Enabling Shared Memory Communication in Networks of MPSoCs

DOI:
10.1002/cpe.4774

Document Version
Accepted author manuscript

Link to publication record in Manchester Research Explorer

Citation for published version (APA):
Lant, J., Concatto, C., Attwood, A., Pascual Saiz, J., Ashworth, M., Navaridas, J., Luján, M., & Goodacre, A. (2018). Enabling Shared Memory Communication in Networks of MPSoCs. *Concurrency and Computation: Practice and Experience*. https://doi.org/10.1002/cpe.4774

Published in:
Concurrency and Computation: Practice and Experience

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

Download date:09. Aug. 2024

Received <day> <Month>, <year>; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

SPECIAL ISSUE PAPER

Enabling Shared Memory Communication in Networks of MPSoCs

Joshua Lant*, Caroline Concatto*, Andrew Attwood*, Jose A. Pascual*, Mike Ashworth*, Javier Navaridas*, Mikel Luján*, John Goodacre*

[1]School of Computer Science, University of Manchester, Manchester, UK

Correspondence
*Email: {joshua.lant, caroline.concatto, andrew.attwood, jose.pascual, mike.ashworth.compsci, javier.navaridas, mikel.lujan, john.goodacre}@manchester.ac.uk

Summary

Ongoing transistor scaling and the growing complexity of embedded system designs has led to the rise of MPSoCs (Multi-Processor System-on-Chip), combining multiple hard-core CPUs and accelerators (FPGA, GPU) on the same physical die. These devices are of great interest to the supercomputing community, who are increasingly reliant on heterogeneity to achieve power and performance goals in these closing stages of the race to exascale. In this paper we present a communication backend, designed to sit inside the FPGA fabric of a cutting-edge MPSoC device, enabling networks of these devices to communicate within a distributed and shared memory context, with reduced need for costly software networking system calls. We will present our implementation and discuss the main design decisions relevant to the use of the Xilinx Zynq Ultrascale+, a state-of-the-art MPSoC, and the challenges which arise owing to the constraints imposed by the device's natural limitations. We demonstrate the working prototype system connecting two MPSoCs, with communication between processor and remote memory region and accelerator. We then discuss the limitations of the current implementation and highlight areas of improvement to make this solution production-ready.

KEYWORDS:
MPSoC, FPGA, networks, interconnect, HPC, distributed shared memory

1 | INTRODUCTION AND MOTIVATION

Over the past decade the embedded systems landscape has changed dramatically due to the growing demands from the mobile market and the rise of the Internet of Things. These advances led to the SoC paradigm, with increasingly complex and heterogeneous systems placed upon the same physical die. At the same time the High Performance Computing (HPC) community has had to deal with the consequences of the breakdown of Dennard scaling [1], causing an explosion in the core count and power consumption of the largest machines to keep pace with the demands for increasingly greater computing capabilities.

These two phenomena have created an opportunity for convergence between the HPC and the Data Centre markets, and to the use of low-power, mobile processors, which are beginning to penetrate the server market [2], and are even being used by the RIKEN institute for the next stage of their roadmap to an Exascale class machine, the *post-K* computer [3]. It is unsurprising that this shift is happening, given the greatest challenge which computer and system architects now face in the race to exascale is the need for increasing energy efficiency. Naively scaling out current architectures, e.g. in the TOP500, would result in an exascale machine requiring in excess of 100MW power, which is unrealistic in terms of infrastructure

[0]This work was funded by the European Union's Horizon 2020 research and innovation programme under grant agreements No 671553 and 754337.

and cost. Designers are facing the challenge of reducing power consumption by a number of means: increased component density, tighter coupling between processor/memory/accelerator/network, shorter paths for copper lines, increased performance/Watt of components, hyper-converged storage etc. The relentless quest for increasingly more power-efficient systems opens up the potential use of new FPGA devices as suitable candidates for the main compute element of a system.

1.1 | FPGAs for HPC

The key advantage of FPGA technology is its impressive performance-per-watt compared to GPU, and flexibility compared to ASICs. As well as this, FPGAs can outperform GPUs for a number of problem domains, such as in neural networks [4], signal processing [5], finance [6], computer vision [7], data mining [8] to cite a few. There have traditionally been several key barriers towards the uptake of FPGAs for use in HPC and Data Centre applications. These barriers are now seemingly being broken down, as is evident from the interest shown in FPGA vendors from companies such as IBM, Intel and Microsoft [9, 10, 11]. We see the barriers as follows:

Cost- Traditionally, the cost of logic cells has prohibited their uptake in the HPC domain. However, in the ten years between 1999 and 2009 the price of FPGA per logic cell fell by an order of magnitude, and the compute performance per logic cell increased by two orders of magnitude [12]. Current FPGAs can now realize vastly more complex designs, and costly floating point operations can be more readily performed.

Programmability- As High-Level Synthesis (HLS) tools and languages mature, bespoke hardware acceleration is no longer the preserve of specialized hardware engineers. There are now many options which raise the level of abstraction above register transfer language (RTL), such as Vivado HLS [13], OpenCL, Bluespec System Verilog [14], FCUDA [15] among others.

Memory Bandwidth- Traditionally, FPGAs were used as a coprocessor to a CPU and connected through PCIe lanes. This was very restrictive, particularly given the limited on-chip memory of FPGAs. Tighter coupling between the FPGA and memory provides a far more suitable model for data-intensive HPC workloads. Similarly, tight coupling between compute and network will achieve much lower latency for small message , as those of HPC workloads such as in LQCD codes (lattice quantum chromodynamics) or certain machine learning applications. This tight coupling is featured in modern MPSoC devices, e.g. Xilinx Zynq, and indeed increases their suitability for a number of applications within the HPC domain. The remainder of this paper considers Zynq devices, which consist of hard-core ARM based processing system **(PS)**, with reprogrammable FPGA logic **(PL)** on the same die.

1.2 | Network Requirements for Clusters of FPGAs

There are two distinct forms of communication which need to be dealt with in order to build a suitable network for HPC applications on any given system, and some special considerations which are required for FPGA-based systems. We must first consider small, ultra-low latency packets in the form of barrier and synchronization messages which are common to MPI-like applications. The second form of communication is based around larger RDMA (Remote Direct Memory Access) based transfers, used to exchange data between nodes.

Typically the smaller synchronization messages will be used to tell applications that they need no longer block the running of an application, informing all other nodes involved in a job that the data they require for the next stage of computation is available. This sort of communication relies on low latency to avoid bottlenecks in running the application.

Larger transfers are typically initiated using RDMA engines, so as to free up the processor to perform meaningful work while the transfer is underway. Here, latency is typically less significant than throughput, especially for larger data transfers. Zero-copy is a key desired property for the RDMA transfers as it avoids the immense overheads introduced by the OS, such as slow system calls and copying data between memory and network buffers.

Designing networking components to sit on the FPGA fabric in the context of a HPC system environment presents its own challenges. The first one is the limitation of memory usage on the FPGA. Typically switching and routing requires significant amounts of memory, which is in limited supply in the FPGA fabric, and too slow to access from main memory. Designers must therefore be mindful of this and present tailor-made solutions which are less memory intensive.

The main reason for using an FPGA-based compute unit is its flexibility and capability to be used as an accelerator. Although having the networking components on the FPGA is beneficial for tight coupling between network, processor and accelerator it uses valuable resources on the FPGA. Care needs to be taken to minimize the footprint of the design, to leave maximum resources for the accelerator to be placed.

1.3 | Contributions

The aim of this work is to demonstrate a prototype which reconciles large scale networking with the SoC communication interfaces seen on the processor side of the MPSoC. We provide a detailed account of the implementation issues faced. By doing this we hope to enable the reader to gain valuable insight into the design aspects of networked MPSoC communication which require the most careful consideration. The

design of our NIC (Network Interface Card) enables transparent, NUMA-style (Non-Uniform Memory Access) read and write operations into remote locations, as well as user-initiated RDMA operations for data transfer between remote nodes. The main contributions of this work can be summarised as follows:

1. Design of a new NIC architecture which enables **(i)** Transparent read/write operations (in user-space) to regions of remote memory/accelerators, as well as **(ii)** User-initiated RDMA operations to allow high throughput data transfer to remote memory.

2. A low-latency protocol translation mechanism between ARM's memory-mapped AXI, used in the PS, and our interconnect intended for large-scale networking.

3. Implementation of a hardware prototype able to perform full end-to-end remote memory operations between two MPSoC devices, connected via 10Gbps serial links.

4. Outlining a hardware based resiliency mechanism in the NIC, which allows packet retransmission and safe use of early acknowledgements for AXI transactions.

5. A comprehensive evaluation of many important performance metrics such as latency, throughput, area, and power.

6. Testing functional correctness and robustness of the design with real-world applications. We run the STREAM memory benchmark, and a custom compute-intensive kernel from the weather forecast domain.

