Karlsruhe Institute of Technology

Deterministic PU (detPU): FPGA-based fast, flexible coordination for datacenters

Master's Thesis
submitted by

Philipp Ludwig Waldemar Berdesinski

to the KIT Department of Informatics†

in cooperation with the Software Systems research group

at Università della Svizzera italiana (USI)⋆

Reviewer:	Prof. Dr. Patrick Thomas Eugster⋆
Second Reviewer:	Prof. Dr.-Ing. Frank Bellosa†
Advisor:	Davide Rovelli⋆

1. July 2024 – 2. January 2025

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, January 2, 2025

Abstract

Distributed systems primitives, such as consensus, play an important role in current software deployments. Especially in datacenters, where applications are split between different servers, mechanisms for coordination are required. The libraries and applications which provide such functionality currently execute alongside other applications and the operating system on the host. As a consequence of task switching and system calls, they are subject to unpredictable spikes in latency, which adversely affects performance. Recent integrations of FPGAs into PCIe networking cards present an opportunity in this respect. Due to increased proximity to the network transceivers and operational model of FPGAs, they experience lower and more stable latencies. To leverage this advantage, distributed systems primitives must be located on the FPGA itself.

This thesis analyses the communication latency and bandwidth characteristics provided by these accelerators in a datacenter setting. It then develops and implements an abstraction for synchronous algorithms to be implemented against. To show the viability of the framework and determine its performance characteristics, this thesis further implements a consensus primitive based on existing algorithms to leverage the developed abstraction. In order to evaluate the performance of the algorithm, as well as the framework components, a number of micro-benchmarks as well as full-system experiments are run. Results show that the framework is able to saturate 100G links while introducing minimal overhead, both in latency as well as required FPGA resources. This enables a new class of synchronous algorithms to be implemented with both reduced effort and a lower barrier to entry.

German version

Primitive aus dem Feld der Verteilten Systeme wie Konsensus spielen eine wichtige Rolle im heutigem Einsatz von Applikationen. Koordinationsmechanismen sind besonders in Datenzentren vonnöten, da Anwendungen dort oft über mehrere Server verteilt ausgeführt werden. Die Bibliotheken und bestehende Anwendungen, welche solche Dienste bereitstellen, teilen sich Prozessorzeit mit anderen

Programmen und dem Betriebssystem. Damit sind sie unvorhersehbaren Latenzspitzen ausgesetzt, was sich meist in schlechterer Leistung niederschlägt. Um diesen Effekten entgegenzutreten ist eine Implementierung dieser auf FPGAs naheliegend FPGAs sind eine Art von Recheneinheit welche durch ihr besonderes Rechenmodell und ihrer Nähe zu Netzwerkadaptern extrem schnelle und stabile Kommunikation erreichen. Dies wird durch die Integration von FPGAs in netzwerkfähigen PCIe-Erweiterungskarten ermöglicht.

Diese Masterarbeit analysiert die Latenz und Bandbreite dieser Beschleuniger in einer Datenzentren ähnlichen Umgebung. In dieser Arbeit wird zusätzlich eine Abstraktion zur vereinfachten Implementierung von verteilten, synchronen Algorithmen geschaffen. Darauf basierend wird ein existierender Konsensalgorithmus implementiert, und mit Bezug auf bereits bestehende Werke evaluiert und verglichen. Um dies Umzusetzen werden mehrere, kleinere, Versuche durchgeführt, sowie das Gesamtsystem gemessen. Die Ergebnisse zeigen die Fähigkeit der entwickelten Abstraktionen, die 100 Gigabit/s Netzwerkverbindungen auszulasten. Dies wird mit sehr geringen zusätzlichen Latenzen und extrem geringer Implementierungskosten auf dem FPGA umgesetzt. Hiermit eröffnet sich die Möglichkeit, bisher kaum genutzte, synchrone, verteile Algorithmen zu implementieren. Zusätzlich wird hierbei der Implementierungsaufwand erheblich reduziert.

Contents

Chapter 1

Introduction

Datacenters (DCs), often via cloud providers, are the primary deployment option for the majority of user-facing application, such as streaming providers [27], social networks [11], and more recently, large language models [26]. These installations are characterized by a high number of individual machines, densely connected to each other and organized into racks. In order to fully leverage the performance provided by such installations, applications must utilize multiple machines at the same time. A single server is constraint in its networking, computational performance and available storage, due to cost and hardware capabilities. This is best exemplified by databases, which need to scale with the number of users as well as the complexity of the interaction, and thus quickly reach the limits of individual servers. Another benefit of DCs and their high number of individual machines is the possibility for replication. This technique enables a level of fault tolerance [16, 28], resulting in reduced down times for services, a critical aspect for businesses with service level agreements. These goals can often be achieved in tandem by splitting an application into multiple processes.

To form a cohesive system from these individual processes, they must interact with each other. The field of distributed systems (DSs) strives to define, characterize and develop primitives and algorithms to that extend [9]. Providing coordination between processes is a common task, solved by many different abstractions, e.g. consensus. The protocols which reach consensus decide on common value, formed or chosen from a set of propositions. Most importantly, they enforce that all nodes decide the same value, eliminating discrepancies in the overall system. DCs offer an interesting environment for consensus implementations in this context. Servers have extremely low latency between one another, depending on the configuration, they might even sit in the same rack. Inside a rack, servers are commonly connected to a single switch, which is located in the highest slot, the top-of-rack-switch (TOR-switch) [19]. This physical and topological proximity to each other results in low transmission latency and switching delays respectively.

Yet, the operating system (OS) running on the servers creates additional latencies for applications [10], which are often highly unpredictable.

This jitter can be reduced when leveraging FPGAs. Field programmable gate arrays (FPGAs) are a type of compute element which can be programmed to mimic digital circuits. These circuits exhibit a highly predictable behavior and can be accurately simulated. More recently, FPGAs have been integrated into PCI Express (PCIe) expansion cards and include network ports [3]. Such an arrangement results in an accelerator which can operate in tandem with the host central processing unit (CPU), commonly referred to as a smart network interface card (SmartNIC). Since FPGAs do not have an OS layer between their network connection and their computational elements, they introduce little and highly stable processing delays to their network data path. Many distributed systems primitives can benefit from reduced latencies provided by DCs and FPGAs. Synchronous algorithms in particular, since outliers traditionally force the use of timeouts and round periods which far exceed the average case. Especially in the case of logical dependencies between consensus decisions, a reduction in decision time can speed up the overall system.

This thesis designs and implements a framework for synchronous algorithms against FPGA-equipped network accelerator cards, specifically the AMD Alveo™ U50. To structure the exploration of the topic, this thesis tackles the following research questions:

> **RQ1** What characterizes the latency of interactions and transmissions from and to FPGA based SmartNICs, especially w.r.t. stability?
>
> **RQ2** How can synchronous distributed systems primitives be more easily implemented on FPGA based SmartNICs?
>
> **RQ3** Can our synchronous consensus implementation outperform state of the art when leveraging FPGAs?

Results show that the FPGA experiences very stable network latency, even at higher utilization rates. This enables a number of abstractions to be designed, simplifying the implementation effort for synchronous DS algorithms. The synchronous consensus primitive implemented using these abstractions holds the potential to outperform state-of-the-art solutions.

This thesis is structured into 5 further chapters. Immediately following this chapter is the Background 2. It introduces a number of concepts and technologies employed in this thesis. Chapter 3 expands on how the algorithms can be embedded into the execution environment of FPGAs. Implementation 4 explores the details of how the systems presented in 3 are implemented in the accelerator.

After that, chapter 5 evaluates the system and the implementation and compares them to existing solutions. Finally, chapter 6 ends the thesis, covering a number of possible expansions and improvements for further study.

Chapter 2

Background

This chapter serves to introduce a number of topics which are required to understand the contributions of this thesis. It will cover distributed systems, FPGAs, as well as their related technologies and finally, related work.

2.1 Distributed systems

Distributed systems are a field of computer science which, as the name implies, tackles systems consisting of more than a singular process. To this end, models are needed to represent the behavior of actual hardware and software. One such model is the link model, which represents the assumptions placed on a link between processes. A perfect link is expected to never loose, duplicate or fabricate a packet, while a fair loss link can drop and/or fabricate a finite number of packets [9, p. 34-37]. How the behavior of a model is ultimately achieved, either through implementation in hardware or by building on top of another model, is not defined by the model. Another important model is the failure model, describing how a process behaves, especially when it fails. While this might not be intuitive, how a process fails can be important since other processes might not be aware that a failure occurred. A process based on the fail-stop failure model, where a failure results in stoppage of all actions and computations, would not expect a process to return malicious data. There are a number of models governing failure, link, and timing behavior.

On top of the assumptions and guarantees of a model, distributed abstractions can be built. Given a fair-loss-link, a perfect link can be created by layering abstractions, each working to provide stronger guarantees [9, p. 36, 38]. This is also the case for more complex abstractions, e.g. broadcast, which deals with dissemination of information.

Many of the algorithms which achieve distributed abstractions are built on

the partially synchronous model. This model assumes that timing assumptions are achieved eventually, but may not be achieved for arbitrarily long, but finite amounts of time [9, p. 47]. The algorithms in this thesis assume a different timing model, so called synchronous systems, motivated especially by the nature of the underlying hardware as covered in section 2.2. This thesis will cover the consensus abstraction, which takes in propositions and decides on a single value, that being the same for all correct processes. An implication of "regular" consensus, which may not be readily apparent, is that a process which fails is allowed to decide on a different, wrong, value, and return it to other abstractions. When enforcing that faulty processes are not allowed to return faulty values, the abstraction is called Uniform Consensus [9, p. 204-212]. Consensus and its other variants, of which there are many, are one of, if not the most important abstraction in distributed systems. The coordination which is enabled by agreement between processes is the basis for other abstractions, such as group membership. Thinking about the correctness of algorithms in the field of DS can be quite difficult. This is partially a result of the sheer number of possible states a system may experience. Formal proofs are the most rigorous answer to these questions but themselves are hard to construct and validate. For this reason, this thesis uses a variation [31] on an already proven algorithm [29]. Section 3.1 explores the workings and implications when using this particular uniform consensus protocol.

2.2 FPGAs

An FPGA is a type of integrated circuit (IC) which can be altered after production to mimic arbitrary[1] digital circuits. Its name, field programmable gate array (FPGA), hints at its inner workings. To better understand how FPGAs operate, it is best to start with one of its smallest components, the lookup table (LUT). At its core, LUTs are memory [17, p. 270]. Interpreting the input values of a boolean function as an address highlights the logical equivalence of a function evaluation and a memory access. In essence, it stores the truth table for that function. The obvious drawback is the increased size of the implementation. Implementing the boolean function $\overline{(a \wedge b \wedge c \wedge d)}$ requires a single 4-NAND gate, but 2^4 memory cells when realized as a LUT.

The power of LUTs lies in their ability to be reconfigured, sometimes even at runtime. Any N-input boolean function can be loaded into an N-input LUT. Packaging one or multiple LUTs with registers, as well as some additional logic such as multiplexers or full adders, results in a configurable logic block (CLB). Sequential or combinatorial functions can then be constructed by connecting and

[1] With some limitations.

```
wire clk, a, b;
reg output;
always @(posedge clk) begin
    output <= a & b;
end
```

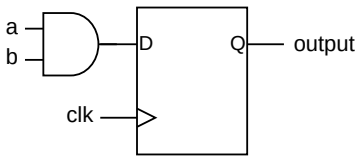

Figure 2.1: D-flipflop with operation in Verilog code (left) and in gate form (right)

configuring a number of CLBs accordingly. An FPGA usually consists of a array of these CLBs. Connections between these CLBs is done via routing channels whose connections can also be reconfigured at certain interconnect points [17, p. 274].

Implementing circuits through the use of CLBs is less efficient than implementing them directly in silicon. This results in increased space and power usage, as well as decreased clock speeds due to propagation delays. To lessen this impact, companies creating FPGAs include additional resources on or next to their silicon. AMD includes memory, memory controllers, I/O transceivers, and even entire GPU blocks in some of their products [3]. The Alveo product family by AMD goes a step further and integrates FPGAs with network transceivers on a PCIe expansion card. Overall, there are a number of companies creating FPGAs and derived products, ranging from very small devices targeting embedded systems to high performance devices with impressive compute capabilities.

2.3 HDL and RTL

Hardware description languages (HDLs) are employed to design digital circuits. These languages usually operate on the level of wires, registers and gates, a level of abstraction called register transfer level (RTL). While these languages do allow for circuits to be defined by connecting wires to gates explicitly, this is not the preferred method. Hardware description languages (HDLs) can be transformed, *synthesized*, to gate level by defining the behavior on a slightly higher level. SystemVerilog and Verilog are the HDLs used in this thesis.

Figure 2.1 shows a snippet of Verilog HDL code and an associated gate representation to illustrate basic synthesis. In the figure, the Verilog block uses 3 wires **clk**, **a**, **b**, and the register **output**. The always block is executed every time a positive edge is detected on the wire **clk**. Only when this condition is met, do the contents of the block take effect. Once a block executes, all operations take

place in parallel, which is the biggest difference from sequential languages. The definition of **output** as a register and the assignment inside an always block with a trigger condition of a rising clock implies a D-flip-flop. A D-flip-flop will save the value at its input **D** internally and output that value on its output **Q** whenever a rising edge is applied to its clock input. Because the value of the assignment is an AND operation on **a** and **b**, the `&` gets synthesized into an actual AND gate. When targeting FPGAs, the AND gate would most likely be realized in a LUT. The AND gate would be called the combinatorial path, since its output, **D**, is only dependent on its inputs and no previous state. More complex functions can also be defined. An addition operand in the HDL code would synthesize into an adder. These functions usually operate on bit-vectors, which represent an ordered group of individual wires or registers.

Still, operating on basic or built-in functions would be overwhelming when designing larger circuits. In order to create a hierarchy and enable reuse, these circuits can be organized into modules, which define a set of connections, usually either an input or an output. These are often referred to as intellectual property (IP) modules. In the context of FPGAs, modules which contain registers usually need at least a clock input. Adding a reset signal is also a good design choice since the data stored in registers is usually unknown at startup.

To test the circuit before deploying it to an FPGA or manufacturing the circuit, Verilog and SystemVerilog contain extra functionality to simulate the circuit. Testing on the FPGA would be difficult as FPGAs usually do not offer much debugging support without introducing overhead. Coming from software development, the lack of observability for a running application requires a shift in attitude towards tests. Simulations are the only reliable and efficient tool to test HDL designs, as well as identify and correct mistakes. This is usually done by individually, or through the use of verification IP (VIP) collectively, defining the input signals with special HDL code. The output can then be observed and checked against expected values.

Using these tools, more complex circuits can be created. From simple logic gates to full adders, from full adders to Arithmetic Logic Units, from Arithmetic Logic Units to CPUs. The book “Digital Design and Computer Architecture” by Harris & Harris [17] is a good resource for further study on the subject.

2.4 FPGA interfaces

To connect different IP modules together, a common interface or rather, a set of common interfaces is needed. There are already a number of interface standards in use like Avalon by Intel/Altera [21] or the Wishbone bus. However, since this thesis is using AMD Alveo U50 FPGAs, as well as the AMD Vivado IDE, the

Advanced eXtensible Interface (AXI) by ARM is used.

AXI is part of the Advanced Microcontroller Bus Architecture (AMBA) and defines three interfaces in its fourth iteration: AXI-Stream, AXI4, and AXI4-Lite. It is extensively used in the Xilinx Runtime Environment (*XRT*), the execution environment natively supported by the Xilinx FPGA accelerators. The following sections will illustrate these interfaces and their general use cases.

2.4.1 AXI-Stream

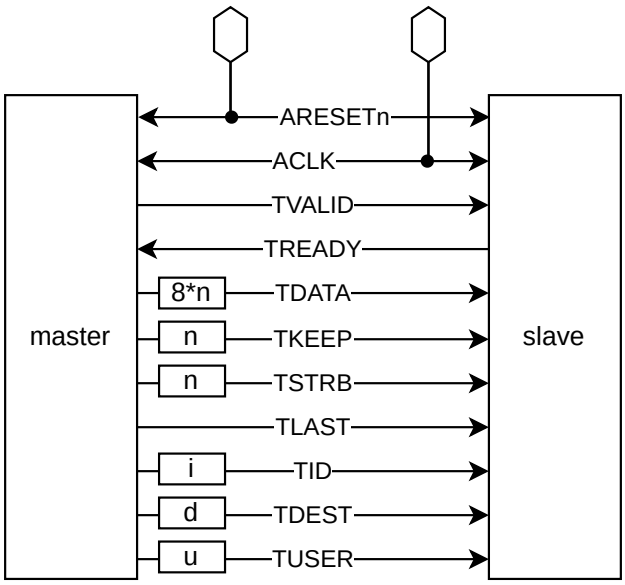

Figure 2.2: Signals in use by AXI-Stream with width parameters n, i, d, and u

AXI-Stream is an interface for transferring data between a data emitting source, the **master**, and a data consuming drain, the **slave** [8]. While the interface definition contains a total of 11 signals, shown in figure 2.2, this can be reduced to 6 types of signals, shown in Table 2.1. To better illustrate the function of these signals, we will focus on four signals, **ACLK**, **TDATA**, **TVALID**, and **TREADY**.

