# A Transport-Layer Network for Distributed FPGA Platforms

***Abstract*—We present a transport-layer network that aids developers in building safe, high-performance distributed FPGA applications. Two essential features of such a network are virtual channels and end-to-end flow control. Since different virtual channels can have vastly different traffic patterns, a proper network design requires flexibility in setting buffer sizes and flow control credits. In addition, the protocol must have very low latency and low memory resource requirements, because the communication links between FPGAs have very low latency, and FPGAs have limited on-chip memory. These resource requirements make protocols such as TCP/IP unsuitable in this environment. Our network implements these features, and takes advantage of the low error characteristic of a rack level network deployment to implement a low overhead credit based end-to-end flow control. Our design has many parameters in the source code which can be set at the time of FPGA synthesis.**

**Our prototype cluster, which is composed of 20 Xilinx VC707 boards, each with 4 20Gb/s serial links, achieves effective bandwidth of 85% of the maximum physical bandwidth, and a latency of 0.5us per hop. Our network exposes a variable width FIFO channel abstraction, with the ability to adjust buffer size and flow control credits per channel. Several applications have already been developed using this network. The user feedback suggest that these features make application development significantly easier.**

## I. Introduction

In order to tackle large data intensive applications, many modern FPGA-based deployments are exploring the use of FPGA clusters, where a network of FPGAs are deployed and a large body of work is distributed across the FPGAs. A network protocol for an FPGA cluster largely has three important criteria: (1) it must be easily usable by an application developer, (2) It must have high performance with low latency, and (3) it must consume only a small amount of scarce on-chip FPGA memory.

Two essential features for a usable network implementation are virtual channel and end-to-end flow control, both of which correspond to the transport layer of the OSI network model. Virtual channels are useful because most distributed applications need to communicate multiple types of messages such as command, data and status packets. The messages often have different priorities which when sent on a single link, can cause head-of-line blocking. Without virtual channel support, the developer must handle multiplexing the network link manually. Another crucial feature for building safe distributed systems is end-to-end flow control, which is needed so that one blocked channel would not block other channels. Ensuring these properties without a proper transport layer protocol makes the development of high performance distributed FPGA applications difficult.

An FPGA cluster is often networked using low-overhead link-layer protocols such as Aurora which runs on the high-speed serial transceivers included in the FPGA fabric. Such links provide reliable multi-gigabit bandwidth at a sub-microsecond latency. Such low-latency network fabric and scarce on-chip memory resources on FPGAs make TCP/IP not an attractive option.

For deadlock-free operations, all virtual channels need separate packet buffers, and such packet buffers have to be large enough to mask network latency as well as bursts from multiple sources. Scarcity of on-chip memory resources can limit the bandwidth even in the presence of low-latency inter-FPGA networks. A solution may be to use a large off-chip DRAM packet buffers, but in such a scenario, the high performance serial links will consume a non-trivial amount of DRAM bandwidth which may affect application performance, especially if the application is using an accelerator on the FPGA. Another solution is clever allocation of buffer space by allowing different amount of buffers for different channels. Application developers can adjust the buffer space per channel to meet the performance criteria without increasing the total buffer requirement.

The contributions of this paper are twofold: We present the design of a paramterized low-overhead transport level network for a cluster of FPGAs that implement the useful features described earlier, and we evaluate the performance of our networ design using a prototype deployment.

Our network design includes transport layer implementations such as virtual channels via multiplexing, and end-to-end flow control. It features an end-to-end low-overhead credit-based flow control per virtual channel, making a distributed FPGA application developer's job much easier. It also includes a network layer implementation including packet forwarding. In our router, we make use of the high reliability of the serial link and deterministic routing to ensure lossless in-order arriving of packets, greatly simplifying the transport layer protocol.

Our transport layer is parameterized such that flow control features for each virtual channel can be configured at FPGA synthesis time. Parameters include buffer size and flow control credits. We demonstrate that a parameterized transport-layer implementation can achieve high performance in a distributed FPGA environment while maintaining a small BRAM footprint, by adjusting a few parameters to best fit the usage characteristics of a virtual channel.

We have implemented a prototype of our network on a cluster of 20 Xilinx VC707 FPGA development boards, with 4 20Gb/s serial links each. Our prototype implementation achieves an effective bandwidth of 17Gb/s per link, which is

85% of maximum physical link bandwidth, at a latency of 0.5us.

The rest of the paper is organized as follows: Section II covers the previous and related work. Section III describes our implementation of the network and transport layer, and Section IV describes the details of a prototype implementation of the network. Section V presents the performance evaluation of our implementation, and conclude in Section VI.

