another use case for CFC is transaction capture and replay. In this application, a co-simulation run is performed on a DUT and the protobuf transactions for that run are recorded. Those transactions can be replayed into the DUT with different delays or relative orderings in order to simulate modified processor environments. For example, the captured RoCC commands can be pushed into the DUT at various rates of cycles per instruction to model processors with different instruction injections rates. Similarly, variable delays can be added to commands that occur after branch instructions to simulate the impact of faster or slower branch prediction resolution.

Finally, CFC can be used to quickly simulate the impact of different memory environments. One way that the memory environment can be changed is by simulating the impact of different bus widths or supported bus operations. For instance, changing the `beatBytes` parameter of the `VerifRoCCStandaloneWrapper` wrapping a RoCC device will simulate changing the TL bus width. Another way to change the memory environment is by modifying the number of wait cycles used by the `TLDriverSlave` inside the `TLPipeConnector`. In the Gemmini tests, the wait cycles were set to zero, so new TL messages were pushed into Gemmini as soon as they arrived. However, by increasing this value it is possible to test the impact of memory system latency on device functionality.

Beyond these detailed future use cases, there are a wide range of other use applications for CFC. Some examples include protocol compliance checking and debugging SoC level bugs. This list will likely grow with future developments and additional research.

5.3 Open Issues

While CFC is sufficiently developed for use with Gemmini, there are a few open issues that offer potential for future research and development. The largest such issue is avoiding deadlock. In particular, the fact that named pipes block until both a reader and writer are connected to them creates an immense challenge during the initial startup phase of co-simulation. Imagine the situation where a producer and consumer attempt to connect to two different named pipes, A and B. If the producer opens an output stream to A before B and the consumer opens an input stream to B before A, then a deadlock will occur. CFC avoids such deadlocks by establishing a specific order in which named pipes should be connected.

Future work could be done to develop a system that avoids deadlock without the need for such a tenuous system. One proposal for how to achieve this is to create a master list of named pipes with their writers and readers demarked in some fashion. The writers and readers can then programmatically negotiate an order in which to open connections such that deadlock is avoided. This is somewhat similar to how Diplomacy is used to negotiate bus parameters during elaboration.

Another open issue with CFC is that it cannot be used to estimate device performance. Because Spike is a functional simulator, CFC is an instruction-accurate framework wherein synchronization only occurs between Spike and the RTL simulation on `fence` instructions. Additionally, because Spike is significantly faster than the RTL simulation, all of the RoCC commands that occur before a fence are available to the RTL simulation immediately following startup or the completion of one fence prior.

While the instruction-accurate approach has the advantage of flexibility and ease of implementation, it removes the ability to characterize performance measurements such as cycle count. This issue would be remedied by converting CFC into a cycle-accurate framework, trading off speed for increased real world accuracy. Doing so would require a cycle accurate software model be used in the place of Spike. To enable this transition, the same co-simulation components added to Spike would need to be added to this new simulation. Additionally, work would need to be done to introduce a communication method for clock synchronization such that software and hardware clock steps occur at the same time. In the meantime, a tool such a FireSim is the best option for performance profiling.

5.4 Repository Links

- Main CFC components: https://github.com/TsaiAnson/verif/tree/master/cosim
- Modified Spike: https://github.com/ryan-lund/esp-isa-sim
- Chipyard with CFC: https://github.com/ryan-lund/chipyard/tree/cosim

Bibliography

[1] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Springer US, 2003. ISBN: 978-1-4020-7401-1.

[2] *Chisel Verification Repository*. URL: https://github.com/tsaiAnson/verif.

[3] Andreas Hoffmann, Tim Kogel, and Heinrich Meyr. "A Framework for Fast Hardware-Software Co-simulation". In: *In Design, Automation and Test in Europe DATE'01*. 2001, pp. 760–765. URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=FE129429ED386A525420146CF43DB8CD?doi=10.1.1.12.6229&rep=rep1&type=pdf.

[4] *HDL Verifier*. URL: https://www.mathworks.com/products/hdl-verifier.html.

[5] *Vitis HLS*. URL: https://www.xilinx.com/products/design-tools/vitis.html.

[6] *Xilinx ISim*. URL: https://www.xilinx.com/products/design-tools/isim.html.

[7] 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.

[8] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[9] Jerry Zhao et al. "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine". In: *Fourth Workshop on Computer Architecture Research with RISC-V*. May 2020.

[10] *Dromajo*. URL: https://github.com/chipsalliance/dromajo.