ACLK is the global clock and all signals should be sampled on its rising edge. The AXI-Stream protocol does not directly support the use of different clocks between the master and slave but there exist clock domain crossing modules for AXI-Streams. The master drives the **TDATA** signal to its proper value and indicates this by also driving the **TVALID** signal HIGH. This signals to the slave that there is data to be consumed. The slave indicates that it is able to receive data by

Type	Associated signals	Direction
Clock	**ACLK**	Global
Reset	**ARESETn**	Global
Data	**TDATA**	master to slave
Data valid	**TVALID**	master to slave
Slave ready	**TREADY**	slave to master
Metadata	**TKEEP**, **TSTRB**, **TLAST**, **TID**, **TDEST**, **TUSER**,	master to slave

Table 2.1: Types of signals in use by AXI-Stream

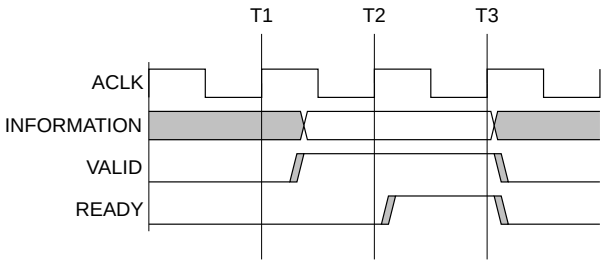

Figure 2.3: Excerpt from [8, p.39], showing a transmission where the master waits for the slave to acknowledge the transmission

driving the **TREADY** signal HIGH. A transfer occurs when, and only when both **TVALID** and **TREADY** are HIGH.

During that cycle, the slave must consume the value currently applied to **TDATA** since the master can, and probably will, change the value of **TDATA**, **TVALID**, or both. It is important to note that the slave is allowed to wait for the master to drive **TVALID** HIGH, as shown in figure 2.3, but the reverse case is not allowed. Not only can a master never wait for a slave to be ready, once it has driven **TVALID** HIGH, both **TVALID** and **TDATA** are not allowed to change until a transfer occurs. This rule does not imply that a slave cannot wait for a master to start a transaction while **TREADY** is asserted. Such a case can be seen in figure 2.4. Depending on the design of a circuit, it is also possible that a transfer can occur during a single cycle. Figure 2.5 depicts this case.

When talking about AXI-Streams, there is an associated terminology. If **TVALID** is driven HIGH but the slave has driven **TREADY** LOW, the stream is *stalled*. Such an event can be seen in figure 2.3. In this context, the slave has applied *backpressure* to the stream since the master cannot advance to the next data transfer. Conversely, if **TVALID** is LOW, the stream is considered *idle*. Figure 2.4 shows a handshake where the stream is initially idle.

There is a special case to this since the **TREADY** signal is optional, meaning

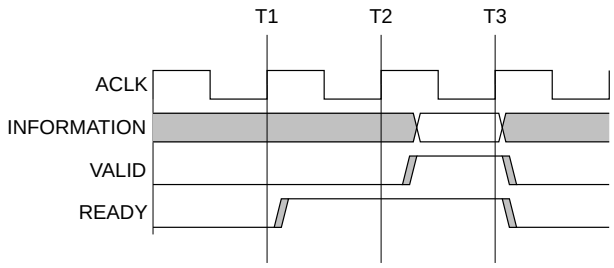

Figure A3-3 READY before VALID handshake

Figure 2.4: Excerpt from [8, p.39], showing a transmission where the slave waits for the master to initiate a transmission

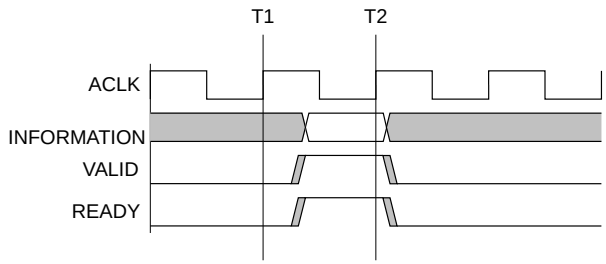

Figure A3-4 VALID with READY handshake

Figure 2.5: Excerpt from [8, p.40]. Sometimes, a master and slave drive both **TVALID** and **TREADY** high in the same cycle. In this case, the transfer completes in a single cycle.

it can be omitted. When the **TREADY** signal is omitted, the master assumes that the slave is always ready to consume the data and a transfer takes place on every cycle that **TVALID** is HIGH. The remaining signals are all generated by the master and read by the slave, and have the following functions:

ARESETn is a global reset signal used to reset all or some components to a known configuration, usually their intended initial state. It is active-LOW, meaning that a logical 0 is used to indicate that a component should reset.

TKEEP is a bit array containing "byte qualifiers", meaning that the interpretation of a single bit should be applied to a whole byte in **TDATA**. A logical 1 implies that the associated data byte does indeed contain valid data and cannot be discarded from the stream. This can happen either when the stream is sparse, e.g. only some of the data bytes contain valid data per transfer, or when the stream is unaligned, meaning that either the start, end, or both of the data is not aligned with the width of **TDATA**. The latter case is often unavoidable when sending data whose length is not a multiple of the stream data width, e.g. sending 48 bits of information over a 32 bit wide stream necessitates at least 2 transfers. Suppose the first 32 bits are transmitted in the first transfer, **TKEEP** would contain the value 1111_2. If the last 16 bits are transmitted in the 2 lower data bytes, **TKEEP** would contain the value 0011_2. The specification does not require transfers to be packed but a more sparse stream will, of course, decrease throughput. **TKEEP** is an optional signal, meaning that it does not need to be implemented by all instances of AXI-Stream. If **TKEEP** is not defined, it is assumed that all bits are a logical 1, and all **TDATA** bytes contain data.

TSTRB is a bit array containing byte qualifiers equivalent to **TKEEP**. This signal identifies so-called "position bytes". This marks a data byte to contain position data on following bytes rather than data itself. The specification does not mention specific use cases but the transmission of a sparse matrix can be sped up this way. Following a non-zero value, transmit the number of zero-values until the next non-zero value and mark the bytes containing this number as a position byte. Caution must be taken with such a use of **TSTRB** since the specification also explicitly states: "Since the data associated with a position byte is not valid, an interconnect need not transmit the **TDATA** associated with a byte for which **TSTRB** is deasserted LOW." [8, p. 21]. **TSTRB** is also an optional signal and its default value for all bits is logical 1.

TID is a signal of variable bit width **i** which differentiates logical streams from each other. This can be used to transmit data to different parts of a component or to different components entirely when using an interconnect. In the same vein,

TDEST is a signal with variable bit width **d** which is used to carry routing information on the specific stream. The specification contains a number of requirements when interleaving, packing, and up- or down-sizing streams with **TID** and **TDEST** signals but these are not required in the context of this thesis. Both **TID** and **TDEST** are optional signals and their absence implies that all data transferred on a stream interface belongs to the same logical stream.

TLAST allows for the discretization of the stream into packets of variable length by marking the final transfer of a packet with a logical 1. This can be used in conjunction with **TID** and **TDEST** to implement more complex networks but is primarily used in this thesis to indicate packet boundaries in a stream w.r.t. their transmission over the network. The absence of this optional signal implies that there are no packets. Different logic on this signal, e.g. driving **TLAST** to a logical 1 every x cycles or every x transfers can influence arbitration decisions when using interconnects.

TUSER is a signal which can be used to transfer out-of-band information. It has a variable bit width **u** and can either hold general or byte-qualifier information. Since the specification does not enforce a particular handling of this signal, care must be taken when using interconnect components since these might interpret the user signal as byte-qualifiers and split them up when up- or down-sizing streams.

When talking about a named AXI-Stream, stream **NAME_AXIS** will have the signals **NAME_AXIS_TVALID**, **NAME_AXIS_TREADY**, etc.. For brevity, especially in figures, the **AXIS_** part will be omitted, resulting in **NAME_TVALID**.

2.4.2 AXI4 & AXI4-Lite

AXI4 full (AXI4) is a complex interface, defined by the AMBA AXI Protocol Specification [7]. Its focus is on providing address-based reading and writing functionality, with the master initiating every transfer. While it focuses on throughput, just like AXI-Stream, the use of addresses always breaks down the data flow into discrete transactions. It can be split in two sections, the read and the write section, which can be broken down further into 2 and 3 channels respectively. Figure 2.6 shows the channels and their data flow direction. A channel in this context is akin to a AXI-Stream interface and follows the same handshaking rules. Each channel has a **VALID** and **READY** signal to facilitate this handshake. The read and write portions of a AXI4 interface can act completely independently and parallel to each other. If read/write ordering is desired, the master must ensure ordering of such transaction itself. Ordering of read/read and write/write instruction is part of the specification, with the included option of dropping the ordering

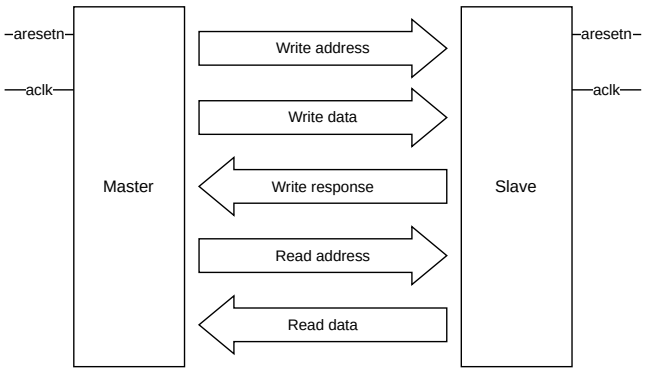

Figure 2.6: AXI4 channels as defined in the Protocol Specification [7, p. 29-35]

requirement for some or all transactions [7, p. 82].

A full read transaction begins with the master setting the appropriate signals on the read address channel and signaling this to the slave via **ARVALID**[2]. The slave should then acknowledge the address reception and internally retrieve the data associated with the received address. Once it does, or was unable to retrieve the data, it drives the appropriate signals on the read data channel and indicates this by setting **RDVALID**[3] HIGH. Depending on the read response included in the read data channel, the master can determine if the data transmitted is correct or wether an error has occurred. The specification calls for four different read response options, to indicate two cases denoting success, one with exclusive access, and two cases denoting failure, depending on whether an error occurred in the slave or an interposed interconnect module [7, p. 57]. To optimize transactions, AXI4 defines a burst mechanic to group multiple accesses together. Three schemes for address behavior in bursts are defined: fix address, incrementing address, wrapping address [7, p. 46]. Not only is the read address channel affected by burst, the read data channel must also signal the end of data associated with a burst. The AXI4 protocol further defines behavior pertaining to caching, quality of service, privileged access, reduced bus width and splitting the address space into regions. The protocol also mentions support for locked transactions but AXI4 has rescinded support for this feature over AXI3. All channels additionally define a **USER** signal which enables more custom data transmission.

The write section of an AXI4 interface differs from the read section through the addition of an additional channel from master to slave, the write data channel. Another channel enables the write section to transfer data in parallel or concur-

[2] AR stands for Address Read
[3] RD stands for Read Data

rently to the address. Equivalent to the read channel, a slave must acknowledge transfers from the master on the write address and write data channel. Once both the address and write data have been received, the slave must pass its write response back to the master. Signaling of write responses is the same as signaling of read responses and the write section supports the same features as the read section.

The AXI4-Lite interface is, as the name implies, a reduced version of the AXI4 interface. Quoting the AXI specification directly: "transactions are of burst length 1 [and] all data accesses use the full width of the data bus [...] of 32-bit or 64-bit" [7, p. 126]. Not only does a reduction of the feature set cut the required signals down to 21 from 47 signals[4], it also reduces the number of possible states a master and slave can inhibit. Additionally, AXI4-Lite allows slave components to ignore write strobes [7, p. 127].

A consequence of removing bursts is a reduced throughput in most cases. This is also acknowledged in the specification: "AXI4-Lite is suitable for simpler control register-style interfaces [...]" [7, p. 125]. XRT uses the AXI4-Lite interface to control kernel execution and provides the user application access to this interface, as well as parsing register addresses to register objects if the adresses are provided by the kernel developer. The v++ compiler automatically connects these control interfaces with an AXI interconnect by Xilinx and maps their address range into the address space accessible from the CPU via the XRT shell.

2.5 XRT

The Xilinx Runtime (XRT) is a set of components developed by AMD/Xilinx to manage devices with FPGA components. The following section is a non-exhaustive summary of the official XRT documentation [6] and will focus on its use with Alveo Cards, specifically the Alveo U50, which use PCIe to communicate with a host. As the name implies, XRT strives to create a runtime environment and standardize system development around it. To illustrate this further, figure 2.7 shows where XRT components are located in a system.

XRT manages kernels, a type of RTL module, executing on the FPGA. These modules follow a number of additional requirements when compared to generic RTL modules. It is possible to define these kernels in higher level languages which are compiled by high level synthesis (HLS) tools into RTL representation. However, this thesis mainly uses HDL to define kernels.

Modules are considered kernels if they meet the following requirements: At

[4] 50 when including the optional low-power interface

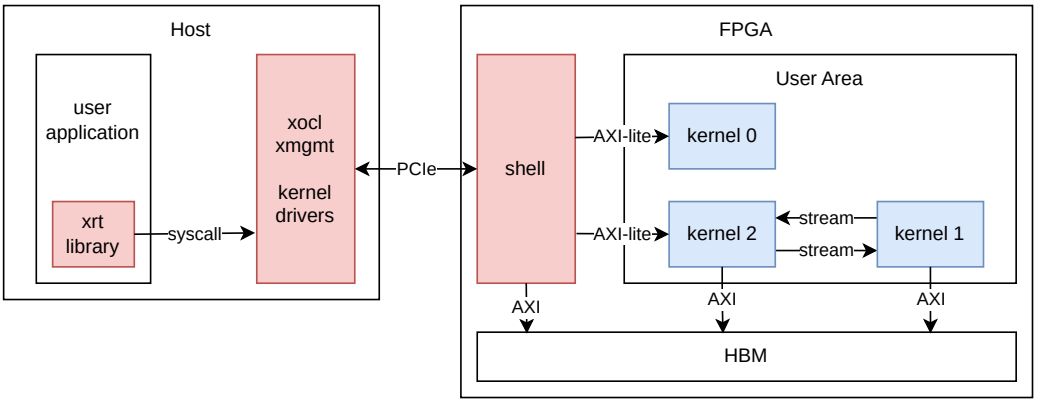

Figure 2.7: Block diagram representing example configuration of host and FPGA accelerator, XRT components marked red, kernels marked blue.

least one clock and an active-low reset input must be present. Depending on the execution model, which will be explained shortly, an interrupt and/or an AXI4-Lite control interface may be present. Additionally, the module may expose any number of AXI4 and AXI-Stream interfaces to communicate with other kernels or special components exposed by XRT on the FPGA.

XRT handles kernel management based on the execution model of the kernel. Kernels which take arguments, operate on them and return a result have the most software support. XRT automatically creates a function representation for them on the host when a binary containing them is loaded through the userspace library. In this case, XRT manages the complete execution, starting at the invocation from application software running on the host system. This execution flow can optionally be pipelined to increase throughput. Another execution model is the unmanaged execution model where a kernel only operates on AXI-Stream data. Because the kernels developed in this thesis require more complex execution handling, XRT consideres them as “user-managed” kernels. Control of such kernels is achieved by a user application writing and reading to and from registers via the AXI4-Lite interface. Compilation of the kernels into a binary is performed by the the v++ compiler.

To facilitate loading and execution of kernels, regardless of the execution model, the FPGA is divided into two partitions: Shell and User. The Shell handles PCIe communication and provides the environment for kernels to function. It is loaded from onboard flash memory at boot time and includes some runtime management functions. Xilinx calls this execution management inside the shell Embedded runtime (ERT). In contrast to other FPGAs, loading a compiled binary onto the User partition is done via the XRT driver on the host via PCIe, as opposed to a separate out-of-band programming interface.

Access from software to the kernels included in an XRT binary is done via the XRT drivers, which are split between two kernel drivers, xocl and xmgmt, and an user application library. Xilinx provides libraries for both Python and C++, both of which are used in this thesis depending on the performance requirements.

2.6 Related Work

Related works can be broadly categorized into two categories: FPGA networking, and consensus services.

2.6.1 FPGA networking

Characterizing Off-path SmartNICs for Accelerating Distributed Systems by Wei et al. [38] analyses the architecture and performance implications of SmartNICs, specifically Off-path SmartNICs and the NVIDIA Bluefield-2 data processing unit (DPU). Off-path SmartNICs are network interface cards (NICs) wherein the computational elements, most commonly a standalone system on chip (SoC), are not located on the critical path. A result of Off-path architectures is that three communication paths exist: Network-Host, Network-SoC, and SoC-Host. The Alveo U50, as well as the other products in the Alveo U family are, however, On-path SmartNICs. This implies that there may be a slightly higher base latency when no compute units are involved, but the data path is shorter when compute units are in use compared with Off-path SmartNICs. Additionally, the paper has a focus on remote direct memory access (RDMA) as the underlying communication primitive whereas this thesis focuses on user datagram protocol (UDP)/internet protocol v4 (IPv4) over Ethernet.