7. We identify issues/solutions surrounding the use of NUMA-style communication on a state-of-the-art MPSoC device.

2 | RELATED WORK

The design of clusters of FPGAs is not a new phenomenon. Several systems have been developed in the previous decade or so, with varying levels of coupling between FPGAs themselves, and with other standard compute elements. Each of the implementations outlined here however, are either ad-hoc networks not designed for general HPC, are non-scalable solutions or rely on Ethernet technology, which is unsuitable for the future of supercomputing.

The Maxwell cluster [16] of 64 FPGAs was built to analyze the applicability of FPGA acceleration for a set of HPC applications. They use a set of Xeon processors for running the application process, and interface to the cluster of FPGAs on a separate interconnect, which then act as an accelerator for kernels of computation. FPGA to FPGA communication is performed using point to point messages only, with no routing, so falls back on host CPU Ethernet communication for certain operations such as MPI collectives and data initialization etc.

Bunker and Swanson [17] create a 32 purely FPGA cluster with a hierarchical network. The FPGAs are grouped in *enclosures* of four, with an all-to-all connection made from single ended wires inside the *enclosure*. They analyze a series of possible solutions for Inter-enclosure communication, including 10G Ethernet and Xilinx Aurora.

SMCFA [18] is a small cluster of Xilinx Zynq-7000s. Communication between the devices is non-scalable, using bus based communication between “master" and “slave" devices. They use the ARM CPUs for transaction management and the FPGA fabric to accelerate feature extraction algorithms.

Zedwulf [19] is a 32-node cluster of Zynq SoCs designed to perform graph traversal algorithms. Communication is performed between a single switch using 1Gbps Ethernet. They trade off the performance and power consumption of the system, and show promising performance-per-watt results against x86 based implementations considering the ARMv7 based CPU architecture.

The AIREN interconnect forms an on- and off-chip network, for connecting in FPGA-based NoC designs with multi-FPGA clusters for HPC [20]. For the off-chip network they use multi-gigabit transceivers and link FPGAs together using SATA connectors. They used Xilinx Aurora for the link layer protocol and form a 4-ary 3-cube topology.

Markettos et al. [21] argue the case for custom-built interconnects on FPGA clusters. They create a new interconnect toolkit called BlueLink, which provides several layers of communication infrastructure; physical, link, reliability, routing/switching and application primitives. They compare their interconnect in terms of area and performance with some other common communication standards implemented for FPGA, such as Aurora, Ethernet, Interlaken and Infiniband.

Like [21], we argue that a custom network is preferable to out-of-the box soft IP solutions, although we differ from them in that we seek to create an interconnect for general HPC application use, rather than targeting a specific application with small message sizes only. Both [20] and [17] consider only standard soft IP solutions for their interconnect. Our interconnect provides tighter coupling between CPU-FPGA resources than [16], which views the FPGA as a co-processor rather than the full compute element.

APEnet+ is an FPGA-based network controller for implementing 3D-Tori [22]. The primary target for this network being GPU-accelerated HPC applications. The FPGA is connected to the compute element via PCIe, and communications between nodes is performed between the interconnected FPGAs. The controller features capability for lossless flow-control and a monitoring scheme for detecting network faults.

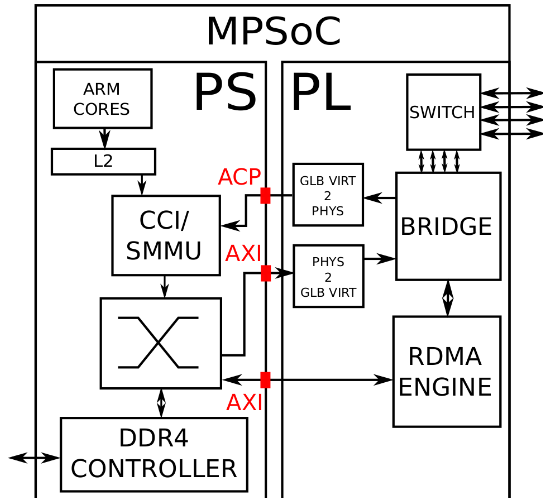

FIGURE 1 Elements within the non-reconfigurable processing system (PS), and programmable logic (PL) for system.

It uses a custom network protocol, and provides hardware support for RDMA operations, enabling low latency communication [23].

This system most closely resembles our work, but uses the FPGA solely to implement the interconnect resources, using it to move data between host-GPU or GPU-GPU using PCIe. Using PCIe devices like this requires additional copies of data to be made. They only support RDMA type transfers, and do not allow for transparent load/store operations in remote memory, enabling the hybrid MPI/NUMA programming model we provide support for in this work. We are also able to implement topologies other than the tori/meshes used in [23], due to our use of an arithmetic routing scheme as opposed to Dimension Order Routing. For instance, we provide routing schemes for both tree-like and Dragonfly topologies in [24], increasing the scalability of our system. There is also no system of hardware retransmission, which provides us an efficient mechanism to overcome the obstacles which limit the performance of the MPSoC.

3 | MPSOCS AND COMMUNICATION PROTOCOLS

The target for this work is Xilinx's Zynq Ultrascale+ device, a state-of-the-art MPSoC using 16nm FinFET technology [25]. Each of these MPSoCs contains 4 ARM-v8 A53 processors with shared L2 cache, two real-time Cortex-R5 processors, a Mali GPU, as well as plentiful FPGA fabric (up to ≈600k logic cells), all on the same die. This advance in the availability of hard-core compute within the device is of great interest for certain portions of the HPC community, particularly in more power constrained contexts. However, in order to scale to large clusters of these devices it is unsuitable to rely on the traditional methods of communication for FPGAs.

Figure 1 shows the distribution of system elements and interfaces between the ARM based processing system (PS) and the programmable logic (PL) of the MPSoC, with custom elements shown on the PL being completely user-defined and customizable. The setup shown is how we envisage the completed system to be set up, with separate communications channels between PS and PL for small messages through the cache coherent ACP (Accelerator Coherency Port), and the RDMA engine based upon a standard memory mapped AXI interface.

ARM's AXI (Advanced eXtensible Interface) is a communication interface designed for high speed on-chip communication. As the design of SoCs has become more complex the need for standardized interfaces has arisen to promote IP reuse and greater design productivity. AXI and its cache coherent extensions (ACE) have been the de-facto standard for communication between Xilinx IP for a number of years, and they form the only interfaces between the PS (Processing System, ARM subsystem) and PL (Programmable Logic, FPGA resources) domains in the Zynq Ultrascale+.

An AXI interface [26] consists of 5 independent channels of communication; Write Address (AW), Read Address (AR), Write Data (W), Write Response (B), and Read Response (R). These independent channels allow for separated control and data planes, and flexible-width data paths. The standard also supports burst transactions and out-of-order delivery. Figure 2 shows the basic transmission of data for read and write transactions within the AXI channels.

While these characteristics allow for high throughput and low latency communication within small, reliable on-chip networks or tightly-coupled devices, it has many characteristics which make it unsuitable for large-scale networking. The main reason is that the AXI standard provides no method of retransmission and that the channels contain only simplistic error codes, with no detection or recovery scheme included. These are based on the assumption that errors in packet delivery will be an extremely rare occurrence due to critical failures from which no recovery is possible. Obviously, this is not the case in large-scale networks where data traverse multiple switches over lengthy cabling.

The issues with AXI cause numerous problems in the design of an interconnect for large networks of MPSoCs. Given that the AXI standard defines the only low level interface for transactions between the Zynq PS and PL, a software solution cannot be obtained for packet retransmission that reconciles easily with AXI. Several standard options are available to the architect to deal with this. However, all of them have their own

FIGURE 2 Channels in AXI interface, taken from [26].

disadvantages which make them unsuitable for our use, which motivated the development of our custom network.

3.1 | Ethernet

Xilinx provides a wealth of IP to enable Ethernet networking capabilities. However, using TCP for retransmission causes its own problems. The networking stack requires numerous system calls, with multiple copies of the data being made, adding greatly to the latency of transmission. Advances have been made to provide support for RDMA transfers, such as through RDMA Over Converged Ethernet (RoCE) to reduce the overhead of the networking stack, freeing the CPU to continue with compute tasks. Unfortunately there are no freely available NIC IPs to provide support for this, and Ethernet also presents many other issues as a HPC network protocol.

The soft IP subsystem forming the Xilinx Ethernet MAC layer requires considerable resources in the FPGA. Given the wish to create direct PL-PL connections in a large-scale network, we must also consider the micro-architecture for the switching elements and the resources which this will consume. Typically Ethernet switches are based upon routing tables which are populated dynamically. The size of these routing tables is a determining factor in the performance of the network, as any packet destined for an IP address not contained within the routing table must be broadcast to all output ports. The use of large routing tables to store destination IPs is not feasible within the FPGA, owing to the limited block RAM resources. The alternative, using DDR memory to store these tables, would mean significant latency penalties for IP lookups at intermediate hops in the network.