## II. Related Work

FPGAs offer very desirable performance and power characteristics, but modern data-intensive applications often require more resources that are available on a single FPGA chip. As a result, exploration of distributed FPGA computing systems is gaining popularity. The scale of distributed FPGA system being built range from a cluster-in-a-box systems such as BlueHive [1], to rack-level deployments such as Maxwell [2], to datacenter scale deployments such as Catapult [3]. Such systems offer a much better power performance characteristics over their off-the-shelf server counterparts. Attempts to exploit the different characteristics of various computing entities such as FPGAs, GPGPUs and CPUs using a heterogeneous cluster have also proven successful [4]. Some have explored inserting FPGA accelerators into the computation datapath, so acceleration happen without the overhead of copying the data to the FPGA accelerator. In BlueDBM [5], the FPGA managed the data transfer between distributed flash devices over an integrated controller network, achieving very low latency acceleration.

The TCP/IP network protocol stack is by far the most popular protocol for internetworking computer systems, but it may not be a good fit for inter-FPGA communication. The IP protocol is a best effort delivery protocol designed for a large and unpredictable network. Because packets delivered over IP may be lost or reordered, it is up to TCP to implement end-to-end management and ensure safe delivery. Such requirements makes the TCP protocol complex and resource heavy, making it a good fit for the internet, but not the best choice for datacenter or rack-level deployments where the constraints are different. Such complexities also make it unfit for implementation on FPGAs. Some FPGA cluster projects have used Ethernet's physical and data link layers for its network, but full implementation of the TCP/IP stack is rare unless it has to interact with a legacy interface [6].

Datacenter scale protocols such as Infiniband [7] provide better managed flow control in the network layers, ensuring no packets are dropped due to network congestion. This allows a more efficient transport layer protocol implementation. A modified TCP protocol DCTCP [8] aims to achieve similar goals by using a special flag set by a router when a packet has experienced congestion to intelligently modify traffic rate, resulting in a much smaller packet buffer requirement. Infiniband also offloads a major part of the protocol implementation to the hardware NIC, in order to achieve much better performance than software implementations of other protocols.

Most existing network solutions provide many transport layer features, such as virtual channels and end-to-end flow control. Virtual channels multiplex the network link so that it can be used by multiple components as if it had exclusive access to a network link. End-to-end flow control hides underlying network details from the virtual channel endpoints, by resending packets that may have dropped, managing reorder buffers to handle out-of-order delivery, or managing flow control credits so that congestion does not cause packet drops.

Due to the high engineering and performance overhead of existing network solutions, many inter-FPGA networks on a distributed FPGA deployment are implemented using low-overhead link-layer protocols such as Aurora using multi-gigabit serial transceivers included in the FPGA. BlueLink [9] demonstrated that a new protocol using high-speed serial links has a better area-performance characteristics than trying to implement existing network protocols. In a rack-level deployment, the reliability of such links are so high, that the constraints for the design of a network are different from larger scale networks. Many distributed FPGA computing systems have their FPGA nodes networked over such high-speed serial links [10], [2], [11], [12]. These systems have demonstrated very high network performance by organizing the nodes into various topologies optimized for their target applications. Some have developed meta language compilers that generate application-specific network logic with features such as flow control from separate network specifications [13].

Most distributed FPGA computing systems using high-speed serial links as the network fabric often provide link and network level interfaces, but they rarely provide higher-level functionality such as end-to-end flow control. We have discovered during our own FPGA cluster construction that an FPGA developer who is attempting to implement an accelerated application on our cluster had trouble writing deadlock-free code without per-virtual channel flow end-to-end flow control.

## III. Network Architecture

The overall architecture of the network components can be seen in Figure 1. The network architecture can be divided largely into two parts, the network layer and the transport layer. The network layer is implemented in the form of the router, and the transport layer is implemented in the endpoints that are chained to the router interface. Flow control is implemented in both layers. Link level flow control is implemented in the network layer so that back pressure can be propagated across the network to ensure there are no packet drops. End-to-end flow control is implemented in the transport layer so packets in the network are always assured to have empty receiving buffers.

The distributed application components communicate with remote nodes using the network *endpoints*. Endpoints expose send and receive interfaces, and behaves like a FIFO, in that it blocks when it cannot safely send any more packets. Many endpoints can be instantiated, resources permitting, and each endpoint can have a different *type*, meaning it can expose send and receive interfaces of different bit widths. Sending a message can be done by calling *send* with the data and destination node ID, and *receive* returns a tuple of data and source node ID.