ACCL+ by He et al. [18] propose an FPGA-based communication layer for collective operations such as Broadcast, Scatter, Gather, and Reduce. They outperform classical message passing interface (MPI) RDMA communication in which FPGA-based services communicate over a seperate NIC and the host CPU. Another feature of ACCL+ is support for different transport protocols and a microcontroller in the control plane. As a consequence, their design is flexible and can be tuned to different collective programming without recompiling the FPGA bitstream. ACCL+ is similar to the work presented in this thesis, though it targets MPI-like primitives. MPI primitives have different communication patterns and optimization targets than the primitives addressed in this thesis.

nanoPU by Ibanez et al. [20] attempts to minimize network latency and overhead for CPUs by directly pushing socket data to a new CPU register. Their work derives from a RISC-V CPU and expands it by including a hardware scheduler to minimize overhead when switching between processes. Since a tape-out for

their modified architecture would be extremely expensive and time consuming, Ibanez et al. simulate their design on AWS FPGAs. They achieve a wire-to-wire latency of 69 ns in this scenario, assuming a 3.2 GHz clock. However, Ibanez et al. assume a direct link between their NIC, bypassing typical PCIe latency, and run their benchmark on bare metal without an OS, which typically introduces additional jitter. The only major similarity between nanoPU and this thesis is the attempt to reduce processing latency in the network context and the use of FPGAs, though nanoPU does not target FPGAs directly.

FPsPIN by Schneider et al. [33] implements the sPIN machine model on an FPGA to evaluate it on actual Hardware. The sPIN machine model consists of compiled handler functions which are executed on independent cores at packet reception. FPsPIN is thus an implementation of the sPIN framework for the design of algorithms and packet processing at a higher level. It might be possible to implement the algorithms used in this thesis using the sPIN machine model, but this amounts to a different architecture.

Corundum by Forencich et al. [12] is an FPGA-based SmartNIC. Its open source architecture and integration into the Linux kernel networking stack present a potential framework for this thesis. The complexity of integration into the project is a major reason why this thesis elects not to use Corundum. It may very well be possible to integrate the work presented in this thesis into the Corundum framework since it too uses packet based AXI-Stream primitives. Further study is required to definitively assert such a compatibility.

2.6.2 Coordination services

A seminal work in the field of DSs is the Paxos algorithm by Lamport [23, 24]. It introduces 3 roles which together arrive at a decision after two phases. The paper itself proves safety of the algorithm and introduces the assumptions under which it can make progress.

Raft by Ongaro & Ousterhout [28] is a consensus algorithm inspired by Paxos. A main motivation of Raft is the need for a simplified algorithm. To that end, raft splits consensus into three components: leader election, log replication and safety.

High-Performance Byzantine Fault Tolerant Consensus is a master thesis by Gebauer [13]. In his thesis, Gebauer implements a BFT consensus algorithm in C, designed with future offloading in mind. Profiling his implementation, Gebauer shows that the cryptographic functions heavily contribute to execution time.

RACS and SADL by Tennage et al. [37] propose splitting consensus metadata and data. Consensus rounds operate only on metadata referencing previously asynchronously distributed data. The focus of Tennage et al. is on their algorithm and its performance in a WAN setting and is implemented entirely in Software.

uKharon by Guerraoui et al. [15] is a Membership Service which uses RDMA semantics to detect application and OS failures, as well as a 1ms timeout to detect network failures. It uses a modified version of Paxos to provide user nodes a lease on membership. This lease is renewed periodically in the background while other user apps get membership status from that lease. uKharon algorithm and network primitive differ from this thesis and achieves the best performance known to date.

Consensus in a Box by István et al. [22] is an implementation of Zookeepers atomic broadcast [25] on an FPGA. It is a seminal work in the field of consensus implementations by leveraging the highly predictable execution of FPGAs and their networking capability. Their results are bottle-necked by the available bandwidth of the 10G network stack by the ETH Zürich Systems Group [34] and the FPGA used. This thesis uses a newer 100G network stack [35], of which the ETH Zürich Systems Group is also a contributor.

Waverunner by Alimadadi et al. [1] adapts the Raft [28] consensus algorithm by moving 3 functions of the Raft algorithm into an FPGA accelerator. These 3 functions allow their system to respond to client requests, log append messages, and the corresponding ACK on the FPGA during stable operation. Failure and recovery, as well as other states are handled entirely in software. In this sense if follows the general concept of Consensis in a Box [22] by leveraging the potential performance improvements of FPGAs. This allows Waverunner to achieve request rates of up to 26 mega packets per second (Mpps), bounded by the 100G FPGA Network connection. Waverunner is similar to the work in this thesis, though this approach differs by choice of algorithm. Due to the leader-based algorithm of Waverunner, it is limited by the bandwidth of the leader, a fact Alimadadi et al. already notice in their work. This thesis also expands on the scope by enabling FPGA based applications to leverage DS primitives.

Chapter 3

Design

Since this work implements a uniform consensus algorithm in the Crash Stop with Omission (CS+O) failure model with assumed synchrony [29, 31], we provide a brief overview below.

Synchronous physical clocks [9, p.46] assume a clock at every process with a bounded clock drift from a global clock. As long as a node operates without failure, it can execute a process in a bounded time, i.e. assuming *synchronous computation* [9, p.46]. Message transmission is also assumed synchronous with regard to omission failures [29, p. 3]. A message is either not delivered at all or has a bounded transmission delay. Omission occurs if a process A sends a message m along a link to process B but m is never delivered to process B. Such an omission failure can be transient as opposed to a process failure. This neatly translates to the use of UDP/IPv4 over ethernet.

This chapter first explains the algorithm, based off the algorithm by Parvédy and Raynal [29] and the changes by Rovelli [31]. After that, the design of an implementation framework is addressed as part of ***RQ2***.

3.1 Consensus

To simplify consensus algorithms, they can be split into an ordered sequence of rounds, each consisting of phases. Parvédy and Raynal [29] describe an algorithm, henceforth called Parvédy & Raynal Algorithm (PR-Alg) which does exactly that. It separates a single round into three phases: send, receive and computation, as shown in figure 3.1.

During the send phase, every node sends information about its current state to all other nodes. Every node then waits to receive the packets from the remote nodes. Finally, the computation phase decides whether to continue or a decision has been reached. The algorithm does not require the packets to either arrive in

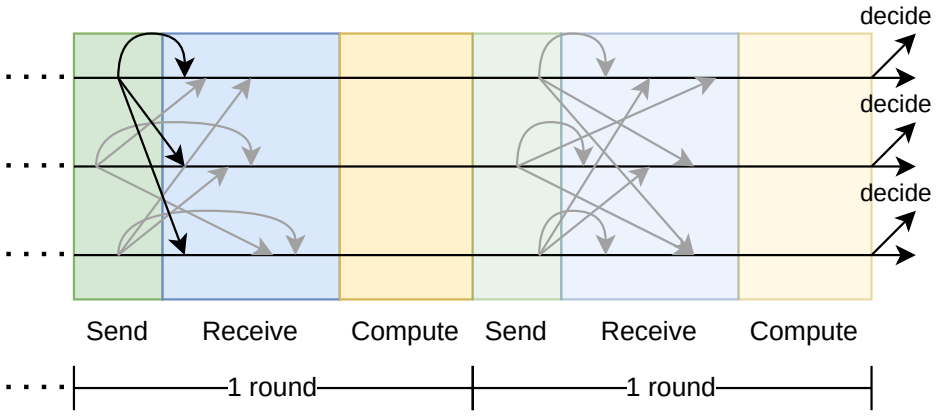

Figure 3.1: Round chaining for PR-Alg

some order or be processed in some order. Each node keeps a set of proposed values V, a set of suspected dead nodes $suspected$, a set of locked nodes $locked$, and the current round number r. The locking condition for nodes is highly intertwined with the theoretical proof of PR-Alg and will not be repeated here. Conceptually, a locked node has learned every value it expects to ever know, no matter how many more rounds follow this point. Once the number of locked nodes exceeds a separately defined quorum, the algorithm decides on the smallest value in the set V and terminates. The control flow of this algorithm is captured in figure 3.2. Their algorithm is also considered *uniform*, which implies no two processes decide on different values.

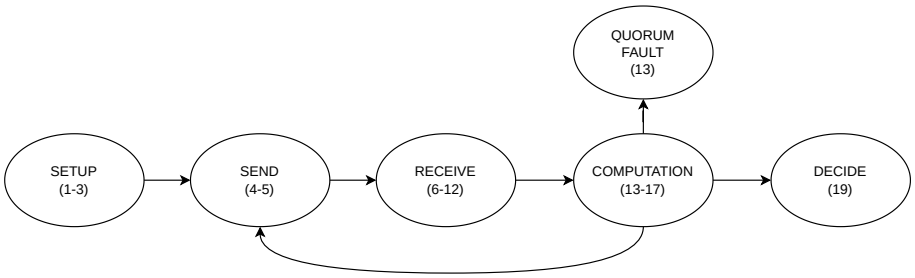

Figure 3.2: Control phases of the PR-Alg [29]

Rovelli modifies PR-Alg to create cool kids club (CKC) [31][1], shown in the appendix under 2. The biggest change in the behavior of CKC as opposed to PR-Alg is the value being decided. PR-Alg decides on the smallest value from V, while CKC decides upon V itself. Rovelli further changes the locking criterion as outlined in his work. As part of this, he changes the algorithm to operate on sets of live nodes instead of suspected faulty nodes. To facilitate a continuous operation, he also pipelines the algorithm to run repeatedly. Once a decision has

[1] Paper pending

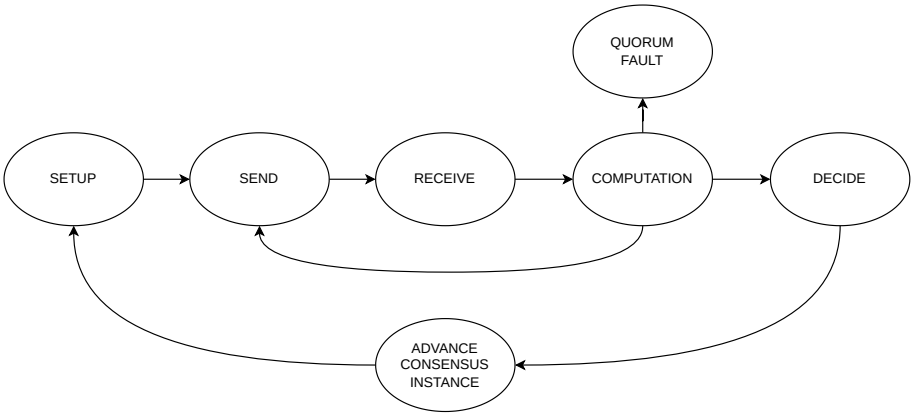

Figure 3.3: Round phases of the reduced CKC algorithm

been reached, the processes advance to the next consensus instance and reset the round structures. This change to execution flow is is illustrated in figure 3.3. There are also a number of changes introduced by Rovelli in regards to group membership. Since the logic underlying them is rather complex, we will focus on a simplified mode of operation first. This simplified version can be seen in algorithm 3 in the appendix.

3.2 Implementation abstraction

Observing the consensus algorithm presented in the preceding section, there are 5 interactions which represent interaction with outside components. Namely, this comes down to these main tasks:

1. Input (i.e. proposing a value)
2. Output (i.e. deciding on a value)
3. Message emission
4. Message reception
5. Time indication

Figure 3.4 shows the completed layout and modules which are required to perform these core tasks.

In our use case, proposition of values, as well as the decisions should be accessible for both applications running on the FPGA as well as applications running in software. AXI-Stream seems to best match this requirement since it is sequential,

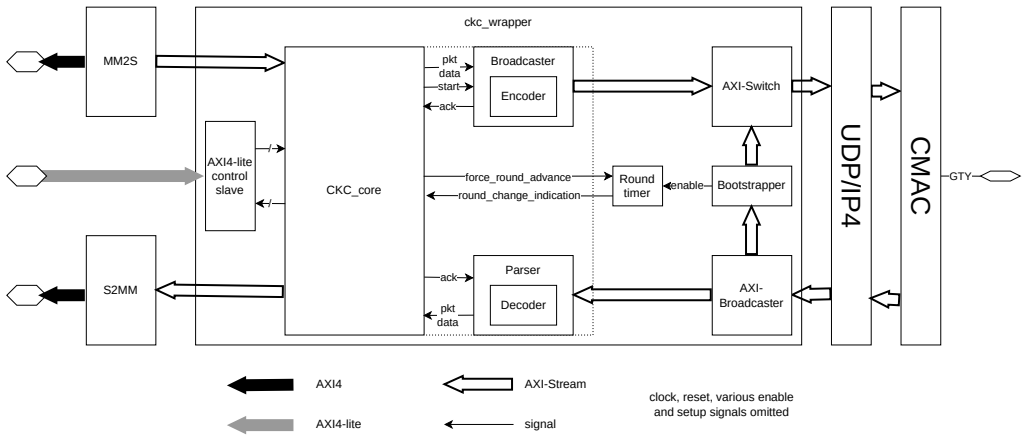

Figure 3.4: Implementation framework

has a performance focus and is limited in its complexity. Additionally, an instance can detect if the input does not have any data if there is no data pending on the input bus as explained in section 2.4.1.

If the output is not consumed fast enough, this can stall execution, which presents an issue. Hence we include the timely consumption of data as a requirement for services which leverage the framework. Consequently, an algorithm can assume failure of the local process if the output is stalled for too long.

The UDP/IPv4 layer described in 4.1.2 together with the packet injector module described in 4.3.3 provide a rudimentary primitive. UDP best represents the communication model in use by PR-Alg and CKC, since it does not contain implicit retransmission logic like transmission control protocol (TCP) which could incur additional delays, hampering the synchronous communication required by the algorithm. To provide a more elegant approach, the broadcaster module bundles a packet injector with an encoding module and emission logic.

By defining the encoding and decoding of full packets as separate modules, the *encoder* and *decoder*, we attain multiple benefits. The broadcast module can focus entirely on managing message targets and output semantics like backpressure, achieving modularity. Taking a number of inputs as bit vectors or arrays and packing them into a full packet can be error-prone. By defining the encoder and decoder together, ensuring that data is transmitted correctly becomes easier. Especially when verilog arrays are packed into a single bit vector, endianness can be a source of errors. Endianness describes the ordering of bits in a bit vector, i.e. whether the bit at index 0 is the highest valued or lowest valued bit. Additionally, the encoder can include, and the decoder can check for, identification of

a message.

The broadcaster simply takes a number of input variables, encodes them using the encoder and links the resulting packet into the packet injector. It then autonomously queues the packets to the UDP/IPv4 layer. The main algorithm is free to continue execution without having to immediately handle backpressure from the network. If the algorithm queues packets faster than the broadcaster can emit them, the algorithm can either slow down or fail.

Message reception logic also relies on the UDP/IPv4 and the CMAC as described in 4.1.2. Analogous to the broadcaster, packet reception logic can be greatly simplified if the algorithm only operates on information extracted from complete packets. To achieve this, we can wrap the decoder module with a separate state machine. This results in a parser module. It listens on an AXI-Stream for packets and either drops them if they do not match in length or the decoder deems them outside the message defintion. The parser only requires a simple handshake from the algorithm, akin to AXI-Stream as explained in 2.4.1, to indicate the algorithm has seen and processed the message before the next message can be received and decoded.

The final module consists of two somewhat separate components. If we look at the concept of a round indication mechanism, it must satisfy three requirements:

1. Provide a periodic and stable indication to the algorithm that a new round should begin

2. Ensure that nodes in a cluster do not drift too far apart

3. Start the cluster in sync to bootstrap cluster operation

The first requirement can be easily implemented using a counter, relying on the system clock to increment it periodically. This counter needs to be adjustable in order to vary the actual frequency of indication events. Algorithms can often discern if the local node is receiving messages from a subsequent round by consensus instance number and round number. If we provide a signal to indicate such a condition, the node can compensate by resetting the timer early. This can happen if the timer is not in phase with the other nodes or it experiences larger clock drift than anticipated. To counter this, the counter will adjust its period on such a signal, a trim which decays over subsequent rounds. While there are more elaborate methods to ensure little clock drift such as precision time protocol (PTP), this mechanism is simple and provides some stability.

Starting the cluster in a (somewhat) synchronous manner could also rely on PTP, but PTP is a complex protocol. The first attempt at implementing a bootstrapper module involved a packet injector and a receiver which listens for a defined packet. A single node is designated as the bootstrapper and must be started as the last node in the cluster. It then emits a special packet to all nodes, including itself. Once the packet is received, a node will start its timer and indicate this to the algorithm as a round change indication. Since the setup involves a single TOR-switch, the latency experienced by each node should be very close, as can be seen in the latency evaluations. There is, however an issue with the current setup. Some component filters out the packet that the bootstrapper sends to itself. It is not entirely clear whether this is caused by a lower level function of the 100G MAC (CMAC) or the filter rules of the switch employed in the cluster. The CMAC shows the correct number of emitted packets. The switch does not register a packet drop and exhibits this behavior even with the corresponding filter rule disabled. Since observing transmission on the wire is prohibitively complex and expensive, this problem is more easily solved by designing around it. Since the latency is extremely stable, the local node can be started by setting a timeout equivalent to $\frac{1}{2}$ of the round trip time (RTT).

Since the network is being used both by the algorihm and the bootstrapper, two additional modules are required. An AXI-Stream Broadcaster and a AXI-Stream switch to duplicate and merge the network streams respectively. Finally, the wrapper includes an AXI4-Lite control slave in order to manage the system from software.

These design choices result in the design as visible in in figure 3.4. To answer ***RQ2***, by providing I/O abstractions to applications and the network, as well as a round indication mechanism with feedback, a synchronous distributed system protocol can be implemented by transforming the control flow into a state machine. The abstraction of I/O to simple events especially reduces the states required as all message data is presented during one cycle. Defining the encoding and decoding of information into the message together, yet as separate modules, increases modularity and decreases susceptibility to parsing errors.