This broadcasting of the packets destined for unknown IPs also causes limitations with the network topology. In order to prevent cycles in the topology creating broadcast storms we must overlay a virtualized spanning tree on top of the network for a given source-destination pair. What this means is that we are unable to make full use of path diversity in a given topology, as a given packet can only have one route through the network. In HPC applications taking full advantage of multipath routing would give much better network performance. While the ability to use multiple paths can be implemented in the TCP/IP software stack, the full path for packet/connection must be made at the source node alone. This does not provide sufficient flexibility in a very large-scale network to prevent congestion from building. Using per-hop routing decisions should give far superior network performance, reducing congestion considerably.

3.2 | Infiniband

Another obvious candidate for the networking protocol is Infiniband. Infiniband is a high throughput, low latency protocol that is common in the HPC domain, making up over a third of the system interconnects on the current TOP500 list (Nov' 2017). It provides capabilities for complex operations such as RDMA, multicast packets, and atomics, with capability to use very high signaling rates, making it ideal for HPC applications.

The major issue around the use of Infiniband is its complexity, and the lack of freely licensed/open source IP to support it inside the MPSoC. In any case, the protocol being highly complex means an in-house implementation for the FPGA would be incredibly time consuming, and likely use lots of the available FPGA resources. The possibility of using a dedicated card attached to the MPSoC accessed via PCIe or other means is discarded due to the additional overheads in terms of cost, power consumption, latency, compute density, etc. For these reasons we decided to use a custom packet format.

3.3 | Custom Network Switch

The architecture of the custom switch design is described fully in [24]. Written in Manchester using Verilog the switch can be implemented in the system without incurring the heavy costs associated with Infiniband. The switch has an AXI-stream interface which we use to wrap our custom network packet. It provides variable width data-path, with read/valid handshaking for the transfer of data to prevent buffer overflow . The connection between the MAC and PHY uses a standard XGMII (10G Media Independent Interface) interface.

The switch uses a 3-stage pipelined architecture with virtual cut-through switching in which the packets (composed of header, payload and footer) are split into 64-bit flits. The data sent and received by the asynchronous AXI-data stream FIFO is connected to the 10Gbps MAC layer which is attached to the 10Gbps transceivers (PHY). The PHY serializes/deserializes the data between the switches and transfers the data over an optical fiber. The clocks that feed the asynchronous FIFOs come one from the PHY on the receiving side and from the router on the transmitting side. The switch uses a 3-stage pipelined architecture with virtual cut-through switching in which the packets (composed of header, payload and footer) are split into 64-bit flits. The data sent and received by the asynchronous AXI-data stream FIFO is connected to the 10Gbps MAC layer which is attached to the 10Gbps transceivers (PHY). The PHY serializes/deserializes data between the switches and transfers it over an optical fiber. The clocks that feed the asynchronous FIFOs come from the PHY on the receiving side and from the router on the transmitting side.

Our switch uses an arithmetic routing scheme, which has clear benefits over alternatives in the context of large-scale networked FPGAs. The first is that we see lower area overhead in implementing the switch over Ethernet and Infiniband. As discussed previously, Ethernet relies on large routing tables to store information on destination ports to send packets over, and Infiniband is a far more complicated protocol to implement. They are infeasible for implementation inside the FPGA, as they are heavily memory constrained (only 32.1Mb of BRAM on the target chip).

The second major benefit of our protocol is that it enables the use of per-hop multiple paths for a single source-destination pair for many common HPC network topologies. This has obvious benefits for load balancing and throughput in the network over Ethernet. Moreover, it implements *Virtual Output Queues* (VOQs) [27] to eliminate Head-of-Line (HOL) blocking, thus minimizing network congestion and preventing the spread of congestion trees.

4 | NETWORK INTERFACE

The network interface (Figure 3) forms a bridge between the memory-mapped AXI domain used by the PS to access the PL, and the AXI-stream wrapped custom network protocol for use over the high speed serial links connecting the FPGAs. It consists of *five* main data interfaces. The first two are AXI stream interfaces, one master and one slave, and form the TX (transmit) and RX (receive) interface to the global interconnect (either a local switch or to a transceiver connected to an external switch). There are then two full AXI slave interfaces which handle incoming RDMA transfers and shared memory transactions from the local PS. Finally there is an AXI master interface which handles inbound transfers from the network. The NIC is attached to a Xilinx AXI Central DMA IP block, used for the RDMA transfers. We pass the configuration transactions from the PS through our NIC, enabling us to keep track of the transfers to be scheduled and support hardware retransmissions.

4.1 | Packetizer

The packetizer is formed of a state machine which waits for the arrival of a read or write request (AR/AW). Once a request has been initiated a 128-bit header can be generated in two flits. The current IP uses a 64-bit data path, as the Xilinx MAC/-PHY interface has 64-bit data width, transmitting only 64-bits per cycle, although this can be extended to a 128-bit datapath, as the PS-PL AXI interfaces are variable width. This would provide internal speedup, allowing us to operate at a lower frequency, or lower the latency within the PL data path.

On the outbound request (AR/AR/B) side there is no information which is required upon the arrival of its respective response (R/W), so forwarding is simple. A two-flit, 128-bit header is first formed, then the body is transferred, then a two-flit 128-bit footer follows. The header contains information regarding the destination address, the packet type and size, and error detection/codes, while the footer encodes the source address for the sending node and the ID of the transaction, among other information.

In the event of a read request (AR) the header and footer are formed, with no body needed. If a write is requested (AW/W) then the formation of the footer is stalled until the last piece of data has arrived. Data can only be accepted the cycle following an address request, but this is not an issue because the header requires two cycles for formation and egress from the Master-Stream port regardless, so latency is masked.

When a packet arrives from a remote node (incoming on the AXI-Stream Slave port) the header is decoded. If the incoming message is a write response (B) the footer is decoded to retrieve the ID, at which point the response packet can be reconstructed. If the packet is a read response (R) then we must queue the incoming data, and wait for the footer. As the read data has an associated transaction id (R_ID) we cannot form the Read Data (R) channel until the footer has been received. This is inefficient in terms of both additional latency and due to the additional buffering resources required. However, it is impractical to move the channel ID (R_ID/W_ID) to the header without extending the size of the header. Extending the full width of the header beyond 128-bits would mean that once the datapath is extended to 128-bits wide, two flits would be required for the header, and a lot of padding would

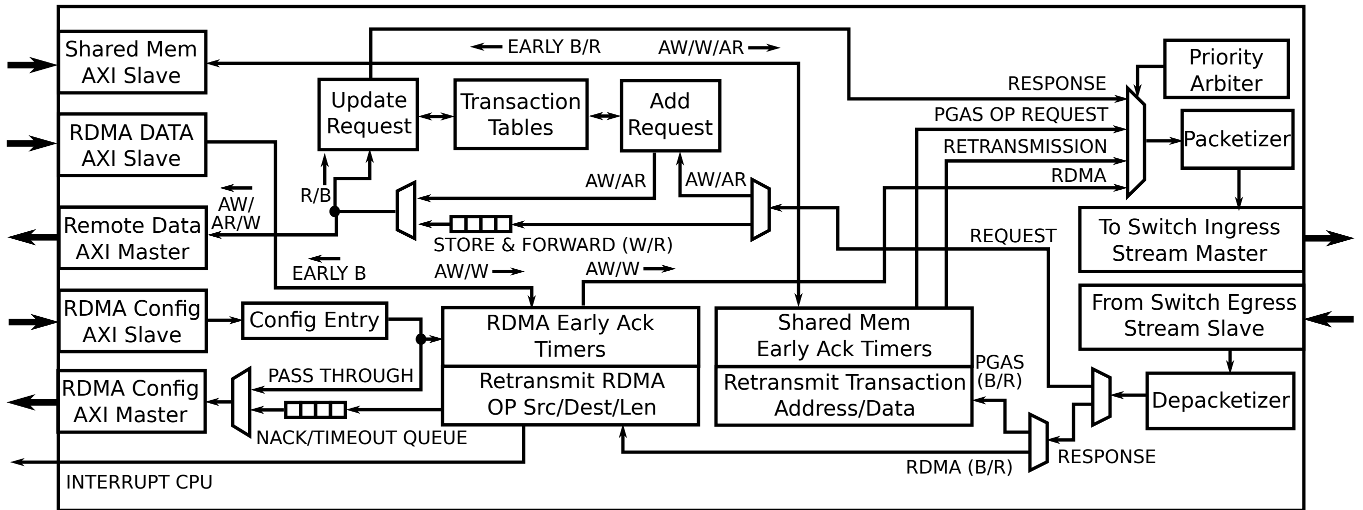

FIGURE 3 Micro-architecture the NIC.

then be required within the header (given the only additional information in the header to be the ID). Given the relatively small intended maximum packet size for the network protocol (for congestion control reasons), this would impact on the maximum available goodput of the network.