### A. Network Layer

The network layer implements lossless, in-order routing, which assures that packets always arrive in the order they

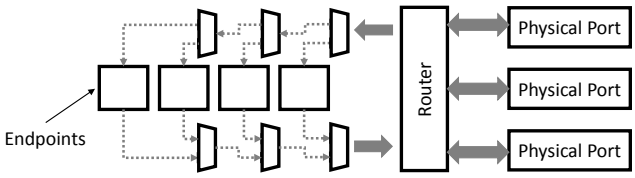

Fig. 1: Network Architecture

were sent. This removes the need for try-resend or reordering functionalities at the transport layer, allowing a much simpler design.

**Router Architecture :** The router logic is deterministic, in that a certain packet type from a certain source node being delivered to a certain destination node will always travel through the same path, even when there are multiple possible paths. Parallelism is achieved by distributing packets from different endpoints to different paths, when there are multiple different paths that the packet can be routed through. This is to ensure in-order delivery of packets to an endpoint, eliminating the need for packet reordering logic and buffer resources at the receiving end. Because this design eliminates network level congestion control, the network may suffer suboptimal performance if one endpoint generates the majority of network traffic.

Each physical port implements a link-layer flow control. It has a large enough buffer to saturate the physical link bandwidth, and assures there are no packet drops when even the data rate exceeds to available physical bandwidth.

**Endpoint Interface :** User endpoints expose two separate interfaces, the user interface and the system interface. The user interface exposes send and receive ports for the application to use to communicate with remote nodes, while the system interface exposes another set of send and receive ports, which are chained together to communicate with the router, as seen in Figure 1. The multiplexers used to chain the endpoints together are designed to have alternating priorities between its two inputs, in order to achieve fairness while maintaining high performance.

The system interface of each endpoint can send a pair of payload and destination node ID to the endpoint chain. The chain logic will augment the packet with the packet type, which is the index of the endpoint in the endpoint chain. The final piece of the packet is the source node ID, which is filled out at the router. The system interface can also receive a pair of payload and source node ID. The router inserts a received packet into the endpoint chain if the destination node ID matches its own, and the chain logic forwards the packet to the correct endpoint according to the packet type.

**Packet Structure :** The network layer manages packet forwarding between the user endpoints, and the physical ports that connect to neighboring nodes. Each packet consists of four fields: source node ID, destination node ID, packet type and payload data. The contents of a packet can be seen in Figure 2. Each packet is broken into multiple fixed-width *beats* when it's sent over the physical link. The control field at the beginning is used to deliver meta-information, to designate the beat as a flow control credit, or to mark the beat as the last one of a packet. The router is oblivious to the existence of multiple network endpoints or virtual channels they represent. The router only deals with routing individual packets, and higher level functions such as virtual channels and end-to-end flow control is implemented by the chain of endpoints.

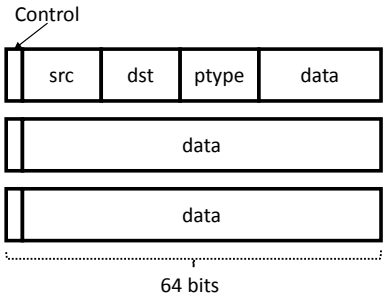

Fig. 2: Packet Structure

### B. Transport Layer

Virtual channels multiplex a single physical network link to provide the logical interface of multiple links. Figure 3 describes the flow of packets in such an environment. For virtual channels to be useful, the channels needs to be logically separate so that they do not interfere with each other. Unless they are safely separated, traffic congestion in one channel can cause the physical link to become congested and cause other channels to block. Our network implements a per-channel end-to-end flow control, so that a sender can only send data onto the network when it is guaranteed that the receiving endpoint has enough buffer space to accommodate it. This assures the physical link and router will never deadlock, which will cause the whole network to stop.

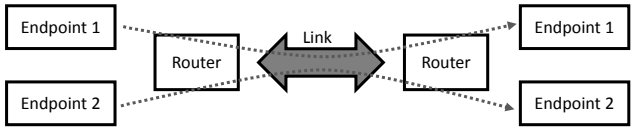

Fig. 3: Packet Flow in Virtual Channels

The transport layer is implemented in individual endpoints, and its design aims to provide a very low latency and efficient memory space usage. Each design can have multiple instantiations of endpoints, parameterized differently. Each endpoint act as a virtual channel entry and exit points. The structure of an endpoint is described in Figure 4. An endpoint performs two major functions: Managing packet transfer between the user application logic and the router layer, and end-to-end flow control. The network also provides an unmanaged endpoint, which does not provide flow control guarantees but provides fastest performance.