Chapter 4

Implementation

The implementation of FPGA functionality is directly based on the Github project xup_vitis_network_example [30]. It is provided by Xilinx as an entry point and example for leveraging the networking functionality of their Alveo FPGA accelerator cards. Naturally, it is built in the Xilinx Runtime. Since it was developed withing the Xilinx University Program, there are a number of different contributors outside Xilinx. Namely, these are the Systems Group of ETH Zürich, Switzerland and HPCN Group of UAM, Spain [32, 35]. There are three licenses covering the whole repository, all of them BSD 3-Clause Licenses.

The chapter is structured as follows: First, the network stack inside the FPGA is addressed, followed by considerations for the design of modules. Following that, a number of modules whose functionality is not directly tied to the CKC implementation, but are used as components, are introduced. Finally, the implementation of CKC is covered.

4.1 Network stack

Networking functionality is split between two distinct units, the CMAC kernel and the UDP/IPv4 kernel. This split occurs between OSI layers 2 and 3, the Data Link layer and the Network layer respectively. In this case, the UDP/IPv4 kernel handles both the network and transport layer, effectively merging layer 3 and 4. Since this implementation only uses UDP as a transport protocol, a merging of multiple layers is possible. On a higher abstraction level, the whole network stack provides packet based network input and output to and from a set of predetermined targets. Any other network implementation that implements such an abstraction can also be employed as an underlying network primitive for this work.

4.1.1 CMAC

Media access control (MAC) is handled by the UltraScale+ Devices Integrated 100G Ethernet Subsystem by AMD/Xilinx [2]. It requires a license to be installed in the compiling system which is provided free of charge by AMD/Xilinx. It connects directly to the network transceiver wires and exposes two AXI-Stream interfaces as well as an AXI4-Lite interface for statistics and control register access. One AXI-Stream interface is a TX slave and the other is a RX master.

The Ethernet subsystem is encapsulated in the 100G MAC (CMAC)[1] kernel to provide further functionality. This Kernel is provided as part of the `xup_vitis_network_example` [30]. It includes frame padding and clock domain crossing for the stream interfaces since the Ethernet subsystem is clocked at 322.265 MHz, while other kernels are clocked at the standard 300 Mhz. Both the TX and RX AXI-Stream interfaces have a data width of 512 bits. Actual data transfer to and from this kernel is in the form of raw Ethernet frames.

4.1.2 UDP/IPv4 layer

Operating on raw Ethernet packets coming and going from and to the CMAC kernel can become exceedingly complex. The project `100G-fpga-network-stack-core` [36], by Sutter et.al. [35] and Ruiz et.al. [32], provide further abstraction. It implements UDP/IPv4 functionality and exposes this to other kernels through two 512-bit AXI-Stream interfaces, one for TX and one for RX. The theoretical maximum bandwidth of these internal interfaces at clock frequency of 300MHz is given below.

$$512\text{bit} * 300\text{Mhz} = 153.6\text{Gbps}$$

The payload interface bandwidth of the UDP/IPv4 layer is thus sufficient to saturate the 100 Gigabit/s network bandwidth. How long the bus can be stalled is directly dependent on the packet size. Taking a 64 byte payload (I) and a 1408 byte payload (II) as worst- and best-case, the AXI-Stream interface is at the following utilization:

$$\frac{100\text{ Gbps}}{(64+8+20+38)\text{ byte} * \frac{8\text{ bit}}{\text{byte}}} \Big/ \frac{153.6\text{Gbps}}{64\text{ byte} * \frac{8\text{ bit}}{\text{byte}}} = 32.05\% \tag{4.1}$$

$$\frac{100\text{ Gbps}}{(1408+8+20+38)\text{ byte} * \frac{8\text{ bit}}{\text{byte}}} \Big/ \frac{153.6\text{Gbps}}{1408\text{ byte} * \frac{8\text{ bit}}{\text{byte}}} = 62.20\% \tag{4.2}$$

This assumes a UDP header of 8 bytes, an IPv4 header of 20 bytes, and an Ethernet overhead of 38 bytes per packet, as well as a 100 Gbps physical link.

[1] The roman numeral C to indicate 100Gbps.

Broken down into its components, Ethernet includes a 8 byte preamble and start of packet sequence, 14 byte header, 4 byte frame check sequence, and 12 byte interpacket gap.

The UDP/IPv4 layer further includes an ICMP echo block which greatly simplifies network configuration and associated debugging. ICMP echo is more commonly known as a simple ping message. Since the UDP functionality is based on IPv4 addresses as opposed to MAC addresses, an address resolution protocol (ARP) translation block is included as well. It is possible to both insert up to 256 entries manually and use automatic ARP discovery. [30, /NetLayers/README.md].

Actual UDP sockets are defined by their entry in the socket table and consist of 3 entries and a valid bit. The 3 entries are the remote IPv4 address, the remote port, and the local port. The local IPv4 address must be set separately and is the same for all sockets. All received messages belonging to a socket are stripped of the Ethernet, IPv4 and UDP header and only the UDP payload is pushed to the RX stream. To distinguish between the different sockets on the RX and TX stream, the **TDEST** value of the stream is used to identify the socket.

The project includes a minimum viable configuration to send and receive data from the host CPU. This is accomplished by two kernels, `mm2s` and `s2mm`, which bridge the stream-oriented nature of the UDP/IPv4 layer and the memory-oriented structure of Host-FPGA transfers. The `mm2s` kernel takes a buffer handle, size of the handle, and a destination socket number. It then sequentially reads from that buffer and outputs the data to its AXI-Stream master interface. Since the UDP/IPv4 layer requires packet separation, it marks every transfer including the 1408th byte with **TLAST** to indicate the end of a packet. A packet size of 1408 bytes is close to the maximum transmission unit (MTU) of UDP over IPv4 over Ethernet, yet cleanly divides by up to 128 without remainder. This simplifies packetization in the `mm2s` kernel since it would need to account packet boundaries not being aligned with AXI-Stream transfers. In the case of 512-bit wide streams, this constitutes a packet transmission every 22 cycles. Reception is even simpler. The `s2mm` kernel writes all incoming data to a buffer until the buffer is full. In cases where the buffer is not fully filled, the `s2mm` kernel stalls but it is still possible to read the partially filled buffer from software. The kernel does not have its own timeout mechanism to indicate this, and calling the appropriate wait function never returns until the buffer is filled completely or the thread is interrupted otherwise.

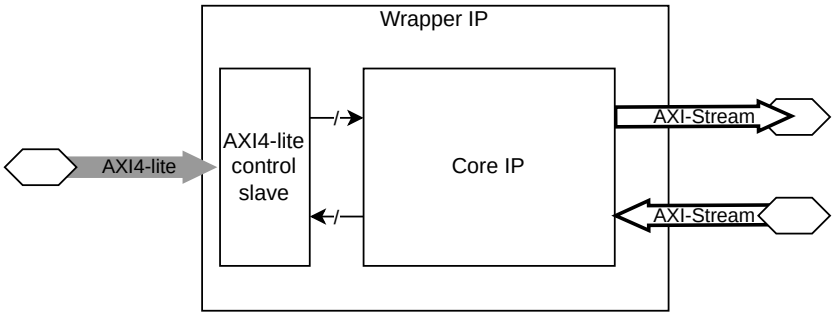

Figure 4.1: Kernel design with wrapper, control slave and core IP

4.2 Modular kernel design

Kernels, like all modules, need to be tested via simulation, which entails input simulation. XRT requires their control and status registers to be accessible through an AXI4-Lite interface. Since AXI-Stream is complex, both in design and simulation, even when employing the associated VIP, it makes sense to separate core functionality from the AXI4-Lite control interface. This separation is shown in figure 4.1 and enables simplified development and simulation of the core IP module. The wrapper module can focus on conforming to the XRT specification, while the core module does not need to implement any AXI4-Lite logic. In general, the design flow for a kernel becomes twofold:

First the core module is created without any regard to the XRT environment. The core module definition can include singular wires and normal bit-vector inputs and outputs. Linking these is not directly supported by the v++ compiler in an XRT-based project and would need to be done with tcl scripts. Doing so fragments linking information and complicates the project setup significantly.

Once the core module is implemented and simulated, a wrapper module is created which joins the core module with an AXI4-Lite slave module and exposes the required ports. In the beginning, an existing AXI4-Lite slave from the wb2axip project was used [14]. Since this code is licenced under the Apache-2 Licence, a new, but slightly less performant AXI4-Lite slave was created. This slave (`axi_lite_control_slave_base`) was implemented against the official AXI4-Lite specification [7] and simplifies adding new registers. The wrapper module also ensures that only XRT-compliant ports are exposed.

Using this hierarchy enables faster development, since the core module itself can input and output from and to simple bit-vectors. Designing simulations which directly operate on bit-vectors is syntactically shorter and more precise than using an AXI4-Lite bus. The wrapper can also be tested in simulations but the interaction through AXI4-Lite interfaces requires the use of AXI VIP provided in each installation of Vitis. The AXI4-Lite VIP provided by Vitis does have some draw-

backs. There is a significant lag between issuing a transaction in the simulation and the transaction occurring on the bus. Simulating with direct access to input and output registers is therefore preferred over the use of VIPs. Lastly, the separation of core module functionality and XRT compliance also enables much easier reuse of existing modules. If all modules were to use AXI4-Lite exclusively for control, each parent module that intends to use a child module would need to have their own AXI4-Lite logic, complicating the design.

4.3 Utility modules

4.3.1 Timestamping

To assess the latency characteristics of the network layer, two kernels were implemented to handle timestamping, shown in figure 4.2. The first kernel is called the time generator (`time_gen`) and consists of a counter whose value is simply exposed on a number of 64-bit AXI-Stream interfaces. These interfaces do not implement the **TREADY** signal as their sole use is to transmit the current time to timestamping blocks. Adding the capability of back-pressure in this case is not necessary, since the value of the counter changes every clock. Both stopping the clock and transmitting outdated timing information is antithetical to the purpose of these kernels. As a consequence, the bus value needs to be sampled in one clock cycle, since reading the bus in two cycles would introduce race conditions.

The only block to consume this AXI-Stream is the timestamping block (`time_inserter_adjustable`). It combines two modules, a skidbuffer and an AXI4-Lite control slave. The timestamping block modifies the skidbuffer to take the time information from the time generator and inserts the value at a given byte offset into each packet. Any data at this position is simply overwritten. This offset is adjustable via two control registers, accessible through the AXI4-Lite control slave. The advantage of this approach is that the length of the packet is not altered. Since this operation is performed in a cut-through manner, the latency introduced by the timestamping module is between 1 and 2 clock cycles, depending on the state of the skidbuffer. Internally, the offset calculation is performed by counting the number of transfers and calculating the offset into a 512-bit transfer. To reduce design complexity, this block does not support sparse transactions, meaning that it assumes that **TSTRB** and `TKEEP` are all HIGH, except for the end of the packet. It also does not have support for timestamp insertion which crosses transfers. A timestamp starting at byte 62 and ending at byte 70 is thus not supported.

Since the clock frequency of the design is known at compile time and can also be roughly inferred from software, this enables timestamping of packets. This includes incoming and outgoing packets. Stream routing and UDP socket infor-

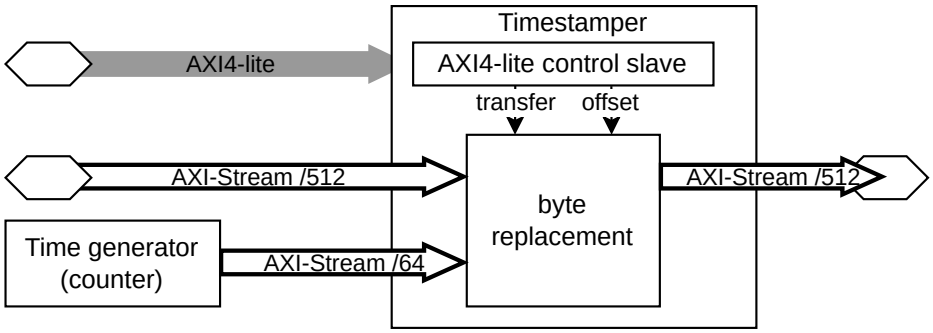

Figure 4.2: Clock diagram of timestamp inserter and time generator, additional /64 AXI-Stream interfaces omitted.

mation is not disturbed as all, signals besides `TDATA` are passed through the block unmodified, be it delayed by one to two clock cylces.

One critical aspect of this design is that the inserted timestamp value for a given packet slightly varies by its insertion offset. Since only 512 bits are transferred at a time, the value inserted increases for every 64 increments of the offset variable. If the bus is stalled during a packet transfer, this time difference will increase even further. Since the kernels are clocked at 300Mhz by default, this means that during ideal conditions, the timestamp will be inaccurate w.r.t. the begining of a 1536 byte packet transmission by up to 160ns as calculated below.

$$\texttt{Inaccuracy} = \frac{(1536 * 8)\texttt{bit}}{512\texttt{bit}} * \frac{1}{300\texttt{Mhz}} = \frac{48}{300\texttt{Mhz}} = 160\texttt{ns}$$

As long as timestamps are inserted in the same 512-bit aligned block, there is no inaccuracy when comparing them. Since a single cycle at 300Mhz takes $3.\bar{3}$ns, inaccuracies do not grow very quickly, given that the offsets are close to each other.

For this reason, the timestamping block also technically violates the AXI-Stream specification. If a slave stalls the bus while the master has **TVALID** HIGH, `TDATA` is not allowed to change. Since the timestamping block inserts the time of the actual transfer, `TDATA` changes every clock. This has not posed a problem in the current designs but should be taken into consideration when using other AXI-Stream IP.

4.3.2 Control slave

A very common use case in the design of XRT kernels is reading and writing to single registers for control purposes. During development, an open-source tem-

plate by Gisselquist Technology, LLC [14] was employed. Due to license concerns, the `axi_lite_control_slave_base` module was developed by adhering to the official AXI4-Lite specification [7]. The module code makes it simple to add new registers by separating all control signals from the actual data transfer. For reads, this means that it saves the incoming address in `outstanding_read_adress` and sets the flag `outstanding_read`. The `rdata` output register must be set to the correct data in the following cycle if the `outstanding_read` flag is set. Similarly, during an incoming write request, both the write address and the write data are saved and acknowledged on their respective sub-bus. Once the flag `outstanding_write_both` is set, implying both an address and it's associated write data are present, the module can simply save the value in a register or perform arbitrary actions. It is thus possible to implement logic other than simple reads and writes, such as a single cycle trigger whenever a read or write to a particular address occurs. Care should be taken when implementing special behavior as this might break expected behavior for some registers, e.g. a read after a write returning unexpected data.

This particular implementation has the drawback that a transfer can take multiple cycles. Since the module is only employed for low-throughput control signaling, this limitation does not influence performance of the overall system.

To further simplify the design, the module chooses to ignore all write strobes and assumes that all writes use the full data bus. This behavior is permitted by the standard [7, p. 127]. The module can be customized to any AXI4-Lite address width bigger than 2. An address width less or equal to 2 would clash with the decision to ignore all write strobes. Another simplification is that the standard behavior for non-implemented addresses is to treat them as 0-read-only registers. If such an address is accessed, the module responds normally with an `OKAY` response on both the read and write channel. The XRT API is aware of the custom registers, given that their addresses and widths are entered under the "Addressing and Memory" tab when packaging the module in Vivado. Even when the XRT API is not directly aware of which registers exist, entering raw addresses is still supported in software.

4.3.3 Packet injector

Since the network stack and most other kernels transfer packet data with 512-bit wide AXI-Stream interfaces, injecting a packet into an AXI-Stream is a recurring task. Since packets are usually longer than 64 byte, they require multiple cycles to be transmitted over an 512-bit AXI-Stream interface. The packet injector implements the logic needed to sequentially transmit the 512-bit chunks of a packet. Figure 4.3 depicts a schematic representation of the module. It is parametrized in the number of 512 bit chunks of a packet, a design decision which has not posed a

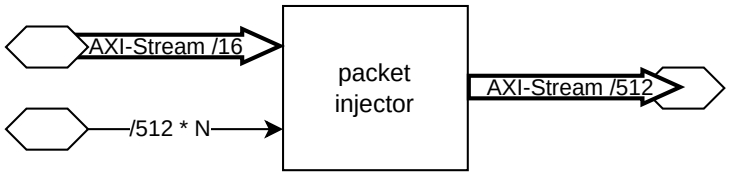

Figure 4.3: Schematic showing the packet injector IP module.

limitation for this thesis, but reduced complexity. The data that is to be transmitted in the packet can be set in a simple bitvector which defines the entire packet.