4.2 | Transaction Tables

If the remote packet is a request (AW/W/AR) then we must again wait for the footer to construct the ID, but this time we must also place a request in for access to a "transaction table". This table stores information about the outstanding transactions that have arrived from remote nodes, to which responses have not yet been sent back. Information must be stored here in order to route response packets back to their source. This is because the AXI standard does not have any signals for the source address. All routing of the response packets is done using the transaction ID signals (AW_ID,AR_ID,R_ID,B_ID). Typically this can be achieved by appending bits which correspond to the source port/link/location to the variable width ID signals, thereby allowing simple decoding and routing back to source. However, this is not suitable for a large-scale distributed system, as we cannot append additional bits which are encoded with the routing information for the whole system. For this reason we rely on the transaction tables.

The table is formed of an 8-entry binary CAM (Content Addressable Memory), to which requests for entry are submitted, and updated ID values are returned for AXI packet formation. Responses (R/B) then post an update to the CAM entry on their way back, retrieving their original ID for network packetization. The CAM search is performed on both the sending node coordinates, and the transaction ID. Each table entry also contains the original ID, a new ID issued at each new CAM entry,

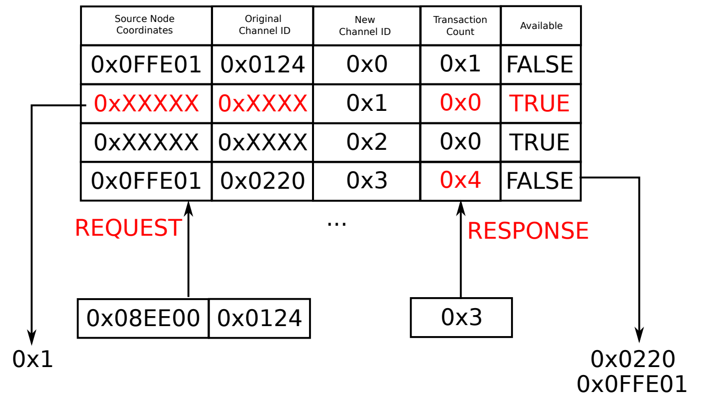

FIGURE 4 Example CAM table entries. Shows the request returning a new ID, and a response retrieving the source node location and the original ID for use in the remote source node.

a count on outstanding transactions, and information regarding whether this is an RDMA or shared memory transaction. Figure 4 shows the structure and insertion/replacement policy for the CAM.

If a new request is given which has the same transaction ID and source coordinates as an existing table entry the count of the entry is simply increased. In this manner we can handle multiple outstanding transactions from the same master in flight at once, for example from a large-scale DMA transfer. If the source coordinates are different then a new table entry must be issued regardless of whether the transaction ID is the same, as they are independent transactions. The returning AXI response then posts an update to the table, lowering the outstanding count. If the count reaches zero this entry will become available for a completely new transaction.

In the event that a new entry request is made and there is no available space in the table the request will be stalled until an entry becomes available. This is likely to cause issues with backpressure in the network in certain circumstances, but should be able to be alleviated by allowing a table large enough to provide sufficient entries to mask the latency of the average round trip time from the IP to memory and back, which should not be a significant number.

4.3 | Early Acknowledgements

In the Processing System (PS), transactions out to the Programmable Logic (PL) can cause serious performance degradation if the response is not provided in a timely fashion. This is because the ARM Cortex-A53 has an in-order pipeline, so will stall until the corresponding load/store instruction is completed, meaning the AXI response must be seen for the processor to continue execution. While this may be acceptable for communication within a single MPSoC, considering the round trip time for traversing a large network, this is completely unworkable.

Also worth noting is the fact that this can cause particular problems when debugging PS-PL communications for testing new IP. There is no hardware to provide negative responses using timeouts within the PS. This means if a rogue program writes to some undefined location in the PL and a response is not provided the whole system will hang (assuming OS and program sharing a single core).

While there is little to be done with regards to remote read operations because the processor needs the data to continue, we can alleviate the latency issues for remote writes by providing an early acknowledgement. We receive the full request into the NIC, and begin a process to create the early acknowledgement. We include a set of (currently 16) timers, which have an associated ID register. Available timers are popped from a queue of uninitialized timers. The ID is stored and the timer started. Once a given transaction has been granted an associated timer and all of the associated data has been transferred into the NIC (i.e. WLAST signal is seen high), then a response is sent back to the PS. If the timer overflows or we receive a negative acknowledgement then retransmission is attempted. If the real response arrives back from the remote node before timeout then the timer is reset and pushed back onto the list of unused timers.

Currently the timeout value is static in the timers, which may lead to suboptimal values being chosen. If the timeout is too short then it may be the case that a timeout occurs when the response is simply delayed traversing the network. However, if the timeout value is too long then the process of retransmission is delayed and so latency of the transaction is lengthened further still. Given that the round trip time will heavily depend on the distance to the destination and the system load, we intend to explore how to adjust the timeout values dynamically based on local information.

4.4 | Reliability Layer

Two separate methods of hardware packet retransmission are implemented in the system (Figure 3); one for the small low-latency remote load/store operations, and one for the bulky RDMA data transfers. Neither of them requires immediate intervention from the CPU. For the shared memory transfers we store the transactions on the local NIC, issuing a new ID for the transaction. In the event of a negative acknowledgement or a timeout we simply reissue the transaction. Multiple attempts are made, the limit of which is capped, at which point a more serious network error than packet loss or CRC (Cyclic Redundancy Check) failure is assumed and an interrupt is provided to the CPU. To prevent read-after-write dependency issues arising we block and queue all read operations in the NIC to addresses which reference to an in-flight write transaction until we receive the genuine acknowledgement from the remote side.

Given the area limitations on the FPGA it is infeasible to store the transaction data for RDMA transfers within the PL. In order to provide reliable transmission for RDMA transfers we keep track of the individual transactions which pass through from the DMA engine, storing their base address, destination address offset and transaction length using the pass-through AXI interface which configures the DMA, and keeping count of the transactions which are issued. We allow the DMA engine to complete a full transfer, storing the transactions which time-out or have negative acknowledgements. We then reconfigure the DMA from the NIC to retry these transactions, again capping the number of reattempts and then elevating the severity of the failed transfer.

While the retransmission is transparent to the programmer, performed completely in hardware and avoids the immediate utilization of CPU resources (unless severity is elevated), there are several caveats with this retransmission method which concern the software developer. Until there has been notification from the RDMA engine and NIC that the full transfer has successfully completed, the user must not alter the data being transferred, and the data must be pinned in memory. In the case where they wish to continue processing the data while transfers are in progress they must create a local copy of the data to ensure no inconsistencies in the data are introduced. This situation should be infrequent however, and copying can therefore be postponed; only creating a copy of the data *if* the local node wishes to change the data mid way through an RDMA transfer. This ensures no significant impact on the available memory bandwidth.

5 | EXPERIMENTAL SETUP

Our current system setup consists of two interconnected *Xilinx ZCU102* evaluation kits, connected using a 10Gbps SFP (Small Form-factor Pluggable) link. Each FPGA is loaded with the same bitstream, consisting of the design detailed in Figure 5 . We have created a simple address mapping IP block as an interim module to test the interconnect until a block is created which will perform the physical to global virtual address translation within the programmable logic side of the MPSoC. This needs to be performed for various reasons. Firstly, all addresses which leave and enter the PS must be physical addresses. There is no way to route through the internal interconnect out through to the PL using virtual addresses. There are only specific windows of memory within the Ultrascale+ that the PS interconnect will route to the PL, and only through a single static AXI interface. This is not customizable. For the purposes of this initial work we map only a small segment, 1MB of memory at 0xA0000000 which is visible to user-space applications, and issues AXI commands through the PS-PL interface HPM0.

The address mapper is a simple block which contains a configuration register and two small RAMs that can be written or read. The block can be configured to send AXI read or write requests, of varying burst length, with the target address and data being stored in the RAMs. The IP can be configured to send one shot requests or repeating requests with a given set of IDs, with a variable number of outstanding transactions allowed.

The DMA requires no additional work inside the PL, as the interface from the DMA to the PL is a standard memory-mapped AXI interface. Which can then be sent directly to the NIC (once the address has been remapped to mimic the behavior of the PL page table). The DMA engine does not require this mimicked translation process between the PS-PL interface and the networking IP, as the local DMA engine configuration registers are accessed using a fixed physical address. Configuration registers are loaded with source/destination address pairs, which can be any address, either global virtual addresses or physical. Data is pulled from remote memory using a physical address, and then pushed to the NIC with transactions formed with a global virtual address, complete with destination node coordinate information.