[11] Maria Muñoz-Quijada, Luis Sanz, and Hipolito Guzman-Miranda. "SW-VHDL Co-Verification Environment Using Open Source Tools". In: *Electronics* 9.12 (2020). ISSN: 2079-9292. DOI: 10.3390/electronics9122104. URL: https://www.mdpi.com/2079-9292/9/12/2104.

[12] Alon Amid et al. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40.4 (2020), pp. 10–21. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996616.

[13] *sbt*. URL: https://www.scala-sbt.org/.

[14] Martin Maas, Krste Asanović, and John Kubiatowicz. "A Hardware Accelerator for Tracing Garbage Collection". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 138–151. DOI: 10.1109/ISCA.2018.00022.

[15] Qijing Huang et al. "Centrifuge: Evaluating full-system HLS-generated heterogenous-accelerator SoCs using FPGA-Acceleration". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942048.

[16] J. Bachrach et al. “Chisel: Constructing hardware in a Scala embedded language”. In: *DAC Design Automation Conference 2012*. June 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[17] *ChiselTest*. URL: https://github.com/ucb-bar/chisel-testers2.

[18] Hasan Genc et al. “Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures”. In: *arXiv preprint arXiv:1911.09925* (2019).

[19] *Spike*. URL: https://github.com/riscv/riscv-isa-sim.

[20] Kenton Varda. *Protocol Buffers: Google’s Data Interchange Format*. Tech. rep. Google, June 2008. URL: http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html.

[21] Henry Cook, Wesley Terpstra, and Yunsup Lee. “Diplomatic Design Patterns : A TileLink Case Study”. In: *First Workshop on Computer Architecture Research with RISC-V*. CARRV’17. Boston, Massachusetts, Oct. 2017.

[22] *Verilator*. URL: https://www.veripool.org/verilator/.

Appendices

A Additional Feature Additions

The following appendix covers features that were created during the development of CFC but ultimately not used within the framework. While unused within CFC, these features are still useful for other projects.

A.1 Coverage Annotations for ChiselTest

One of the main reasons for testing a component under co-simulation is to expose the device to real world workloads. Doing so lets a designer see what coverage is achieved on realistic workloads. This coverage information is often automatically collected by software simulators when specific flags are passed during their compilation process. However, while the process of adding coverage is simple from the simulator side, ChiselTest lacked the ability to add these flags.

In order to add specific directives for simulation, ChiselTest uses annotations added to a test harness. For example, the `VerilatorBackendAnnotation` annotation directs ChiselTest to use Verilator as the backend simulator for the RTL emitted by the test harness.

To add support for coverage to ChiselTest, a set of new annotations was added. The `LineCoverageAnnotation`, `ToggleCoverageAnnotation`, `BranchCoverageAnnotation`, and `ConditionalCoverageAnnotation` annotations turn on coverage collection for the structural elements that correspond to their name. In cases where full structural coverage is desired, the `StructuralCoverageAnnotation` annotation turns on all of the previously mentioned annotations. Finally, the `UserCoverageAnnotation` annotation turns on coverage for user defined cover-points. It is important to note that these annotations have no effect if the selected backend simulator does not support that coverage type. For example, since Treadle does not support any type of coverage collection, passing any of these new annotations along with the `TreadleBackendAnnotation` would do nothing. Table 8 summarizes the coverage types supported for each ChiselTest backend.

	Treadle	Verilator	VCS
Line		x	x
Toggle		x	x
Branch			x
Conditional			x
Structural		x	x
User		x	x

Table 8: Backends and supported coverage types.

When any of these new annotations are passed to a test harness, the executive for the test's selected backend simulator interprets them and creates the proper string of coverage flags for the simulator. Those flags are then appended onto the global flags list that is used during simulator compilation. Listing 19 illustrates how the flag map is created and appended to the global flags for VCS.

```scala
    val coverageFlags = (compiledAnnotations collect {
      case LineCoverageAnnotation => List("line")
      case ToggleCoverageAnnotation => List("tgl")
      case BranchCoverageAnnotation => List("branch")
      case ConditionalCoverageAnnotation => List("cond")
      case UserCoverageAnnotation => List("assert")
      case StructuralCoverageAnnotation => List("line", "tgl", "branch", "cond")
    }).flatten.distinct match {
      case Nil => Seq()
      case flags => Seq("-cm " + flags.mkString("+"))
    }

    val vcsFlags = moreVcsFlags ++ coverageFlags
```
VcsExecutive.scala

Listing 19: Creation of coverage flag map for VCS.