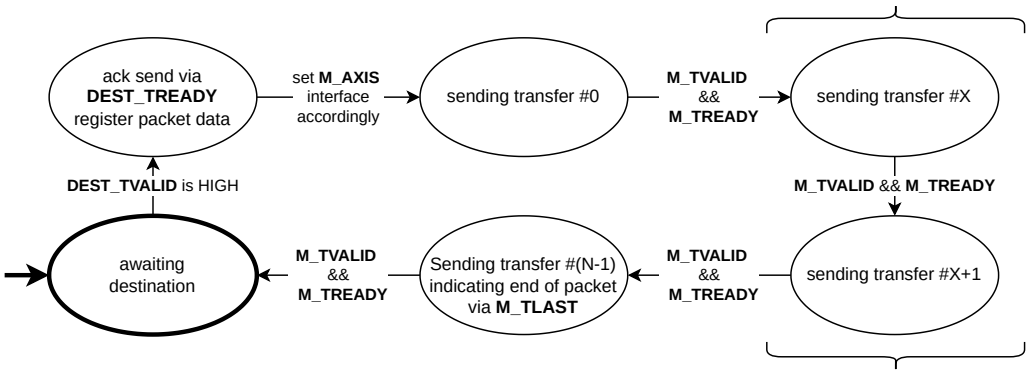

Figure 4.4: State machine of the Packet Injector module. N being the number of 64 byte chunks making up the packet. $X \in [1, N-2]$ to represent variable number of states, shown in brackets. Note that in the case of N = 1, there is only one transmission state, which sets **M_TLAST** to HIGH.

The state machine which defines the Packet Injector module can be seen in Figure 4.4. Triggering a packet transfer and defining the target socket are highly coupled in the **TDES_AXIS** AXI-Stream interface. This stream has a 16 bit width, the same as **TDEST** of the Network layer. Once a transfer occurs on this bus, the Packet Injector starts the injection process to the socket specified in the transfer and does not acknowledge any subsequent transfers on that bus until a packet has been sent out the main AXI-Stream interface. If no backpressure on a send event is desired, a module using the packet injector must simply add a FIFO of sufficient depth on the **DEST_AXIS**.

Once a transfer on the destination port is acknowledged, the packet data input is saved in an internal register. Not registering the packet data could lead to race conditions where a send event is acknowledged on the **DEST_AXIS** interface but actual transmission on the main bus stalls. If the module then changes the packet data to a subsequent send event, the injector would sample this data instead of the proper payload.

A minor deficiency of this module is the fact that it idles the main output for 2 cycles between packet transmission. To facilitate testing and debugging some components on the FPGA, there is a kernel which wraps the packet injector. This enables setting packet data and triggering the injection of a packet from the host via XRT.

4.3.4 Xilinx IPs

Since the AXI-Stream specification has been adapted in the Xilinx FPGA ecosystem, Xilinx provides a number of AXI-Stream infrastructure IP modules. This thesis uses a number of these IP modules. Namely, the AXI-Stream Data FIFO, the AXI-Stream Broadcaster, the AXI-Stream Switch and VIPs for AXI-Stream and AXI4-Lite. These cores can be instantiated in a Vivado project by TCL scripting or using the Vivado GUI. Their behavior is defined in the AXI-Stream Infrastructure IP Suite [4].

The AXI-Stream Data FIFO is a first-in-first-out buffer for AXI-Stream interfaces. It can be configured to different interface setups, clock domains and buffer depths of powers of 2. The fill level of the FIFO needs to be read from different interfaces, depending on whether data should be enqueued or dequeued. This is because the internals of this module require a number of cycles to propagate transfers from the input to the output stream. A central fill level indicator could cause race conditions when a core writing to the FIFO expects a level of backpressure. This IP is used to buffer send requests, as well as input and output data in the CKC implementation.

The AXI-Stream Broadcaster is a very simple module. It has a single input stream and a variable number of output streams. Its main feature is the replication of stream data. As a consequence, the input stream can only advance once a transfer has been acknowledged by all attached slaves. If continued data flow is desired in the case of an error condition, slaves attached to this module core should continue acknowledging transfers even if they discard the data. This is important when multiple modules are monitoring a stream for particular data, e.g. a special packet as in the case of the bootstrapper module.

The AXI-Stream Switch is a generic interconnect IP module. As such, it has a variable number of input interfaces, as well as a variable number of output interfaces. Routing of data from an input to an output interface can be done with registers or by leveraging the **TDEST** signal of the input. The specification does mention that there is a limitation to this module regarding throughput. When an input stream continuously writes to an output, it gets stalled after each transfer [4, sec. Data Flow Properties]. It follows that setups in which this property constitutes a bottleneck must either design their own switching solution or rethink the datapath. The module is also used in simulations to mirror the functionality

of an Ethernet switch without simulating the entire network stack. This is particularly convenient as cores using the socket abstraction provided by the UDP/IPv4 layer already tag their traffic with a **TDEST** signal.

Verification and driving of AXI-Stream and especially AXI4-Lite interfaces from scratch is complex and error-prone. Xilinx provides the `axi4stream_vip` and `axi_lite_vip` VIP to support the development of such simulations. Some of the functionality is documented in the AXI Verification IP LogiCORE IP Product Guide [5]. Both VIPs are able to act as a master, slave, or pass-through and warn the user when violations of the respective interface specifications are detected. When acting as a master, both VIPs can generate transfers on the bus. The documentation on the API used to initiate transfers is somewhat sparse for both the AXI-Stream and AXI4-Lite VIPs. Additionally, the Vivado IDE does not recognize the package imports required for the VIP drivers and flags them as errors. When issuing a transfer from the simulation, the actual queuing behavior of that command inside the VIP drivers, as well as the latency of that command is, again, somewhat nebulous. For AXI4-Lite transactions, this is not an issue for this thesis as the interface is lower speed by design. When looking at AXI-Stream interfaces, this can result in simulations with higher backpressure and idle times. In these cases, the AXI-Stream transfers are generated by driving the signals directly.

4.4 CKC - Consensus

Taking the graph representation of the CKC algorithm from section 3.1, we can expand it to include the operations needed in each state. The resulting graph can be seen in figure 4.5. Shifting the computation to the beginning of a round simplifies round timing. A round change indication during the listen state is the most appropriate since it is the only phase whose length should vary with different round periods. As implemented, computation and broadcasting always take a fixed time, depending only on wether the round transition includes a consensus instance transition.

Representing the receive logic as a control graph is cumbersome due to its nested clauses. It is shown in pseudo code in Algorithm 1 and is encapsulated in a single state. There are essentially 8 cases, leading to 3 different state transitions, forcing a round transition (A), forcing an early consensus transition (B) or indicating an error (C). It is also important to understand that a round change indication forgoes the receive logic entirely and directly forces transition **A**. In essence, the state machine processes packets up to the point when it receives the round change indication.

The implementation is parametrized on the number of nodes. When more than 8 nodes are required, some changes to the packet structure might be required. To

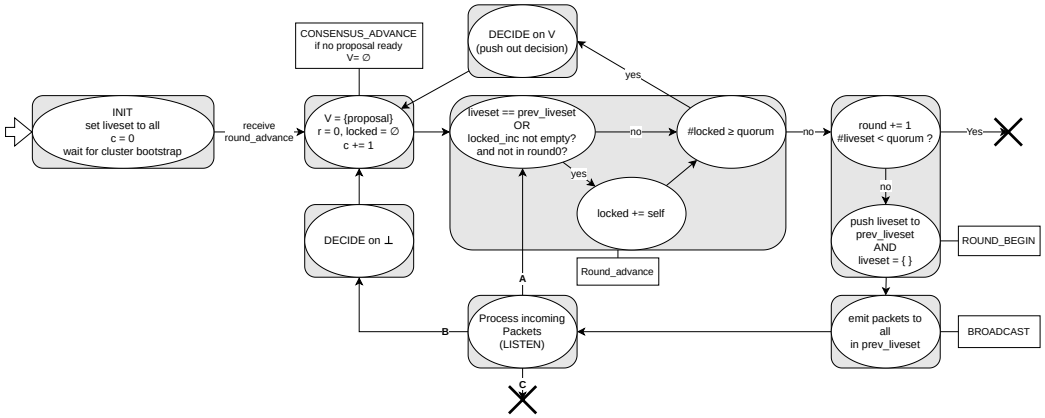

Figure 4.5: CKC control graph implemented in `ckc_core`

represent sets containing per-node data, a bitmap is employed. This bitmap has a fixed width of 8 bits in packets. When expanding to more nodes, those fields must grow together with the number of nodes.

As implemented, values are 64-Byte values. This greatly simplifies handling and parsing since a value can be completely transmitted or ingested in a single cycle over a 512-bit AXI-Stream. Expanding this to larger values is possible, however, there is a limit to individual packet size because of the underlying data link layer. Taking a the MTU of Ethernet II, this limits the UDP payload to 1472 bytes.

The scalar inputs like the local node id and the required quorum are exposed as bit-vector inputs. These, along with indicators about current liveset and current consensus instance are available to read and write through the wrapper as explained in chapter 3.2.

Algorithm 1: Receive logic with regard to figure 4.5

```
1  upon RECV(p_inc, r_inc, c_inc, locked_inc, V_inc):
2      if p_inc ∉ prev_liveset then
3          NEXT_PACKET // drop packet
       else
           if c > c_inc or r > r_inc then
4              NEXT_PACKET // drop packet, it is outdated
           else if c + 1 == c_inc then
5              FORCE_ROUND_INDICATION
6              TRANSITION(B) // pkt retained for c+1
           else if c + 1 < c_inc or r + 1 < r_inc then
7              TRANSITION(C) // Error out, local node too far
                  behind cluster
           else if c == c_inc then
               if r + 1 == r_inc then
8                  FORCE_ROUND_INDICATION
9                  TRANSITION(A) // pkt retained for r+1
               else if c == c_inc then
10                 V ← V ∪ V_inc
11                 locked ← locked ∪ locked_inc
12                 liveset ← liveset ∪ {p_inc}
```

Chapter 5

Evaluation

RQ1 and ***RQ3*** require evaulation of the FPGA, as well as the CKC implementation. An answer to ***RQ1*** consists of an evaluation of the three main cmmunication methods from host to card and vice versa, as well as the network latency. ***RQ3*** requires an understanding of the processing latency and stability of the algorithm at different round periods. With this information, the data rates associated with the CKC implementation are calculated and comparisons to Conensus in a Box [22] and Waverunner [1] are drawn.

Evaluation was performed on a 3-Server cluster at USI. All three are Supermicro SYS-120U-TNR systems with Dual Xeon Gold 5315Y processors and 190GiB DDR4 RAM. They are connected to an Edgecore Wedge100BF-32X-O-AC-F tofino switch via Mellanox 100GbE ConnectX-7 SmartNICs and 100G DAC QSFP28 cables. Each server contains an AMD Xilinx Alveo U50 which is also connected to the switch via a 100G DAC QSFP28 cable. The Alveo U50 card is equipped with 8 GiB of high bandwidth memory (HBM) memory, split into 32 256MiB chunks or partitions. Remote access to the servers is achieved via a separate switch using the inbuilt RJ45 connectors. Linux 5.15.0-125 is employed on all host systems, together with XRT version 2.16.204, which constitutes the 2023.2 branch. The xocl and xclmgmt drivers are also using version 2.16.204. Vitis 2023.2, which includes Vivado and the v++ compiler, is used to compile and package all binary images targeting the FPGA.

The switch is configured to treat all connections as belonging to the same virtual local area network (VLAN), in particular a /24 subnet. Low power mode and Reed Solomon Forward Error Correction (RS-FEC) is disabled for all interfaces. Autonegotiation must be turned off for the interfaces with FPGA connections since the free license of the Ethernet Subsystem does not include link speed negotiation.

Scripts to perform experiments and gather data are written in Python and leverage Ansible to manage multiple nodes. If not specified otherwise, the evaluation

of generated data is done in Python using Jupyter-lab. The matplotlib, pandas, and numpy libraries are used for plotting and data analysis during evaluation. This chapter initially covers system latency and its constituent components. After that, the CKC implementation is tested and performance bottlenecks are determined.

5.1 System latency

The overall system latency can be split into 3 parts: The software processing and Host-FPGA latency, the FPGA processing latency, and network latency. Software processing and Host-FPGA latency are grouped together since much of their jitter is incurred on the CPU. It is also the latency limiting software interactions with the FPGA-based services. FPGA processing latency depends on the system being tested, and is highly predictable through simulation. Network latency is the same for all FPGA services and is tested independently from them. The following sections therefore look at Host-FPGA latency and Network Latency while FPGA processing latency is addressed in the sections respective to the tested systems.

5.1.1 Host - FPGA

An important aspect is the interaction of the FPGA and the Host system since this can stall input and output pipelines, affecting the correct behavior of the consensus service. We analyze three different interactions provided by XRT: singular AXI4-Lite transactions, XRT kernel executions, and buffer syncing. Understanding the performance characteristics of these interactions is critical when attempting to answer ***RQ1***.

AXI4-Lite transactions

Simple reads and writes to single AXI4-Lite registers are atomic on an RTL level. To measure the read and write latency, an image consisting of a single kernel is used. This kernel implements a 32-bit counter which is incremented every clock cycle. At a clock speed of 500MHz and 300MHz, this counter overflows in 8.6 seconds and 14.3 seconds respectively. The kernel exposes this counter on address `0x0004`, an internal r/w register on address `0x0000`, and a magic number, in this case `0xeeffaabb`, on all other addresses. To measure the latency of the AXI4-Lite transfers, a simple C++ program was written using the C++ XRT application programming interface (API). Measuring the latency is also possible using the python API, but C++ was preferred due to the interpreted nature of Python, which can introduce overhead and timing inaccuracies. The `chrono` library, which is part of the C++ standard library, allows for reads and writes to be

timestamped with nanosecond resolution. In order to limit pipelining and batching inside the XRT API, a delay of 100 milliseconds between accesses is employed. To ensure the XRT API does not return cached data, an internal counter is used. An increment on each access ensures the data is actually read from the kernel. As long as subsequent accesses to the counter are more frequent than the overflow interval, the overflow can be accounted for in software trivially.

The first access to a kernel, whether a read or a write access always takes 54μs. Since this heavily skews the data, the first sample point has been excluded in all calculations and graphs. Between experiments, this initial delay only fluctuates by around 1 μs. Such a pattern suggests that either the XRT library, its kernel drivers, or the shell are initializing data or control structures.

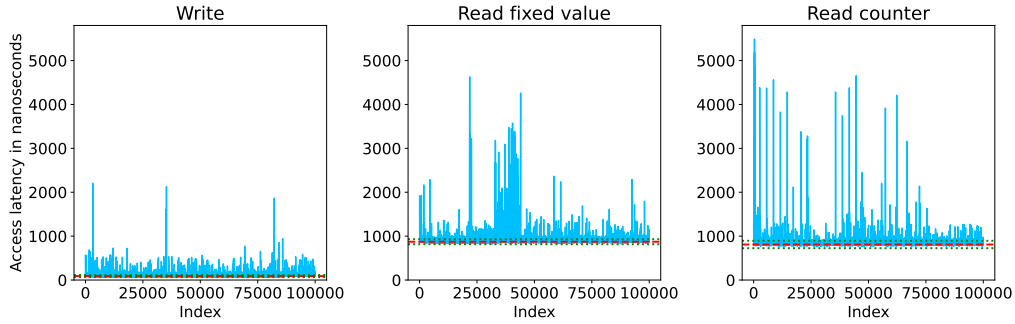

Figure 5.1: Plot of access latencies for write and read operation, first entry omitted. Mean shown in red, $\pm\sigma$ in green.

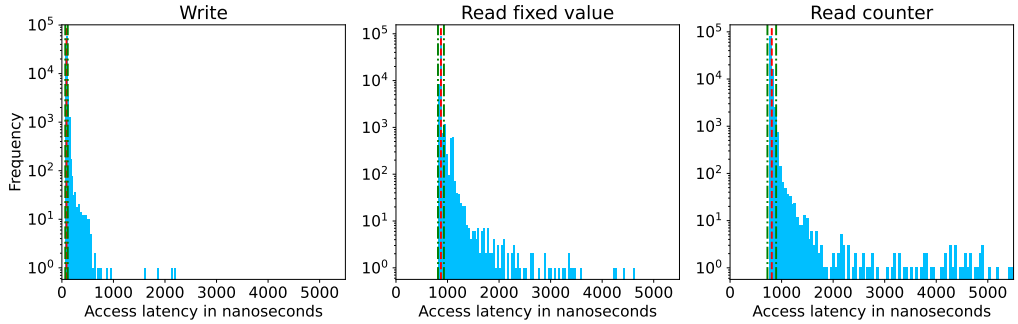

Figure 5.2: Histogram of access latencies for write and read operation, first entry omitted. Mean shown in red, $\pm\sigma$ in green.

All latency plots shown in Figure 5.1 exhibit outliers with multiple times the mean value. Since the mean and standard deviation are hard to set in context at this scale, mean, standard deviation, as well as skewness and curtosis are shown

—	Write	Read fixed value	Read Counter
mean	89.91 ns	872.60 ns	809.48 ns
σ	24.13 ns	57.64 ns	84.56 ns
skewness	25.83	26.37	35.54
kurtosis	1665.2	1115.0	1495.8

Table 5.1: Statistical variables for Host - FPGA AXI4-Lite transaction latency

in table 5.1. Looking at the histograms in figure 5.2, this is reflected in the observed distribution exhibiting long tails and high kurtosis. This makes AXI4-Lite operations unsuitable when stable, low-latency access is required.

AXI4-Lite-writes seemingly outperform reads in overall performance. Since a mean latency of 89 nanoseconds seems unlikely, we should assume that the XRT library features an early return model. A write transaction is queued and the library call returns immediately. While this invalidates the results for writes, they give an insight into the library overhead. Since AXI4-Lite-writes are subject to the same latency spikes as reads, a likely assumption is that these are caused by the XRT library performing syscalls which can be delayed by the OS.