Since we only require two ports for the test we connect in and out port 0 of the first switch to in and out port 1 of the second switch, and tie the other IO ports off (ensuring that they are still synthesized as if connected). In the design we configure the address mapper to send to address 0x0520140011110000 (upper 22-bits 0x00014805). Configuring the *local port* value of the two routers thusly will emulate the routing of the packet to a node in another chassis of the network:

	Address	Cabinet	Chassis	Daughter Card
R1	0x00014403	00000101	0001	0000000011
R2	0x00014805	00000101	0010	0000000101

This means that the packet will be routed out of the network switch over the first output (port 0/TX_o0), towards the node in chassis 2. As the receiving node matches the node ID in the destination address we route out of the second switch through the local port (Local_o0). If the packet was destined for a different cabinet the packet would be directed out of the uplink port (port 3 in this implementation).

5.1 | Transceiver Setup

The Zynq Ultrascale+ MPSoC contains several transceiver types (GTH/GTY/GTR) with different characteristics (throughput capability, power consumption, PS/PL accessibility). In the setup we use the GTH transceivers with a line rate of 10.3125Gb/s, with a full duplex link – parallel lanes for TX (transmit) and RX (receive).

The transceivers are grouped into “Quads”, which is a bank of four transceivers, each consisting of TX and RX serial lanes, along with reference clock IO pins and routing. The reference clock is typically set up in *Input Mode*, in which the reference clock input is used to clock the transceiver. There is also available an *Output Mode*, in which the transceiver is clocked from another transceiver using the recovered clock (RXRECCLKOUT) [28]. The resulting output clock can then be routed out to be used at a different location. In our design we use two transceiver pairs from a single quad with the clocking in *Input Mode*.

To interface between the Media Access Control (MAC) layer (used to provide a preamble and end of packet as well as idles) and the transceiver we use the Xilinx 10G/25G Ethernet Subsystem IP block [29]. This block is capable of providing both MAC and PHY (Physical Layer) functionality, however, licensing restrictions prevent us from implementing the MAC functionality of the IP block on the FPGA The block is thus configured as a 64-bit PCS/PMA (Physical Medium Sublayer/Physical Medium Attachment) with 4 TX/RX pairs.

The MAC layer used in this work is an open source implementation with a compatible interface for the Xilinx 10G PHY. The MAC provides a CRC check and converts frames between XGMII (10G Media Independent Interface) and AXI Stream formats. The widest datapath this implementation currently supports is 64-bit, so other options will be sought to provide internal speedup, required for higher (25G) line rates. The Xilinx MAC functionality provided in the 10G/25G Subsystem IP can perform this, but with paid licenses only. The only properties changed for the MAC for correct functionality with our

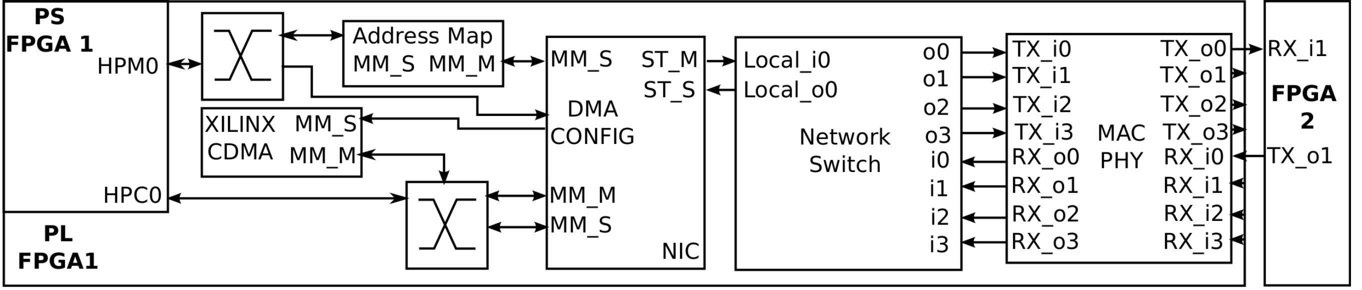

FIGURE 5 Full system setup between the two MPSoCs. MM_(M/S) is memory mapped AXI master/slave interfaces, and ST are the streaming interfaces.

custom network packets is a reduction in the minimum frame size. The Ethernet protocol states a minimum frame size of 512 bits, but our custom protocol is able to create frames as small as 256 bits (header and footer, with no body).

Three clocks are required to instantiate the PHY, a reference clock, an RX clock, and a *dclk* used to initialize the transceiver. For a 10Gbps line rate the PCS/PMA reference clock requires a frequency of 156.25MHz, the standard for a 10G Ethernet PHY (802.3) [29]. Since we use a 64-bit datapath in the PS-PL interface there is no internal speedup, so we set the RX clock and dclk to the same frequency.

In previous investigations using Xilinx Virtex-7/Ultrascale devices and soft-core Microblaze processors it was required to set up an external chip for the reference clock using I2C commands issued following reset from a hardware driver, as this was not done by default. Similar to the Virtex device, the Zynq contains Silicon Labs chips for this purpose (Si570, and Si53340 for low jitter), but no additional setup is required for the 10Gbps line rate, as the system defaults to a clock frequency of 156.25MHz [30].

In future extensions to this project it will be desirable to utilize the higher available line rates of the transceivers. If this is the case the clock will require setting up with a frequency of 390.625MHz. This can be done simply from the PS side as there is access to the two I2C buses as part of the Multiplexed IO banks (MIO 14,15,16,17 by default). This can be accessed using drivers which expose a character device in sysfs to be written to for setting up the clocks [31].

6 | EVALUATION RESULTS

We have performed several experiments, as well as gathering reports from Xilinx's Vivado tools during synthesis/implementation of the system in order to analyze the design of the NIC and the network switch. All of these experiments are run on two interconnected ZCU102 evaluation kits, with production grade MPSoC (part number EK-U1-ZCU102-G), using the setup shown in Figure 5 unless otherwise stated. The ARM PS is by default clocked at 1.2GHz, and the PL components are all clocked at 156.25MHz as required by the transceivers.

TABLE 1 Area utilization of interconnect in absolute values and % total resources on Xilinx xczu9eg-ffvb1156-2-i FPGA.

Component	CLB LUT(%)	CLB REG(%)	BRAM (%)
Total	**48374(17.6)**	**53139(9.70)**	**350(38.4)**
MAC/PHY Subsystem	**18620(6.79)**	**16659(3.04)**	**62(6.8)**
⇒ PHY (4 port)	7399	9468	0
⇒ ETH MACx4	2131	807	3
⇒ Async FIFOs	898	2128	20
⇒ Switch	1771	1810	30
NIC	**12876(4.70)**	**13797(2.52)**	**11.5(1.26)**
⇒ Shared Mem Reliability	2871	4833	0
⇒ RDMA Reliability	2273	3205	0
⇒ CAM Tables	2911	1360	0
⇒ FIFOs	4302	750	11.5
DMA Engine	**2215(0.81)**	**2169(0.40)**	**0**
On Chip Interconnect	**10823(3.94)**	**14851(2.71)**	**4.5(0.49)**
Address Mapper	**1691(0.62)**	**2862(0.52)**	**0**
Other	**2149(0.78)**	**2801(0.51)**	**272(29.8)**

6.1 | Area, Performance and Power

Table 1 shows a breakdown area consumption of the individual components of the system described in the top of Figure 5 . The table shows that the system utilizes around 17.6% of LUTs (Look-Up Tables) in the FPGA as logic, and uses

around 9% of the Block RAMs (implements 2x 8-entry CAMs in NIC). The high block RAM utilization (in Other) is caused by a block memory generator IP used for testing purposes. Considering this is the entirety of the networking hardware for 4 10G ports, as well as interfacing for PS-PL communication this seems a reasonable price to pay, and leaves ample resources for accelerator logic and additional network functionality to be implemented. We implement the CAM tables and other memory using distributed RAM in order to further preserve precious Block RAM resources.

All area comparisons we make here are based upon data available in Tables I and II of [21], which pulls implementation details from numerous sources to compare area consumption. The largest component by some margin in our design is the MAC/PHY subsystem, but compares similarly or favorably compared with other implementations. The open source MAC component we use for a single 10G lane consumes almost the same amount of LUTs as other implementations in the literature and over 3x less registers. We notice that the Infiniband implementation with reliability layer consumes about 32% more LUTs, suggesting that a smaller, more lightweight protocol can afford us area savings (not to mention licensing cost). The results for a hardware TCP implementation in an Ethernet based system are only provided with a single 10G link, and show an improvement in resources over our design, but given that the area results for our reliability layer compare similarly to the custom interconnect implementation given in [21], this seems acceptable.