**Packet Management :** Packets arriving from end endpoint chain are enqueued into the receive buffer as tuples of source node ID and packet data, and the user logic can dequeue the receive buffer to consume the packet. User logic can send a packet to a virtual channel by inserting a tuple of destination node ID and packet data into the send buffer. Independently

from this, the endpoint internal logic can insert flow control packets into the ack queue to notify remote endpoints of available buffer space, and a multiplexer interleaves packets from the send buffer and ack queue onto the endpoint chain. When a packet arrives from the endpoint chain that is directed at this particular endpoint, it is pushed into the receive queue. The flow control logic ensures there is always available space in the receive queue, and the user logic can receive a pair of source node ID and payload data by dequeuing from the receive queue.

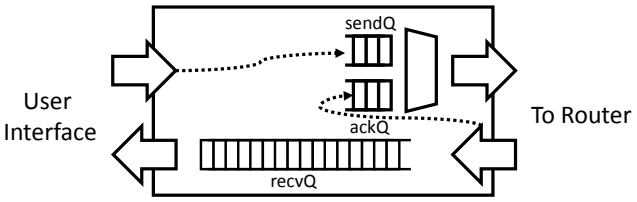


Fig. 4: Endpoint Architecture

**Parameterized Flow Control :** Whenever a packet is received by an endpoint, it checks a table of packets received per source node to determine if it is time to send a flow control credit to the source node. If the send budget of the source node is predicted to have become small enough, it enters a request into the ack queue. A multiplexer that arbitrates the send and ack queues only sends the flow control packet into the network if there is enough space left on the receive buffer, and then marks an amount of space as allocated. By enqueuing flow control packets and sending them as soon as buffer space is ready, the sender can receive send budget updates at a low latency without having to repeatedly send requests.

Determining when is the right time to send a flow control packet is very important in an effective endpoint design. In order to maintain a high bandwidth, the data transfer packets and flow control packets must be overlapped, as seen in the different between Figure 5 (a) and Figure 5 (b). For such an overlap to happen, the flow control packet must be sent a certain amount of time before the send budget of the source node runs out. Ideally, the space allocated per flow control packet (or `Stride`) is large enough, and the packet send offset large enough, that the flow control packet will receive it before its budget runs out. However, it is often not possible to provide a large enough buffer to conservatively accommodate traffic from all nodes in the system.

Under such constraints, each endpoint can have a different flow control configuration that attempts to best suit its usage. For example, for some endpoints the expected traffic pattern may be that most of the data transfer happens between a pair of two nodes. In such a case, it might be effective to have a very low granularity flow control (or a large `Stride`), so that a large buffer is allocated on request, but the total receive buffer may be small. On the other hand, if many nodes are expected to send data to one node, a fine granularity (or smaller `Stride`) flow control and a large buffer may be required for performance. If the endpoint is used for low-bandwidth traffic such as commands, the buffer size and granularity can be set to a small value. To enable such control, endpoints are initialized with the parameters described in Table I.

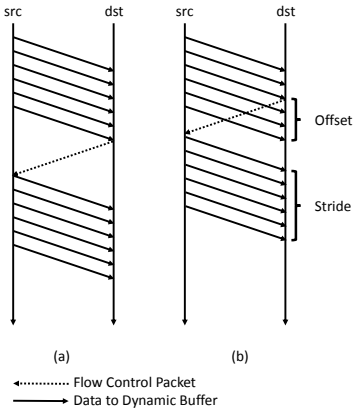


Fig. 5: Packet Timing Comparison

Initially, all nodes start with a small send budget (`initBudget`) to all remote nodes, and therefore the actual size of the receive packet buffer is $initBudget \times nodeCount$ slots larger than the `BufferSize` parameter. When the first packet arrives, space in the receive buffer is allocated to the source node and a flow control packet is sent. When a node decides it will not send more packets for some time and will rather stop receiving flow control packets, it can set a bit in the packet control field that tells the receiving endpoint to only allocate `initBudget` for the next stride, so the send budget state can go back to its initial state. Yielding buffers like this achieves better buffer usage when many nodes are going to send data to one node. The receiving endpoint can also choose to periodically allocate only `initBudget` size buffers, in an attempt to more fairly allocate buffer space across source nodes.

**Unmanaged Endpoint :** The network infrastructure also provides an unmanaged endpoint which does not implement a transport layer protocol. The unmanaged endpoint can sustain the highest bandwidth and lowest latency as it does not check if the receiving endpoint has available buffers before sending packets. It should be used very carefully, since if the arriving data is not always immediately consumed and dequeued from the receiving buffer, it may cause the entire network to block.