With the early-return functionality for writes, a write latency cannot be determined directly. Logically, the actual write latency should be somewhere around half of the read latency since only 1 PCIe Transfer is required in this context, as opposed to 2 for a read. Taking PCIe latency from literature as a guideline, a transfer takes a minimum of 552 ns [39, p. 7]. Assuming an identical library and FPGA management overhead for reads and writes leads to a theoretical minimum write latency of around 544 ns.

$$820 \ \texttt{ns} - \frac{552 \ \texttt{ns}}{2} = 544 \ \texttt{ns}$$

It is however important to keep in mind that this number is a rough lower bound at best.

Buffer objects operate under a different model. Once a buffer is created through the XRT API, two objects interact. The buffer object in the host, and the buffer on the card. The `sync` function transfers the buffer with the help of the xocl driver. To evaluate the latency and achieved bandwidth, buffers of each size will be written to and from memory 1000 times. The upper limit of buffer objects is currently 256 MiB since the 8 GiB of HBM memory on the card are fragmented into 32 equally sized partitions.

Interestingly, the first synchronization of buffer objects takes an additional 2 to 10 microseconds. Since this holds true even with larger buffers, the first data point of each run is removed to better reflect continuous performance.

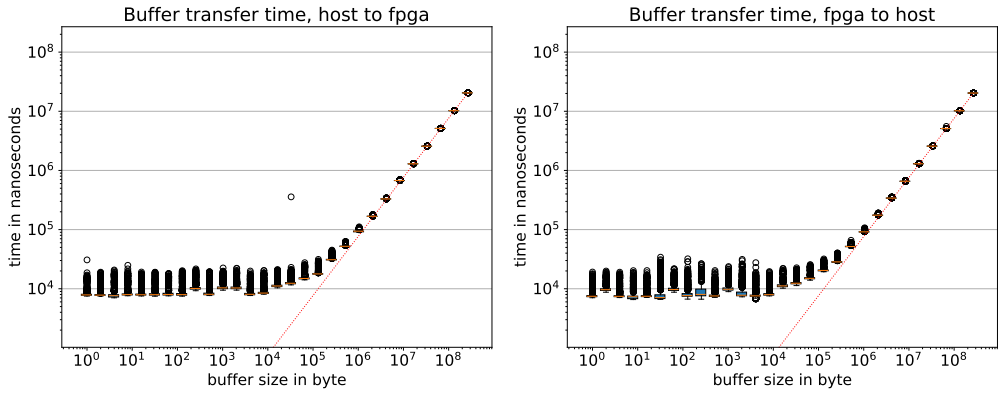

Figure 5.3: Transfer time for buffer syncing, left: host to fpga, right: fpga to host. Red line extrapolates linearly from 256MiB sized buffer to 0.

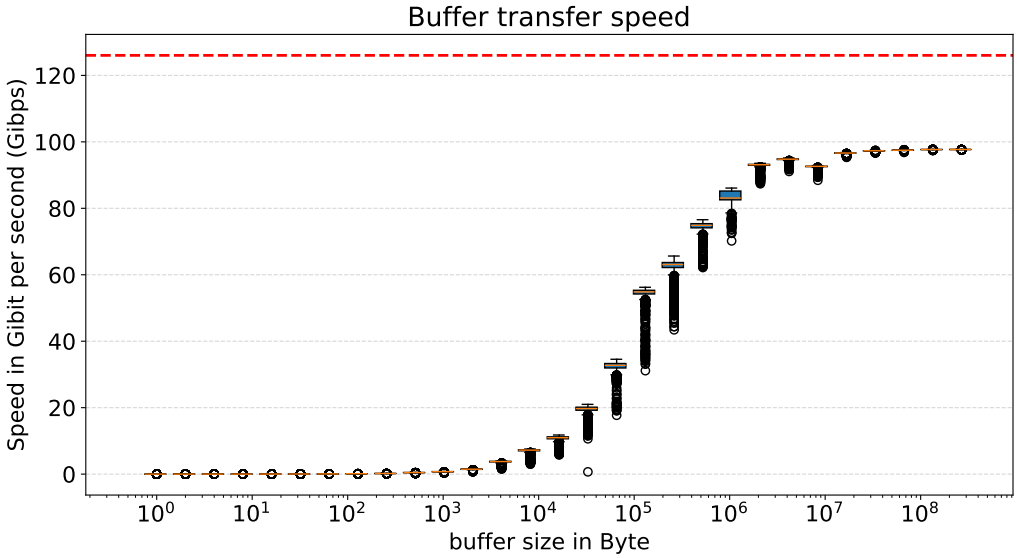

Figure 5.4: Achieved bandwidth during buffer syncing, shown: host to card. Red line indicates maximum PCIe Gen3 x16 goodput bandwidth.

In figure 5.3, the transmission latency for the buffer objects is shown in a box plot. This shows that there are a number of outliers, implying that this latency is not very stable. Latency has an overall upwards trend with increasing buffer sizes picking up at a buffer size of 1024 bytes. Up to this point, the median sync time is fairly constant between 10 and 30 μs, with outliers from 1-byte buffers surpassing the median of 512 byte buffers. When looking at the resulting bandwidth for this in figure 5.4, the effects can be seen more clearly. The achieved bandwidth is extremely low for buffers below 1024 bytes, rises asymptotically, resembling an arc-tangent, up until a buffer size of 1 MiB, from which bandwidth plateaus, up to the maximum buffer size of 256 MiB. This final range of buffers achieves speeds of just under 100 Gibit per second. As an absolute limit, the graph shows the

maximum achievable goodput via the PCIe gen3 x16 connection of the card.

These results show that XRT buffers are sufficient to communicate with services on the FPGA at speeds close to the network line-rate. It certainly reaches the maximum goodput possible with UDP over IPv4 without Jumbo packets. This allows us to partially answer ***RQ1***: Interactions with the host machine are possible at speeds capable of saturating line rate. However, since buffers of at least 1 MiB are required to reach such data rates, the latency for these transactions can be as high as 100 μs. This latency can be addressed by a simple ring-buffer.

Finally, XRT kernel executions are benchmarked in a similar fashion. A `mm2s` kernel takes a buffer object and writes it out on its AXI-Stream interface. By adding a second, free-running kernel which simply consumes the stream, this combination forms a predictable duo. The `mm2s` kernel will run for exactly 16 cycles, 52.8 ns, when supplied with a 1024 byte buffer, since it transfers 64 byte on every cylce over its 512 bit AXI-Stream interface. The time from kernel invocation to notification by the XRT api can be benchmarked. The results of this can be seen in figure 5.5.

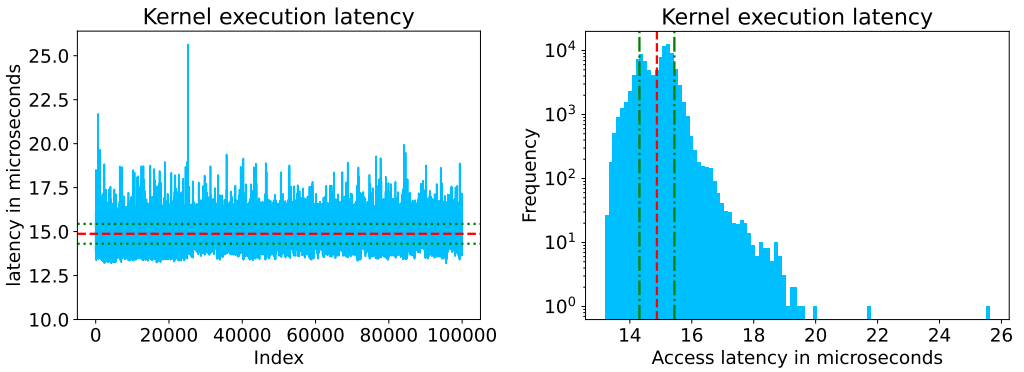

Figure 5.5: Kernel execution overhead latency, runtime of 53 ns subtracted. Mean shown in red, $\pm\sigma$ in green.

The results show a mean value of 14.9 μs, with a standard deviation of 0.59 μs. As can be seen by the outliers in both the plot and histogram, the distribution is heavy-tailed. The outliers appear to only sporadically cross 20μs, with only 2 instances surpassing it during 100,000 executions. There appears to be some overhead associated with kernel execution. According to the XRT API specification, a kernel execution should involve only a small number of reads and writes. Compared to the actual execution time of short-run kernels, the overall execution time is dominated almost completely by the XRT API overhead.

This leads to the following implication regarding ***RQ1***: AXI4-Lite accesses are the most timely communication options, allowing for fast signaling. For higher bandwidth applications, buffer objects provide sufficient bandwidth to saturate

the 100G network bandwidth. Kernel execution is inferior as a communication device from host to FPGA and back, both when compared to simple AXI4-Lite interactions and buffer object transfers.

5.1.2 Network

Testing the network proved to be a bit more tricky. To evaluate the latency of the network, a loopback and an active FPGA image is employed. The active image sends and receives packets to and from the loopback device. They each have their own non-synchronised clock which makes evaluation of the timestamps a bit more difficult. Figure 5.6 shows how and where timestamp insertion modules are distributed. The time generator block is omitted for brevity.

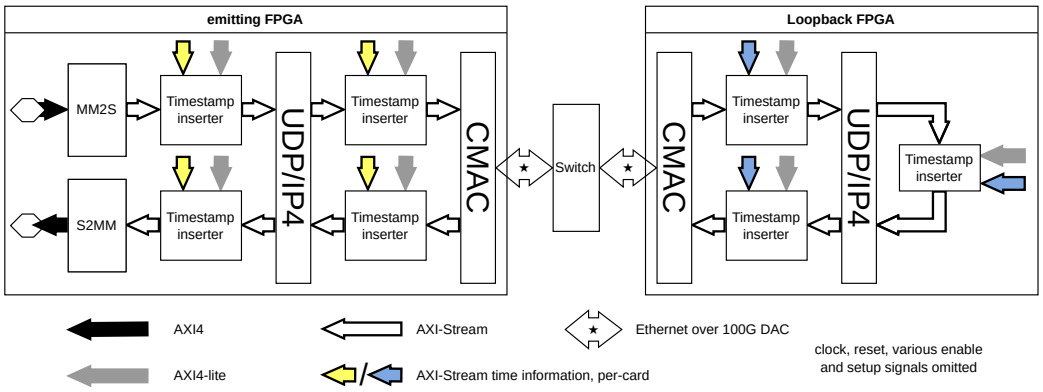

Figure 5.6: Layout of network latency test setup.

There is another aspect that needs to be considered for the timestamp modules between the CMAC and the UDP/IPv4 layer and vice versa. These AXI-Stream interfaces transfer entire Ethernet packets. Since this includes the media access control (MAC) address, IPv4 address, and the UDP port information, a change of these values will lead to the packet being dropped on the receiving end. In order to compensate for the 3 headers, the insertion offset must be incremented by 42. This breaks down to 14 bytes for the Ethernet header, 20 bytes for the IPv4 header and 8 bytes for the UDP header. Ethernets interframe gap, preamble and start frame delimiter (SFD) are removed by the CMAC, and are not accounted for in the offset.

The FPGA is running the design at 300Mhz. Time will be given primarily in cylces since this more accurately reflects sources of delay outside actual transmission time. At 300Mhz, a clock cycles has a period of $3.\overline{3}$ns.

The following evaluation takes a total of 50 million packets. The most important metric for an application using the current networking stack is the total

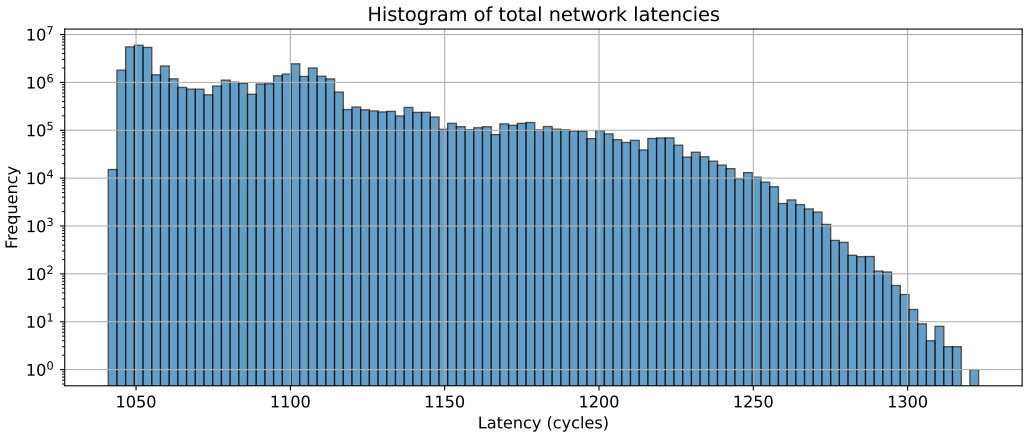

Figure 5.7: Total roundtrip time from insertion at MM2S kernel to reception at S2MM kernel.

network roundtrip time. Figure 5.7 depicts that roundtrip time, defined as the latency between packet emission and reception. The first thing to notice is the lower bound of 1041 cycles, which equates to a RTT of 3.47 μs. The highest recorded latency at 1323 cycles equates to a RTT of 4.41 μs. While such a transmission latency is low, the fact that no significant outliers are recorded is very good. This is in stark contrast to even AXI4-Lite accesses from the host, which while faster on average, do experience a much higher variance.

In order to understand the latency introduced by the network transmission itself, we must first examine the loopback latency, seen on the left side of figure 5.8. While this could be split further into ingress and egress latency, this makes little sense. Any packet will experience ingress and egress latency along the way. When looking at the histogram of the loopback time, it takes between 80 and 88 cycles for a packet to be routed back to the CMAC through the UDP/IPv4 layer. Interestingly, it seems like the distribution is made up of a bulk between 80 and 84 cycles and a small fraction which takes up to 4 more cycles. Such behavior is likely a result of different states within the UDP/IPv4 layer.

By subtracting the loopback delay, which is calculated per-packet from the RTT from the sending CMAC stream to the receiving CMAC stream of the sending FPGA, the network induced latency can be estimated. For simplicity, we will assume a bounded clock drift. The resulting network delay can be seen on the right in figure 5.8. Dividing the measured network RTT by two implies a one-way-latency of between 416 and 423 cycles, translating to a latency of between 1.387 and 1.410 μs. This should be seen only as a very rough guideline. In practice, it is extremely hard to measure the one-way latency, even with high precision clocks.

Overall, the FPGA achieves RTTs between 3.47 and 4.41 μs, though process-

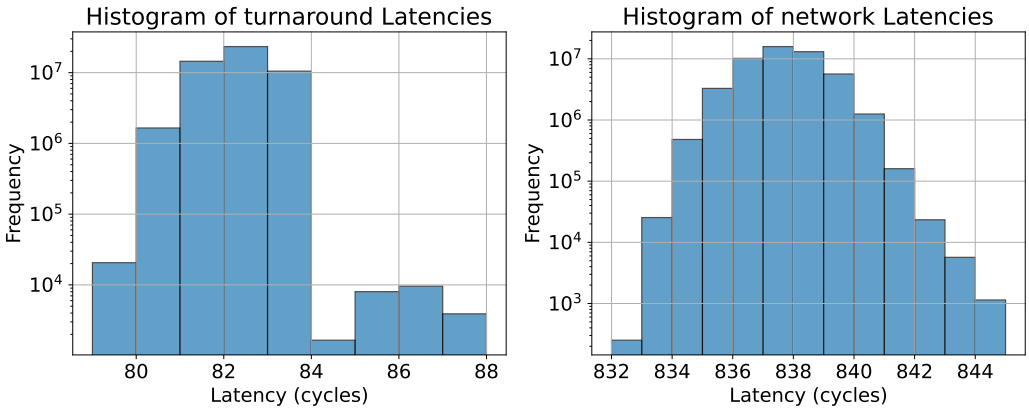

Figure 5.8: Histogram showing the loopback time and total delay introduced by the network

ing induced RTT still makes up 40% in the worst case. By designing against the CMAC, the application RTT could be as low as 832 cycles or 2.773 μs. The biggest source of jitter seems to be the UDP/IPv4 network layer.

5.1.3 Discussion

The preceding section allows ***RQ1***: "What characterizes the latency of interactions and transmissions from and to FPGA based SmartNICs, especially w.r.t. stability?" to be answered. Applications on the FPGA can reach RTTs consistently below 4.5 μs, with some as low as 3.47 μs. Interacting with the CMAC directly brings this range down to between 2.77 and 2.82 μs. Depending on the switch, even lower RTTs may be possible. The AXI4-Lite read/write primitives provided by XRT can reach latencies as low as 809 ns, but are subject to long tail latencies, most likely induced by the host itself. Further, they are limited by the 32 bit width of the bus, which limits their viability to low-bandwidth applications. To saturate the 100G network, XRT buffers are a good solution though the buffers should be larger than 1 MiByte, as smaller buffers quickly degrade the data rate of the transfers.

5.2 CKC - Consensus

The current implementation requires 1629 CLBs and 8 block random access memory (RAM) tiles on the FPGA. This equates to 1.50% and 0.56% of the Alveo U50s capacity. The complete system takes up 19.0% of CLBs ans well as 18.0% of block RAM, mainly due to PCIe communication as well as the UDP/IPv4 and CMAC modules. Clearly, there is still headroom for further logic, as well as accel-

erators from the same product family with more available resources. Furthermore, the CKC implementation satisfies all timing requirements to run at 300 Mhz, and expected power sits at 20.0 W. The XRT utilities report a power use of 18.8 W during stable operation. Again, the FPGA is clocked at 300Mhz, resulting in a clock cycle of $3.\overline{3}$ns.