The NIC consumes the vast majority of its resources implementing the CAM tables or FIFOs at input and outputs. Although this can be reduced by reimplementing the CAM as a cache with lower associativity, as discussed in Section 4.2. The additional latency of the table operations will be minuscule compared with network traversal, so should seriously be considered.

The FIFO resources listed here may be able to be reduced by optimizations in the implementation. The implementation is written using Bluespec, and for readability and ease of development in many cases we hold a structure rather than a single data item. This results in the FIFO in some instances containing redundant information and being of unnecessarily large width. For example, we hold full request structures to take the write data, which results in a FIFO width of 192 bits, but a width of 64 bits (only the data) is sufficient.. This is easily amended, but should be considered as a source of unnecessary overhead.

Table 2 shows a breakdown of the zero-load latency through each component of the networking hardware for a simple remote load/store operation. As can be seen, the total time for an 8-byte payload remote write operation (AWLEN = 0) to complete is around 1.5μs. This includes the return journey of the acknowledgement for the packet. This value is dominated by the traversal of the MAC and PHY, which accounts for nearly 85% of the latency. This value is quite typical compared with similar implementations [17], and can be heavily masked by the inclusion of the early acknowledgement for remote write operations. Notice that the 3-stage switch has a latency of 9 cycles. This is due to the stream interface which wraps the switch, and various additional buffering stages at output and input. The 2N latency on the depacketization stage arises from the fact that the ID for the transaction is stored within the tail of the packet, as discussed in Section 4.2. This latency will therefore be more pronounced with larger packet sizes, but given that the throughput is more important for these larger RDMA transfers, and given the high latency of the MAC/PHY subsystem, this should not be an issue.

TABLE 2 Breakdown of zero-load latency for a remote write and return of acknowledgement. N is the number of payload flits, and the component breakdown latency for NIC is from first flit input to last output (given transformation of interface, this seemed most logical).

Component	Latency (cycles)	Latency (ns)
Total (RTT)	**232**	**1485**
NIC (MM 2 S)	8 (7+N)	51.2
Switch Traversal	9	57.6
NIC (S 2 MM)	9 (7+2N)	57.6
Sin MAC0 - Sout MAC1	100	640

In Table 3 we see a breakdown for the estimated power consumption from the Vivado tools after place and route of the FPGA. As we expect from our design the PS consumes the majority of the dynamic power (≈60%), owing to the fact that the PS clocks at 1.2GHz, and the PL is clocked at 156.25MHz. The majority of the power in the PL is consumed within the transceivers, which cannot be improved upon, as these are hard components within the system. We also implemented the same design with the Xilinx 10/25G Ethernet Subsystem component configured as both MAC and PHY, eliminating the need for the separate open source MAC block. (N.B. Licensing restrictions only prevent us writing the final bitstream, not running implementation.) The reported consumption of this component is 1.191W, which is around 75% of the 1.579W total consumption we see for the PHY plus 4 individual MAC components. This efficiency gain is small in relation to the total overall consumption, and so we are not concerned with the additional power.

TABLE 3 Component power consumption estimates following place and route.

Component	Power (W)
Total	**5.344**
PS	**3.198**
MAC/PHY Subsystem	**1.579**
⇒ PHY	1.124
⇒ MAC x4	0.069
⇒ FIFOs	0.014
NIC	**0.201**
Router	**0.097**
Config Interconnect	**0.128**
Address Mapper	**0.029**

6.2 | Stream Benchmark

In order to test the effects of communicating in a shared memory system we use the standard STREAM benchmark [32] for memory performance. This benchmark performs a series of memory operations using different access patterns, in order to test the whole memory subsystem. The code is compiled from C source using GCC with -O2 optimization flag, and *not* using OpenMP. This experiment is run to show correctness over performance, as this model of shared memory communication is not designed for data transfer in this manner, but for small synchronization and barrier messages. The results are given in Table 4 , and show the achievable memory throughput for the benchmark running in three configurations:

1. Memory is allocated in the local DRAM attached directly to the PS through the cache hierarchy.
2. Memory is allocated in a local Block RAM inside the PL, having to be accessed through the Full Power Domain HPM master port.
3. Memory is allocated in a remote Block RAM, having to be accessed across the network 1 hop away.

We see that the degradation in performance is significant (≈2 orders of magnitude) when we have to use the PL to access memory. There are several factors which cause these poor results. As the user accesses this memory transparently as a simple STR/LDR instruction, the limitations of the processor come heavily into play. The ARM Cortex-A53 is an in-order dual issue CPU. Meaning a maximum of only two read/write transactions can be in flight at any point in time. The processor's pipeline will block until it receives the acknowledgement of successful write or the data from a read operation.

Another cause of the performance degradation is the fact that we can use the entire memory hierarchy when accessing the DRAM in the PS. By accessing the PL through the AXI master ports we are unable to cache the resulting data. Obviously this is required as there is no way of keeping track of stale data in the PS side. We cannot know what a given hardware block is doing at any point in time, so cannot know that it is not changing the data.

We see that the throughput to access remote memory is about an order of magnitude slower than using local PL BRAM. This makes sense from our observations on latency of accesses and the CPU blocking. We see typically it takes about 150ns round-trip time for a write/read into a BRAM in the PL. This is about 10x less latency than we see in the $\approx 1.5\mu$s round-trip time for a remote operations (see Table 2) traveling 1 hop each way. Given the blocking nature of these operations the latency in acknowledgements causes the degradation of the performance we observe here.

It is obvious that this model of communication is not suitable for data transfer between remote memory regions, and that the RDMA operation is required to provide a far more appropriate method to transfer larger chunks of data. The shared-memory path to the network here is only suitable for simple low-latency communication between processors, as typically seen in synchronization messages or control packets. The advantage in having this kind of communication present in the system is that the latency of giving access to the user of the whole memory space allows us to reduce the latency of transfer for these small packets. DMA for these transfers is not sensible because of the buffer setup and memory copies involved. We also prioritize the forwarding of the shared memory requests over RDMA transfers in the NIC. All of which incurs additional latency penalties.

6.3 | Throughput Tests

In order to test the performance of larger RDMA transfers through the network we set up two experiments. In the first we use a traffic generator to vary the packet size and number of possible simultaneous in-flight transactions to show the maximum attainable throughput of the system over a single link. The second experiment uses the completed RDMA engine to send varying sizes of messages, pulling memory from the PS and sending it over the network, waiting for every transaction to be properly acknowledged. We show the effectiveness of adding early acknowledgements to the NIC in improving the performance of the RDMA transfers.

Figure 6 shows how we can increase the achievable throughput over the NIC and switch by varying the number of possible outstanding transactions the DMA can issue simultaneously, and by varying the burst length (and thus network packet size) of the DMA transfers. We see that by increasing the maximum payload size from 128B (16×64-bit) to 512B

TABLE 4 Results for running STREAM benchmark

Function	PS DRAM				Local BRAM				Remote BRAM			
	Best Rate (MB/s)	Avg Time (s)	Min Time (s)	Max Time (s)	Best Rate (MB/s)	Avg Time (s)	Min Time (s)	Max Time (s)	Best Rate (MB/s)	Avg Time (s)	Min Time (s)	Max Time (s)
Copy:	3345	0.0489	0.04784	0.0497	45.4	3.52	3.52	3.53	4.8	33.6	33.6	33.6
Scale:	1826	0.0887	0.0876	0.0905	44.0	3.64	3.64	3.64	4.7	33.8	33.8	33.8
Add:	2033	0.118	0.118	0.119	44.2	5.43	5.43	5.43	4.7	50.8	50.8	50.8
Triad:	1683	0.144	0.142587	0.151	44.6	5.38	5.38	5.38	4.7	50.8	50.8	50.9

(64×64-bit) we are able to increase the saturation point for the throughput by ≈32%. Using a transaction size greater than this may have adverse effects on the network. Keeping packet length relatively short allows us to perform better load balancing, and will aid in lowering the congestion in the network. The increase in throughput is due to the packet header and footer, which contribute 4 flits of overhead on the link per packet, given our 64-bit datapath. This gives a reduction in the packet overhead from 20% (20/16) to 6.3% (68/64). If we were to increase the data width to 128-bits (reducing header and footer to a single flit each) the packetization overhead will drop further still to 3.1%.

We see that saturation occurs at 8.56Gb/s, which is seemingly lower than the 9.4Gb/s promised by the GTH transceivers (given our packet overhead and the link being run at 10Gb/s). However, this additional drop can easily be accounted for. The MAC layer adds a preamble at the beginning of transmission and an end-of-packet flit at the end. An inter-frame gap is also required between packet transmission to aid in keeping the transceiver clocks synchronized. This results in a raw throughput drop of about 9% in our specific case (though this is dependent on frame size).