To answer ***RQ3***, three aspects must be considered: processing latency, minimum stable round period, and finally throughput. The processing latency gives an absolute lower bound for round periods and will be covered first in 5.2.1. Since this does not factor in the timing intricacies of actual implementation, the lowest stable round period is determined experimentally in 5.2.2. To this end, subsection 5.2.3 will cover the performance and implications, given the lower bounds for round period λ. Finally, subsection 5.2.4 will summarize the findings.

5.2.1 Processing Latency

Processing latency of the CKC implementation is best evaluated using simulation. As explained in 2.3, the simulators are cycle accurate. Vivado includes a simulator, waveform viewer and VIP to use AXI4-Lite as a control interface. This makes it possible to simulate almost the entire design. The network stack, the mm2s, and the s2mm kernels are abstracted and the AXI-Stream interfaces are driven by the simulation. Due to a shortcoming in the Xilinx VIP, these are not appropriate for this evaluation. They struggle to maintain a high throughput level, reaching around 12% utilization, a transfer on every 8th cycle. Since both the `mm2s` and `s2mm` kernels, as well as the network layer can, and do stress the AXI-Stream more, they will be emulated directly. AXI4-Lite is a bigger hurdle. Since the control signals accessed by the AXI4-Lite control slave do not require such timing, the appropriate AXI4-Lite VIP can be employed.

There are essentially 3 events which define the overall processing latency of the CKC implementation. A round change as part of a consensus instance and a round change which results in a consensus instance change, as well as processing time for a single packet.

A round transition is a very straightforward event as it involves only 4 cycles until the listen state is reached once again. This can be seen in figure 5.9. Only the round_advance, round_begin and broadcast states need to be inhabited. Of course, the actual transmission of the packets will be determined by the broadcaster module of the framework.

Figure 5.10 depicts the transition of a consensus instance. The CKC implementation takes 11 cycles, 33.33 ns at 300Mhz, to transition from its Listen state to the Listen state of the next round. This is dominated by the DECIDE state which needs 5 cycles to push out the 3 values, as well as validity information out through the decision stream. It is also the only state which is inhibited for more

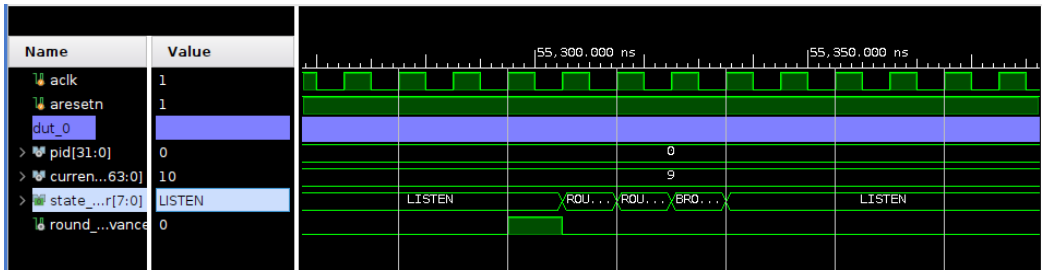

Figure 5.9: Waveform depicting round change, screenshot of Vivado.

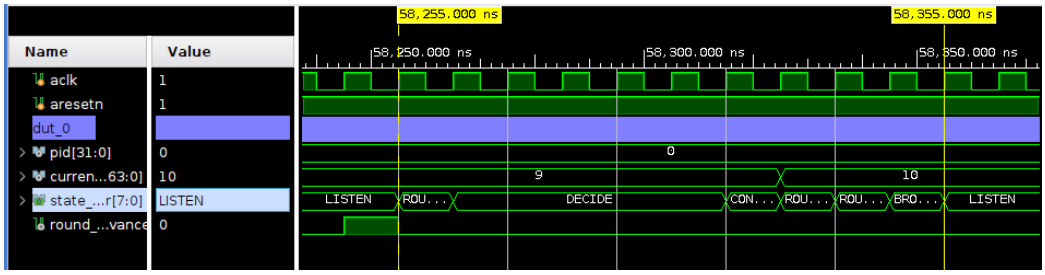

Figure 5.10: Waveform depicting consensus instance transition, screenshot of Vivado.

than 1 cycle. This is, of course assuming no backpressure on the decision stream.

The processing time for a single packet is exactly 2 cycles, since the packet is read, and the following cycle is required to acknowledge the transfer. When taking into account the broadcaster module as well as the the parser, the lowest stable period for consensus is 40 cycles, resulting in a theoretical maximum round frequency of 7.5 Mhz. This assumes a 0 latency connection between all nodes through a single AXI-Stream-switch and will most likely not be reached in hardware. The actual round period needs to be chosen such that it is larger than the total network RTT of the environment. This is further explored in the following chapter.

5.2.2 Stability

The implementation was evaluated by continuously running a 3-node configuration for 5 minutes, and repeating this setup 10 times for each round period. To record potential dropouts, the required quorum for CKC is set at 3. For periods of 960 cycles, 3.2μs and higher, no issues where recorded over all runs. To confirm this lower bound another run was conducted, showing stable operation over a 10 hour interval. This equates to a round period of $3.\overline{3}\mu$s and a round frequency of 312.5 Khz. Consensus is reached at half that, resulting in a decision frequency of 156.3 Khz.

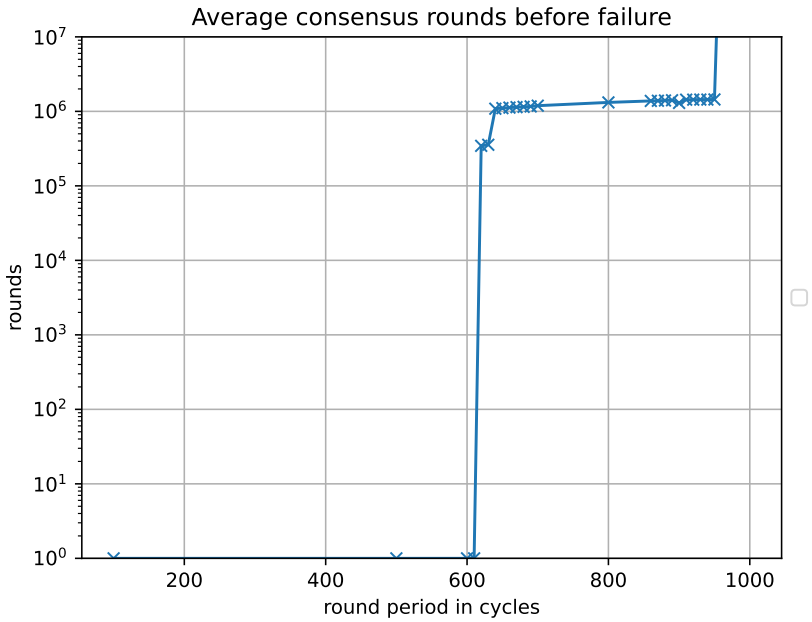

Figure 5.11: Consensus rounds before failure, average over 10 runs per round period λ.

When targeting round periods of 610 cycles, $2.\overline{3}\mu$s, and below, the cluster fails before deciding once. This is very close to the expected behavior since for round periods near half the RTT, messages from round 0 do not have sufficient time to reach the other nodes. As such, all nodes advance to the next round, notice that they have not received any packet and quit operation while indicating that no qourum can be reached.

For periods of 620 to 950 cycles, 2.07μs and 3.17μs respectively, the behavior is quite unexpected. CKC drops out of normal operation after between 1.45 and 1.03 million consensus instances due to message loss. This equates to an average runtime of 9.2 and 2.1 s before package loss occurs. Almost all experiments terminate within this range, shown in figure 5.11, indicating a causal relationship. The exception are runs with periods of 630 and below, where some immediately fail due to packets arriving late, as explained above. This result is perplexing since such behavior was not recorded during the network evaluation, where throughput spiked even higher. Due to time constraints, the investigation into this did not provide an explanation for this behavior. The employed Tofino switch does record reception errors and package drops for some, but not all of the experiment runs. In one case with $\lambda <= 950$ cycles, the switch inexplicably dropped a bootstrapper packet, preventing cluster operation. As the Tofino line has not been developed further since 2023 and has been discontinued recently, support and documentation on the switch configuration is somewhat sparse.

This leaves a period of 960 cycles, 3.2μs, as the lowest bound for stable operation for CKC. This bound was confirmed over the span of 10 hours and results in a round frequency of 312.5 Khz, as well as a consensus frequency of 156.25 Khz under stable operation. Due to the behavior of the system for periods of between X and 950 cycles indicating a systematic problem, lower periods and by associated round frequencies may be achievable if this problem is addressed. A different network setup resulting in a lower RTT would equally improve the lower bound for round periods which achieve stable operation.

5.2.3 Throughput

Throughput of the CKC is directly linked to the round period λ and the associated round frequency F_r. During stable operation, i.e. no failures, the frequency of decisions reached by CKC is simply half of F_r. For 3-node operation, the rate of packet emission is $F_{\text{emission}} = F_r * 2$ since every round involves transmitting a message to the other 2 nodes. Package size is currently fixed at a 256 byte UDP payload, equating to to physical transfer of $256 + 8 + 20 + 18 + 20 = 322$ bytes. The required data rate dr is therefor given by the following equation:

$$dr = 322 \text{ byte} * \frac{8 \text{ bit}}{\text{byte}} * 2 * F_r$$

Taking the lowest stable round period of 960 cycles, 3.2μs and $F_r = 312.5$ Khz, directly translates to these performance figures: Decision frequency sits at 165.6 Khz with a decision data rate of 240 Megabit/s, and the network data rate at 1.61 Gigabit/s. This results in a network overhead of 6.7 when comparing network and decision data rate.

Figure 5.13 shows the network overhead for different value widths. Increasing the width of consensus values to 448 bytes each is feasible and would affect decision- as well as network data rate. With 448 byte values, the messages would contain a UDP payload of 1408 bytes, 1474 at the wire. Any further increases would necessitate increased parsing complexity as the values would not be transmitted AXI-Stream-transer-aligned. This would result in a decision data rate of 1.68 Gigabit/s and a network data rate of 7.37 Gigabit/s, as shown in 5.12. The network overhead in this case drops to 4.4, closer to the theoretical limit. There are higher MTU, a common one being 9000 bytes for Ethernet, though support for this standard is not included in all network equipment. At such large packet sizes, the actual transmission time needs to be factored into the latency and stability considerations. Just transferring a 9000 byte packet over an 512-bit AXI-Stream takes 141 cycles, or 470 ns. Overall, increasing the byte width of individual values does impact timing since the FPGA and the network require longer to transfer and process these values.

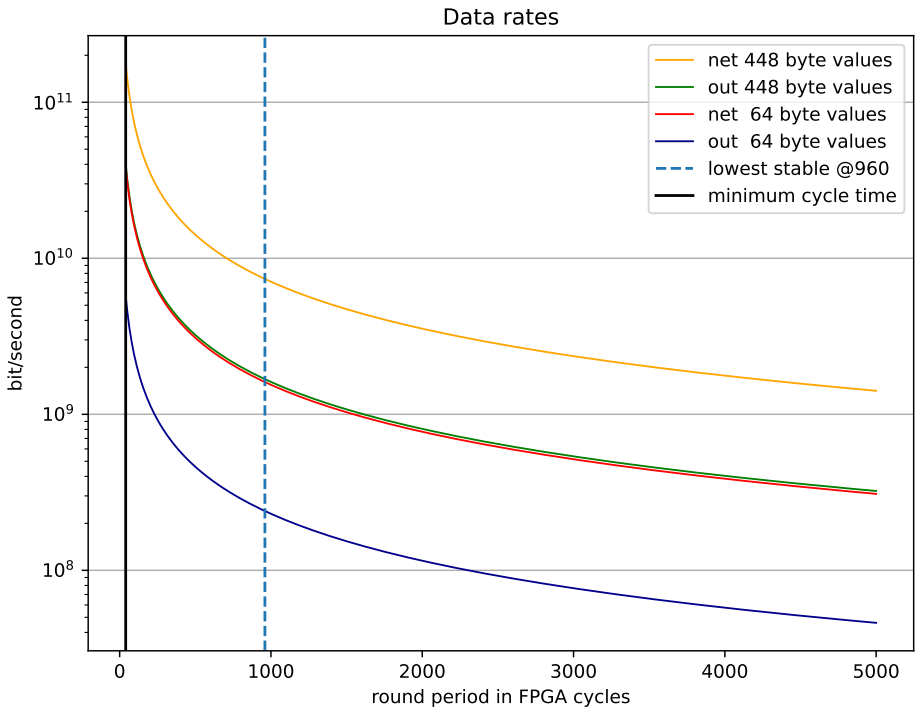

Figure 5.12: Data rates of CKC both network and decision output, given different value widths.

The CKC implementation can be replicated on the FPGA up to 128 times, at which point it reaches the socket limit of the UDP/IPv4 network layer. To achieve link saturation with the enhancement of 448 byte wide values, 13 replicas are required to reach a network data rate of 95.81 Gigabit/s. The decision data rate totals 21.84 Gigabit/s, with the network overhead remaining at 4.39. If maintaining 64 byte values is preferred, 62 replicas are required to reach a network data rate of 99.82 Gigabit/s. Since the network overhead remains constant with replication, the decision data rate only grows to 14.88 Gigabit/s in this case. The concept of decision frequency cannot be fully extended to replicated consensus instances as these decide, and very importantly fail, separately. Replicating CKC instances has other consequences, such as loss of ordering between instances and the need to handle failures of single or multiple instances at once.

5.2.4 Discussion

To address ***RQ3***, we will compare the performance of CKCs implementation with the performance of Consensus in a Box [22] and Waverunner [1]. Since it was not feasible to recreate their experiments on our setup, we must first examine their respective evaluation environment. Next, we will compare the decision frequency,

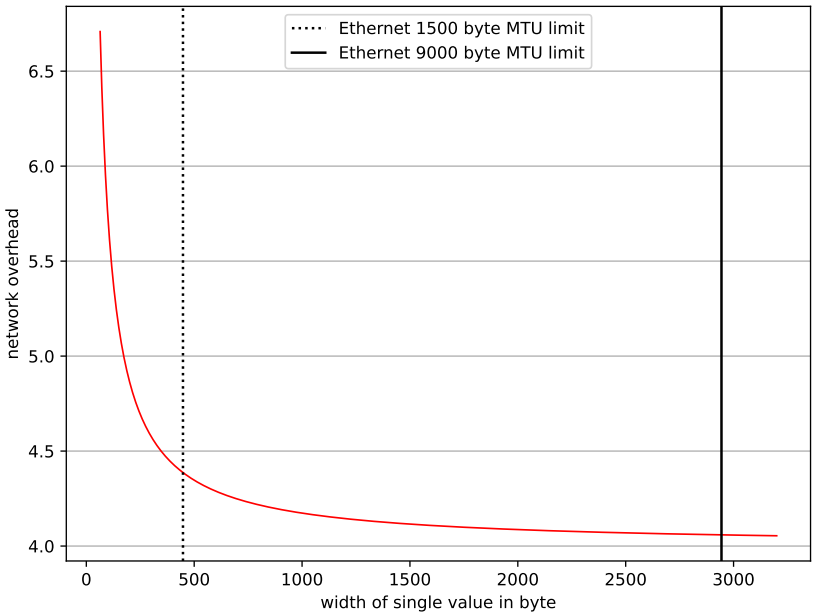

Figure 5.13: Network overhead when comparing network data rate to decision data rate, dependent on width of singular values.

after that the decision data rate and finally, the network utilization. After that, we will reason about the differences in results.

Consensus in a Box uses 10 Gigabit/s links with the Xilinx VC709 board [22]. While this constitutes a lower individual link speed, the board features four links as opposed to one on the Alveo U50. Interestingly, the RTT of their setup is lower than in our setup, reaching 1.4μs at around 512 byte message length. Two connection topologies are evaluated, one being point-to-point links between the FPGAs, the other being a traditional star with a switch. Waverunner emloys the Alveo U280, equipped with two 100 Gigabit/s links, connected through a switch [1]. Network RTT is only given for minimum sized packets, and sits at 1.68μs, again lower than our setup.

Consensus in a Box is able to reach a peak of around 4 Million writes per second over direct links with a decision size of 32 bytes. When expanding this to to 256 Bytes, the number of writes, or decisions, drops to 1.4 Million writes per second, reaching a decision data rate of around 2.9 Gigabit/s. At this point, Conensus in a Box is bandwidth limited. Yet, Consensus in a Box outperforms our CKC implementation, likely because of the pipelined nature of zookeepers atomic broadcast.

Waverunner reaches an even higher frequency of consensus, handling up to 26 Million decisions per second with a value size of 50 bytes. At that rate, Waverunner reaches a decision data rate of 10.4 Gigabit/s and a reported network

data rate of 100 Gigabit/s. The decision frequency for values is not given, but a linear approximation can give a lower bound. Extrapolating to 256 byte values, Waverunner should reach at least 7 Million decisions per second. This should also increase the decision data rate since the protocol overhead reduces with increasing value size.

Overall, both networks experience an around 50% to 66% lower RTT than our setup. Since the CKC algorithm in a non-pipelined form is directly limited by the RTT[1], performance of CKC suffers from the setup itself. At half the current stable sending period λ, the CKC implementaton with 448 byte values would achieve 0.63 million decisions per second, reaching 3.36 Gigabit/s throughput and 14.7 Gigabit/s network utilization. At this point it is clear that pipelining or replication of CKC is required to reach the performance of related works.