Adding the early acknowledgements to the system we see that the latency of waiting for acknowledgements is masked completely if more than a single packet can be in flight at any one time. The reason that there is degraded performance in the event that we have a single in-flight transaction limit comes from the implementation of the timer module for the early acknowledgement. When a request is presented, a request for an available timer to begin takes 2 clock cycles. This timer can only be requested once the entire request has been presented. What this means is that there is a bubble in the pipeline which is only masked when a separate request can begin writing data into the NIC while the timer is being started for the previous transaction.

In Figure 7 we see the time taken for an RDMA operation to process and receive (genuine) acknowledgement for a given message size. The baseline for small sized messages is around 1.9μs, which stays consistent until the message size grows to over 64B. We see the divergence in performance in the

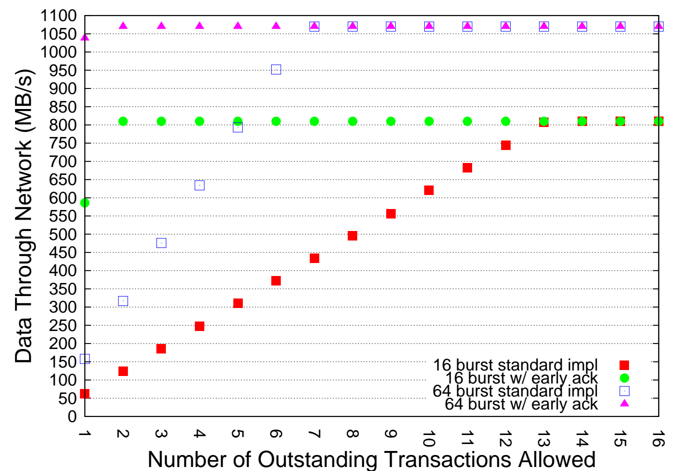

FIGURE 6 Throughput of network varying max in-flight transactions and burst length.

results such that for larger messages adding early acknowledgements to the system results in a transfer time of roughly half. This difference arises because the RDMA engine cannot issue more than five simultaneous transactions per channel. This difference would become even more pronounced if the distance between source and destination were more than a single hop, as the latency between issuing new transactions would grow even further. These results are encouraging given that they include the return hop time for the acknowledgement.

6.4 | Weather and Climate Simulation Kernel

In our sibling EuroExa project, we are porting kernels from the LFRic model to FPGA-based systems. LFRic (named in honour of Lewis Fry Richardson) is the new weather and climate model which is being developed by the UK's Met Office and its partners for operational deployment in the middle of the next decade [33]. High quality forecasting of the weather on global, regional and local scales is of great importance to a

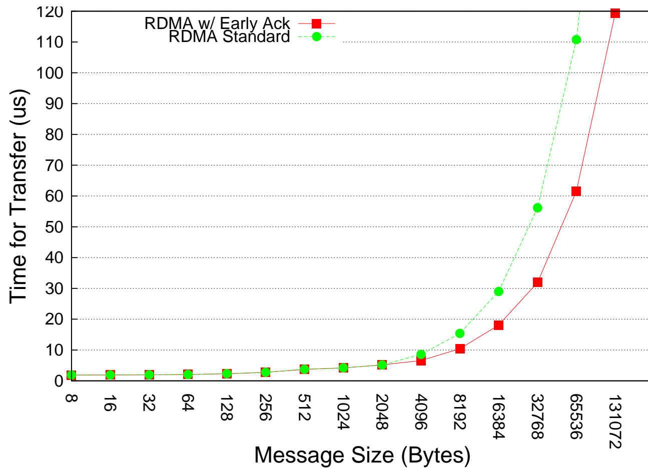

FIGURE 7 Time for RDMA transfers to complete for different message sizes.

wide range of human activities and exploitation of latest developments in HPC has always been of critical importance to the weather forecasting community.

In this context, we are looking at how kernels from LFRic can be accelerated using FPGAs. An example of such a kernel is a matrix-vector update (known as dgemv in the BLAS [34] library) which can contribute significantly to execution time. The matrix-vector updates have been extracted into a kernel test program and converted to C. There are dependencies between some of the updates and a colouring scheme is used such that nodes within a single 'colour' can be computed in parallel. This is used to produce independent computations for multi-threading with OpenMP or to be exploited by FPGA accelerators.

We used the Vivado HLS tool to generate, from C code, an IP block for use in this work. The inner loop of the C code was hand unrolled to enable maximum pipelining within the IP block. We run the kernel in five separate configurations to test correctness, stability and performance of the network. In the first one we perform computation in the PS with data held in DRAM. The next two use the PS and hold the data in either local or remote BRAMs. The final two perform computation in a local and remote accelerator block, with the data provided to the accelerator block from BRAM. The results are shown in Table 5 .

We see that the latency of using remote resources is around an order of magnitude slower, as we would expect from the observations of the STREAM benchmark results. While we see a speedup in computation time over the CPU-BRAM implementation, this does not take into account the memory hierarchy, which as in the stream benchmark, gives significant benefits in performance. Where the data is stored in DRAM

TABLE 5 Climate kernel execution time.

Implementation	Execution Time (s)
CPU Execution in DRAM	0.077698
CPU Execution in Local BRAM	7.520
CPU Execution in Remote BRAM	68.232
FPGA Execution in Local BRAM	2.533
FPGA Execution in Remote BRAM	24.186

during CPU execution rather than inside FPGA resources execution time is significantly faster than using the accelerator block, due to the memory bandwidth limitations discussed previously, and the CPUs use of the cache hierarchy.

What this experiment shows primarily is the ability of our network to be utilized for accessing distributed FPGA resources. Implementation is currently under way to enable the accelerator block to stream the data in bursts directly from RAM itself (basically acting as a DMA). This should provide significant improvement in the performance (as we see in the throughput results in Figure 6), and eliminate the bottleneck that we see. We show that this kernel is capable of executing faster in the FPGA than on the CPU, despite being clocked in this instances $\approx 8\times$ slower. The next step in implementation is to overcome the memory bottleneck. Significant additional infrastructure is also being worked on in a sibling project [35] for viable real-world exploitation, such as runtime reconfigurability, task scheduling and resource allocation.

7 | DISCUSSION AND FURTHER WORK

We present in this paper a novel Network Interface architecture, suitable for facilitating shared memory communications over large scale networks of Xilinx MPSoCs. Our solution provides a method of eliminating many of the issues surrounding the use of these devices for HPC applications. We solve all the problems involved with bridging between the AXI interface, designed for small scale, highly reliable networks, and large HPC networks which aim to scale to hundreds of thousands of nodes. By providing a method of early acknowledgement to the local AXI transactions we overcome the massive performance degradation that is seen with shared memory communications from the PS, owing to the dual-issue, in order pipeline of the ARM A-53s. This also allows the DMA engine to saturate the links to achieve a suitably high throughput over the 10G transceivers.

While this paper is focused on the hardware design and implementation issues surrounding the Xilinx Zynq Ultrascale+, there is one key component of the system which prevents the solution from being utilized for full-scale HPC applications. We require a method of translating from physical

addresses which are able to reach the PL, to a global virtual address space. A solution to this problem requires significant systems software development, setting up the SMMU within the PS, and writing drivers to cope with tables that would be placed in the PL. There are many issues to be faced with this. The most pressing of these is how we plan to reconcile the need for exa-scalable page tables with the limited resources of the FPGA. One solution to this may be to segregate the network into different domains, and progressively translate the virtual address as it travels through the network. Therefore very large regions of address space can be mapped into a single entry, with only a few bits of the address being considered in each portion of the network. This concept is similar to the use of subnets in IP routing, and is proposed in this work by Katevenis specifically for interprocessor communication [36].

Once we can translate into this global virtual address space and run full MPI applications we can analyze performance metrics and tune the system, performing some of the additional work discussed in Sections 4.2 and 4.3. Finding a suitable value for the acknowledgement timeout will be important for striking a balance between prompt retransmission and coping with any jitter experienced by packets traveling through the network, but requires a full network with real contention for resources to test. By the same token, limiting the associativity of the CAM will provide us a smaller design, and give us the ability to handle more outstanding transactions in the same footprint. However, this may not be necessary depending on the traffic patterns we see. It is therefore important that we are able to evaluate the system eventually using real-world HPC applications.

We plan in the immediate future to provide a deeper analysis of the micro-architecture of the reliability layer which has been implemented, and compare it with some other standard hardware and software based solutions. There are likely to be several improvements we can make to this solution, and several parameters such as queue length, retransmission attempt limits, variable retransmission timeouts etc. which can be investigated more thoroughly to provide a more performant solution.

8 | CONCLUSIONS

As the High-performance Computing (HPC) community closes in on the exascale milestone, new architectures for achieving greater performance per watt are desperately sought. We argue that for certain workloads, the use of MPSoCs with low-power mobile processors and FPGA fabric can be utilized to great effect in this cause. They are able to provide dense packaging and tight coupling for cpu/memory/accelerator/network, reducing wire length and data transfer (thereby reducing power consumption).

We show that standard networking solutions for FPGA and HPC are unable to provide us with all of the properties for a large scale cluster of FPGAs that we desire; namely price, flexibility and low overheads on the FPGA. We present a novel network interface and switching architecture, which use a custom network packet designed for use in networks of Xilinx MPSoCs for HPC applications, in a shared memory context. We provide a method for performing transparent low-latency access to a global shared memory via standard load/store instructions, and a method for performing large data transfers over a user-initiated RDMA, avoiding costly system calls and memory copies over traditional networking solutions. We propose a reliability layer within the NIC which allows for hardware retransmission of packets, without immediately resorting to intervention by the CPU.

We discuss in detail the design of the hardware, and the factors which influenced the design based upon the use of the Xilinx Zynq Ultrascale+ MPSoC. We hope that these issues (primarily regarding the use of AXI as the interfacing standard to the ARM Processing System in the device) have been discussed sufficiently so as to give the reader an insight into the main considerations when using such a system or designing a new architecture.

Preliminary results show that this prototype design is comparable in terms of performance and FPGA area consumption to alternative implementations, and that the inclusion of a system to provide early acknowledgements for write transactions can provide significant latency and throughput benefits. The design is lightweight and leaves considerable FPGA resources for use as an application accelerator, and the design allows for many more exotic network features to be explored and implemented in the future.

References

[1] Bohr Mark. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter.* 2007;12(1):11–13.

[2] Marazakis Manolis, Goodacre John, Fuin Didier, et al. EUROSERVER: Share-anything scale-out micro-server design. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016:678–683IEEE; 2016.

[3] Fujitso Post-K Supercomputer https://www.top500.org/news/fujitsu-switches-horses-for-post-k-supercomputer-will-ride-arm-into\-exascaleAccessed: 15-11-17; .

[4] Nurvitadhi Eriko, Sim Jaewoong, Sheffield David, Mishra Asit, Krishnan Srivatsan, Marr Debbie. Accelerating recurrent neural networks in analytics servers: comparison of FPGA, CPU, GPU, and ASIC. In: Field Programmable Logic and Applications (FPL), 2016 26th International Conference on:1–4IEEE; 2016.

[5] Cooke Patrick, Fowers Jeremy, Stitt Greg, Hunt Lee. A comparison of correntropy-based feature tracking on FPGAs and GPUs. In: Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on:237–240IEEE; 2013.

[6] Kozikowski Grzegorz, Papamanousakis Grigorios, Yang Jinzhe. Potential future exposure, modelling and accelerating on GPU and FPGA. In: Proceedings of the 8th Workshop on High Performance Computational Finance:4ACM; 2015.

[7] Hefenbrock Daniel, Oberg Jason, Thanh Nhat Tan Nguyen, Kastner Ryan, Baden Scott B. Accelerating Viola-Jones face detection to FPGA-level using GPUs. In: Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on:11–18IEEE; 2010.

[8] Hussain Hanaa M, Benkrid Khaled, Erdogan Ahmet T, Seker Huseyin. Highly parameterized K-means clustering on FPGAs: Comparative results with GPPs and GPUs. In: Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on:475–480IEEE; 2011.

[9] Abel Francois, Weerasinghe Jagath, Hagleitner Christoph, Weiss Beat, Paredes Stephan. An FPGA Platform for Hyperscalers. In: High-Performance Interconnects (HOTI), 2017 IEEE 25th Annual Symposium on:29–32IEEE; 2017.

[10] Intel Completes Altera Acquisition https://insidehpc.com/2015/12/intel-completes-acquisition-of-altera/Accessed: 15-11-17; .

[11] Ovtcharov Kalin, Ruwase Olatunji, Kim Joo-Young, Fowers Jeremy, Strauss Karin, Chung Eric S. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper.* 2015;2(11).

[12] Sundararajan Prasanna. High performance computing using FPGAs. *Xilinx White Paper: FPGAs.* 2010;:1–15.

[13] Xilinx . Vivado Design Suite User Guide, High-Level Synthesis, UG902 Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf2014.

[14] Nikhil Rishiyur. Bluespec System Verilog: efficient, correct RTL from high level specifications. In: Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on:69–70IEEE; 2004.

[15] Papakonstantinou Alexandros, Gururaj Karthik, Stratton John A, Chen Deming, Cong Jason, Hwu Wen-Mei W. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In: Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on:35–42IEEE; 2009.

[16] Baxter Rob, Booth Stephen, Bull Mark, et al. Maxwell-a 64 FPGA supercomputer. In: Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on:287–294IEEE; 2007.

[17] Bunker Trevor, Swanson Steven. Latency-optimized networks for clustering FPGAs. In: Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on:129–136IEEE; 2013.

[18] Li Lin, Yang Quansheng. SMCFA: A Zynq-based stacked multi CPU-FPGA architecture. In: :303–306IEEE; 2016.

[19] Moorthy Pradeep, Kapre Nachiket. Zedwulf: Power-performance tradeoffs of a 32-node zynq soc cluster. In: :68–75IEEE; 2015.

[20] Schmidt Andrew G, Kritikos William V, Gao Shanyuan, Sass Ron. An evaluation of an integrated on-chip/off-chip network for high-performance reconfigurable computing. *International Journal of Reconfigurable Computing.* 2012;2012:5.

[21] Markettos A Theodore, Fox Paul J, Moore Simon W, Moore Andrew W. Interconnect for commodity FPGA clusters: standardized or customized?. In: Field Programmable Logic and Applications (FPL), 2014 24th International Conference on:1–8IEEE; 2014.

[22] Ammendola Roberto, Biagionil Andrea, Frezza Ottorino, et al. Design and implementation of a modular, low latency, fault-aware, fpga-based network interface. In: Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on:1–6IEEE; 2013.

[23] Ammendola Roberto, Biagioni Andrea, Frezza Ottorino, et al. APEnet+: high bandwidth 3D torus direct network for petaflops scale commodity clusters. In: Journal of Physics: Conference Series, vol. 331: :052029IOP Publishing; 2011.

[24] Concatto Caroline, Pascual Jose A., Navaridas Javier, et al. A CAM-free Exascalable HPC Router. In: Architecture of Computing Systems (ARCS):to appear; 2018.

[25] Xilinx . Ultrascale MPSoC Architecture https://www.xilinx.com/products/technology/ultrascale-mpsoc.htmlAccessed: 15-11-17; .

[26] ARM . AMBA AXI and ACE Protocol Specification, IHI 0022D, ID102711 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.htmlAccessed: 15-11-17; 2011.

[27] Dally William, Towles Brian. *Principles and Practices of Interconnection Networks.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2003.

[28] Xilinx . Ultrascale Architecture GTH Transceivers, User Guide, UG576, pp. 24 Available: https://www.xilinx.com/support/documentation/user_guides/ug576-ultrascale-gth-transceivers.pdf2017.

[29] Xilinx . 10G/25G High Speed Ethernet Subsystem v2.1, User Guide, PG210, pp. 135 Available: https://www.xilinx.com/support/documentation/ip_documentation/xxv_ethernet/v2_1/pg210-25g-ethernet.pdf2017.

[30] Xilinx . ZCU102 Evaluation Board, User Guide, UG1182, pp.46 Available: https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf2017.

[31] Controlling FCLKs in Linux http://www.wiki.xilinx.com/Controlling+FCLKs+in+LinuxAccessed: 15-11-17; .

[32] McCalpin John D. A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter.* 1995;19:25.

[33] Mullerworth Steve. From ENDGame to GungHo then LFRic. In: ECMWF Workshop on Scalability; 2014.

[34] BLAS (Basic Linear Algebra Subprograms) http://www.netlib.org/blas/Accessed 15-11-17; .

[35] Mavroidis Iakovos, Papaefstathiou Ioannis, Lavagno Luciano, et al. ECOSCALE: Reconfigurable Computing and Runtime System for Future Exascale Systems. In: DATE '16:696–701EDA Consortium; 2016; San Jose, CA, USA.

[36] Katevenis Manolis GH. Interprocessor communication seen as load-store instruction generalization. In: In The Future of Computing, essays in memory of Stamatis Vassiliadis, K. Bertels ea (Eds.), Delft, The NetherlandsCiteseer; 2007.

How cite this article: J. Lant, C. Concatto, A. Attwood, J. A. Pascual, M. Ashworth, J. Navaridas, M. Luján, and J Goodacre (2017), Shared Memory Communication in Networks of MPSoCs, , .