The answer to ***RQ3*** is a bit more nuanced than a definitive yes or no. With the changes proposed near the end of section 5.2.3, CKC have the potential to outperform both Consensus in a Box and Waverunner in decision data rate, though Consensus in a Box employs a slower link. With all improvements, i.e. larger values and pipelining, implemented, CKC can reach a theoretical decision data rate of around 21.8 Gigabit/s, as opposed to 2.9 Gigabit/s for Consensus in a Box and 10.4 Gigabit/s for Waverunner. However, the current implementation still suffers from the small value width, higher RTT and by extension round period λ and thus falls short of reaching such performance.

[1] technically the expected one-way latency which is $\frac{\text{RTT}}{2}$

Chapter 6

Conclusion

This thesis centered around the use of FPGAs as network accelerators in a DC setting. While this technology offers benefits, the increased implementation effort still proves a barrier to entry. In the previous chapters, the FPGA was evaluated on the latency characteristics of its communication. Abstractions were designed to more efficiently implement synchronous distributed algorithms on that FPGA. Finally, an existing algorithm was implemented and evaluated for viability against existing implementation works.

Concerning ***RQ1***: "What characterizes the latency of interactions and transmissions from and to FPGA based SmartNICs, especially w.r.t. stability?". Network latency between FPGAs is stable, with jitter only in the range of 46 ns, but still sits at $2.\overline{7}$ μs. The employed network stack does add significantly on top of this latency, up to 1.5 μs. Interacting with the host is not a bottleneck in terms of data rate, but introduces latency, which is dependent on the transfer buffer sizes.Since the communication with user applications still relies on kernel drivers and system calls, the host-FPGA communication is subject to the same latency spikes as traditional networking.

RQ2: "How can synchronous Distributed Systems primitives be more easily implemented on FPGA based SmartNICs?". The exploration of ***RQ2*** in chapter 3.2 results in the layout in use by the CKC implementation as seen in chapter 4.4. It presents an effective tool for the implementation of synchronous distributed algorithms. By transforming algorithms into their control graph, structured implementation on the FPGA is enabled. The modules allow for easy output dispatching and input ingestion, with communication being explicitly structured and the ability to saturate the network.

As for ***RQ3***: "Can our synchronous consensus implementation outperform state of the art when leveraging FPGAs?" Currently, the CKC implementation does not reach the performance presented in similar works. This is not due to the implementation environment reaching design limits, but has another reason. The

sequential nature of the round-based synchronous algorithm cannot make progress quickly enough, since only one consensus instance can be outstanding. If it this drawback is eliminated, the system has the potential to outperform waverunner [1] and scaled[1] consensus in a box [22] in the decision data rate. To that end, the following section on future work serves as an overview of possible improvements and additions.

6.1 Future Work

While the abstraction provided by the Xilinx University Programm network example [30] provides a simple network abstraction, it limits FPGA applications. The ability to issue interrupts to the host and the use of XRT do impose performance penalties, as can be seen in 5.1. UDP sockets must be declared in advance, limiting the possibility of joining existing clusters. To tackle these drawbacks, porting the implementation of CKC in this paper to a different network stack is a viable option. Especially Corrundum [12] is a potential target for such work. Changing the network stack also opens the door to the use of software defined networking (SDN). This necessitates the use of more specialized network equipment, in particular SDN switches and routers.

In a similar vein, the transfer of data to and from the FPGA is done with XRT buffer objects. XRT also offers the use of direct memory access (DMA) primitives. This allows the kernels to directly access host memory. Since AXI4 must be used, this complicates the interaction somewhat. Using DMA also increases risk of race conditions and host memory corruption.

Since the current HDL code only implements the consensus logic of CKC, adding group membership functionality is a natural next step. This is best done in tandem with an expanded UDP/IPv4 layer, as explained above. Adding further abstractions on top of CKC is also a possible avenue. State machine replication (SMR) is a somewhat viable extension, since the Alveo U50 accelerator includes 8 GiB of HBM. The problems in this context are the volatile nature of HBM, as well as the small size when compared to host RAM.

The CKC algorithm is also limited by the fact that a round cannot be shorter than the longest RTT between all nodes. Pipelining or replicating the consensus instances could alleviate this problem. Such a change does increase the complexity of the design. It may be worth exploring a soft-CPU, which only executes in error states, to manage the complexity of fail-over handling. The work by Ibanez et al. [20] can help to optimize such a design.

[1] 10x to account for lower networking speed of the publication.

Bibliography

[1] Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho, Michael Ferdman, Peter Milder, and Shuai Mu. Waverunner: An elegant approach to hardware acceleration of state machine replication. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 357–374, 2023.

[2] AMD. *UltraScale+ Devices Integrated 100G Ethernet Subsystem LogiCORE IP Product Guide (PG203)*, 2024. `https://docs.amd.com/r/en-US/pg203-cmac-usplus/Introduction`. UltraScale+ Devices Integrated 100G Ethernet Subsystem LogiCORE IP Product Guide (PG203).

[3] AMD. *Versal™ Architecture and Product Data Sheet: Overview (DS950)*, 2024. `https://docs.amd.com/v/u/en-US/ds950-versal-overview`.

[4] AMD Xilinx. *AXI4-Stream Infrastructure IP Suite v3.0 LogiCORE IP Product Guide*, 2023. `https://docs.amd.com/r/en-US/pg085-axi4stream-infrastructure/AXI4-Stream-Infrastructure-IP-Suite-v3.0-LogiCORE-IP-Product-Guide`.

[5] AMD Xilinx. *AXI Verification IP LogiCORE IP Product Guide (PG267)*, 2024. `https://docs.amd.com/r/en-US/pg267-axi-vip`.

[6] AMD Xilinx. *XRT Documentation*, 2024. `https://xilinx.github.io/XRT/master/html/index.html`.

[7] Arm Limited. *AMBA® AXI™ and ACE™ Protocol Specification*, 2013. `https://documentation-service.arm.com/static/5f915b62f86e16515cdc3b1c`.

[8] Arm Limited. *AMBA® AXI4-Stream Protocol Specification*, 2021. https://documentation-service.arm.com/static/60d5e2510320e92fa40b4788.

[9] Luís Rodrigues Christian Cachin, Rachid Guerraoui. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011.

[10] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[11] Chunqiang Tang Facebook, Laine Campbell. How meta built the infrastructure for threads. https://engineering.fb.com/2023/12/19/core-infra/how-meta-built-the-infrastructure-for-threads/.

[12] Alex Forencich, Alex C Snoeren, George Porter, and George Papen. Corundum: An open-source 100-gbps nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 38–46. IEEE, 2020.

[13] Petr Gebauer. High-performance byzantine fault tolerant consensus. Master's thesis, ETH Zurich, 2023.

[14] Dan Gisselquist. Building an axi-lite slave the easy way. https://zipcpu.com/blog/2020/03/08/easyaxil.html.

[15] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. {uKharon}: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 101–120, 2022.

[16] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In *International conference on reliable software technologies*, pages 38–57. Springer, 1996.

[17] David Harris and Sarah Harris. *Digital Design and Computer Architecture, Second Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2012.

[18] Zhenhao He, Dario Korolija, Yu Zhu, Benjamin Ramhorst, Tristan Laan, Lucian Petrica, Michaela Blott, and Gustavo Alonso. {ACCL+}: an {FPGA-Based} collective engine for distributed applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 211–231, 2024.

[19] Brad Hedlund. Top of rack vs end of row data center designs, 2009. https://bradhedlund.com/2009/04/05/top-of-rack-vs-end-of-row-data-center-designs/.

[20] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 239–256, 2021.

[21] Intel / Altera. *Avalon® Interface Specifications*, 2022. https://www.intel.com/content/www/us/en/docs/programmable/683091/22-3/introduction-to-the-interface-specifications.html.

[22] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 425–438, 2016.

[23] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[24] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[25] André Medeiros. Zookeeper's atomic broadcast protocol: Theory and practice. *Aalto University School of Science*, 20, 2012.

[26] Microsoft. Microsoft and openai extend partnership. https://blogs.microsoft.com/blog/2023/01/23/microsoftandopenaiextendpartnership/.

[27] Stevan Vlaovic Ruslan Meshenberg Netflix, Yury Izrailevsky. Completing the netflix cloud migration. https://about.netflix.com/en/news/completing-the-netflix-cloud-migration.

[28] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[29] Philippe Raïpin Parvédy and Michel Raynal. Optimal early stopping uniform consensus in synchronous systems with process omission failures. In

Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '04, page 302–310, New York, NY, USA, 2004. Association for Computing Machinery. https://doi.org/10.1145/1007912.1007963.

[30] Xilinx University Program. Xup vitis network example (vnx). https://github.com/Xilinx/xup_vitis_network_example/tree/master.

[31] Davide Rovelli. Synchronous consensus with omission failures for optimal high-throughput services. To be released.

[32] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. Limago: an FPGA-based Open-source 100 GbE TCP/IP Stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 286–292. IEEE, Sep 2019.

[33] Timo Schneider, Pengcheng Xu, and Torsten Hoefler. Fpspin: An fpga-based open-hardware research platform for processing in the network. *arXiv preprint arXiv:2405.16378*, 2024.

[34] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43. IEEE, 2015.

[35] Gustavo Sutter, Mario Ruiz, Sergio López-Buedo, and Gustavo Alonso. FPGA-based TCP/IP Checksum Offloading Engine for 100 Gbps Networks. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, Dec 2018.

[36] Switzerland Systems Group of ETH Zürich and Spain HPCN Group of UAM. 100g-fpga-network-stack-core github page on the udp branch. https://github.com/hpcn-uam/100G-fpga-network-stack-core/tree/872d53d706c04dad50fc50faeb1bb96256060ffe.

[37] Pasindu Tennage, Antoine Desjardins, and Lefteris Kokoris-Kogias. Racs and sadl: Towards robust smr in the wide-area network. *arXiv preprint arXiv:2404.04183*, 2024.

[38] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path {SmartNIC} for accelerating distributed systems. In

17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pages 987–1004, 2023.

[39] Noa Zilberman, Matthew Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W Moore. Where has my time gone? In *Passive and Active Measurement: 18th International Conference, PAM 2017, Sydney, NSW, Australia, March 30-31, 2017, Proceedings 18*, pages 201–214. Springer, 2017.

Glossary

AMBA Advanced Microcontroller Bus Architecture 11

API application programming interface 42–44, 46

ARP address resolution protocol 31

AXI Advanced eXtensible Interface 11, 17

AXI-Stream AXI-Stream 1, 11, 12, 14, 15, 18, 20, 25, 27, 28, 30–39, 46, 47, 50, 51, 53

AXI4 AXI4 full 11, 15, 18, 58

AXI4-Lite AXI4-Lite 1, 11, 15, 17, 18, 28, 30, 32, 33, 35, 37, 38, 42, 44, 46–50

CKC cool kids club 2, 24–26, 29, 37–39, 41, 42, 50–58, 70, 71

CLB configurable logic block 8, 9, 49

CMAC 100G MAC 1, 27–30, 47–49

CPU central processing unit 4, 10, 17, 19, 58

CS+O Crash Stop with Omission 23

DC datacenter v, 3, 4, 57

DMA direct memory access 58

DPU data processing unit 19

DS distributed system 1, 3, 4, 7, 8, 20, 21, 28

ERT Embedded runtime 18

FPGA field programmable gate array v, 1, 2, 4, 7–11, 13, 15, 17–21, 25, 29, 31, 37, 41, 42, 44, 46–50, 53–55, 57, 58

HBM high bandwidth memory 41, 44, 58

HDL hardware description language 1, 9, 10, 17, 58

HLS high level synthesis 17

IC integrated circuit 8

IP intellectual property 2, 10, 32, 34, 36–38, 67

IPv4 internet protocol v4 1, 19, 23, 26, 27, 29–31, 38, 46–49, 54, 58

LUT lookup table 8, 10

MAC media access control 28, 30, 47, 65

MPI message passing interface 19

Mpps mega packets per second 21

MTU maximum transmission unit 31, 39, 53

NIC network interface card 19, 20

OS operating system 4, 44

PCIe PCI Express v, 4, 9, 17, 18, 44–46, 49

PR-Alg Parvédy & Raynal Algorithm 23, 24, 26

PTP precision time protocol 27, 28

RAM random access memory 49, 58

RDMA remote direct memory access 19, 21

RTL register transfer level 1, 9, 17

RTT round trip time 28, 48, 49, 51–53, 55, 56, 58

SDN software defined networking 58

SFD start frame delimiter 47

SmartNIC smart network interface card 4, 19, 20

SMR state machine replication 58

SoC system on chip 19

TCP transmission control protocol 26

TOR-switch top-of-rack-switch 3, 28

UDP user datagram protocol 1, 19, 23, 26, 27, 29–31, 33, 38, 39, 46–49, 53, 54, 58

VIP verification IP 10, 32, 33, 37, 38, 50

VLAN virtual local area network 41

XRT Xilinx Runtime 1, 17–19, 29, 32, 37, 41–44, 46, 49, 50, 58

Appendix

Algorithm 2: CKC uniform consensus block. Executes for every process p_i. To be published by Davide Rovelli, SWYSTEMS USI

1 $V_i \leftarrow \emptyset$, $liveset_i \leftarrow \Pi$, $prevLiveset_i \leftarrow \Pi$, $locked_i \leftarrow \emptyset$, $r_i = 0$
2 $c_i = 0$ `// current consensus instance`
3 $lastDecided = \perp$
4 $MM_i \leftarrow \emptyset$
5 **to** PROPOSE$(c_{next},\ M_i,\ M_{next},\ v)$:
6 **if** $c_i < c_{next}$ **then**
7 $c_i = c_{next}$
8 $V_i \leftarrow V_i \cup v$
9 $liveset \leftarrow M_i$
10 $MM_i \leftarrow MM_i \cup M_{next}$
11 START-ROUNDS(0)
12 **upon** ROUND-NEXT(r):
13 $r_i = r$
14 **if** $|liveset| < Q$ **then**
15 DECIDE$(\perp,\ \perp,\ \perp)$ `//` p_i `is faulty, decide empty set = quit`
16 $prevLiveset \leftarrow liveset$
17 MULTICAST$(\ c_i,\ r_i,\ V_i,\ locked_i,\ MM_i,\ lastDecided_i)$ to $prevLiveset$
18 $liveset \leftarrow \emptyset$
19 **upon** RECV$(c_j,\ r_j,\ V_j,\ locked_j,\ MM_j,\ lastDecided_j)$ from $p_j \mid p_j \in prevLiveset$:
20 **if** $c_i = c_j - 1$ **then**
21 `// we are behind, decide, reset state and advance`
22 DECIDE$(lastDecided_j)$
23 $lastDecided \leftarrow \{V_i,\ MM_i,\ prevLiveset\}$
24 $c_i = c_j, r_i = r_j, V_i \leftarrow \emptyset, MM_i \leftarrow \emptyset, locked_i \leftarrow \emptyset$
25 **if** $c_i = c_j + 1$ **then**
26 `// we are ahead, ignore payload just update liveset`
27 $V_j \leftarrow \emptyset, MM_j \leftarrow \emptyset, locked_j \leftarrow \emptyset$
28 **if** $r_i > r_j$ **then**
29 `// do nothing (filter out async msgs)`
  else
30 **if** $r_i < r_j$ **then**
31 START-ROUNDS(r_j)
32 $liveset_i \leftarrow liveset_i \cup p_j$
33 $V_i \leftarrow V_i \cup V_j$
34 $MM_i \leftarrow MM_i \cup MM_j$
35 $locked_i \leftarrow locked_i \cup locked_j$
36 **if** $liveset_i = prevLiveset_i$ **or** $locked_j \neq \emptyset$ **then**
37 $locked_i \leftarrow locked_i \cup p_i$
38 **if** $|locked_i| \geq |Q|$ **then**
39 DECIDE$(V_i,\ MM_i,\ prevLiveset)$
40 $lastDecided \leftarrow \{V_i,\ MM_i,\ prevLiveset\}$

Algorithm 3: CKC uniform consensus algorithm adapted from algorithm 2. Executes for every process p_i.

```
1  V_i ← ∅, liveset_i ← Π, prevLiveset_i ← Π, locked_i ← ∅, r_i = 0
2  c_i = 0 // current consensus instance
3  to PROPOSE(c_next, v):
4      if c_i < c_next then
5          c_i = c_next
6          V_i ← V_i ∪ v
7          START-ROUNDS(0)
8  upon ROUND-NEXT(r):
9      r_i = r
10     if |liveset| < Q then
11         DECIDE(⊥, ⊥, ⊥) // p_i is faulty, decide empty set =
               quit
12     prevLiveset ← liveset
13     MULTICAST(c_i, r_i, V_i, locked_i) to prevLiveset
14     liveset ← ∅
15 upon RECV(c_j, r_j, V_j, locked_j) from p_j | p_j ∈ prevLiveset:
16     if c_i == c_j and r_i == r_j then
17         liveset_i ← liveset_i ∪ p_j
18         V_i ← V_i ∪ V_j
19         locked_i ← locked_i ∪ locked_j
20         if liveset_i = prevLiveset_i or locked_j ≠ ∅ then
21             locked_i ← locked_i ∪ p_i
22         if |locked_i| ≥ |Q| then
23             DECIDE(V_i)
```