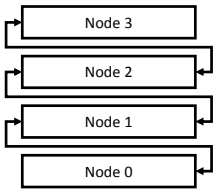


Fig. 6: Prototype Topology

| Parameter | Description |
|---|---|
| BufferSize | Size of the total allocated buffer space |
| FlowOffset | Offset of flow control packet transmission |
| FlowStride | Number of packets each flow control credit represents |

TABLE I: Endpoint Parameters

## IV. Implementation Details

We have implemented a prototype of the network described using a cluster machines. Each node in the cluster consists of one Intel Xeon-based server, Xilinx VC707 FPGA development board, and a network expansion card. Each VC707 development board was augmented with a network expansion card that plugged into the FMC expansion port, which pinned out the four GTX multi-gigabit serial transceivers assigned to the high pin count FMC port into four SATA ports. SATA crossover cables were used to connect the network expansion cards. Two lanes, or two SATA cables were grouped together to form a channel. In the prototype system the cluster was networked into a one-dimensional bar topology, depicted in Figure 6. Since the VC707 board has two FMC ports, each node can have a fan out of up to 4 channels, which can be used to implement other various kinds of topologies. Any direct network with a fan-out per node of 4 or less can be implemented. Some examples of such topologies are depicted in Figure 7.

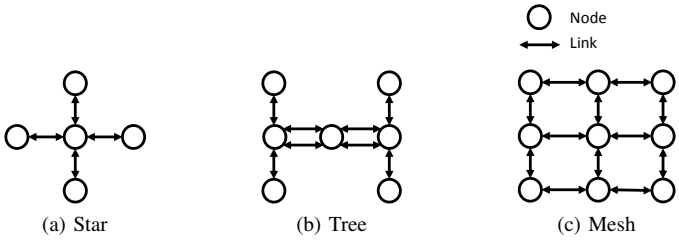


Fig. 7: Any Network Topology With Less Than 4 Fan-Out is Possible

The link latency of an aurora link based on the GTX multi-gigabit transceiver was measured to be around 0.48us, which translates to about 75 cycles on the 6.4ns user clock. The link layer flow control was implemented with a conservative size of 200 beats for the round trip delay.

### A. FPGA Resource Utilization

We have measured the FPGA resource utilization of our network using a simple setup with two endpoints: one high speed endpoint with larger flow control stride and buffer size (Stride of 200 and buffer size of 1024 packets), and one small endpoint with smaller buffers. The endpoint row in the table below described the larger endpoint. The router component includes the chaining logic used to link the endpoints to it. The user logic was clocked at 125MHz.

| Component | LUTS | RAMB36 |
|---|---|---|
| Aurora Link | 4843 | 36 |
| Router | 3743 | 0 |
| Endpoint ($\times 2$) | 753 | 3 |
| Total | 10092 | 42 |
| Virtex 7 Percentage | (3%) | (4%) |

TABLE II: FPGA Resource Utilization

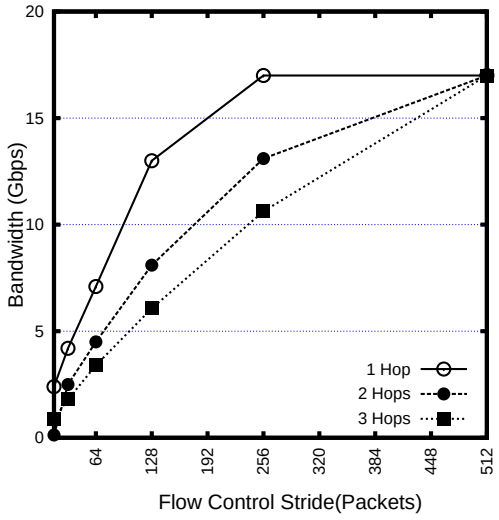


Fig. 8: Network Bandwidth With Variable Network Distance

## V. Performance Evaluation

### A. Bandwidth Evaluation

We demonstrate that the performance of the network does not suffer from the addition of transport-layer network functions. We measured the bandwidth of the network under various configurations, such as transporting data to nodes of variable distance, with multiple endpoints with various buffer size and flow control credits. We show that our network can usually achieve a bandwidth of 17Gb/s, which is 85% of the maximum physical link bandwidth. This performance is reasonable considering the packet header and flow control overhead.

**Single Endpoint Over Multiple Hops :** We measured the bandwidth of the network implementation by using a single endpoint to send a large amount of data to a remote node and measure the elapsed time. Data was sent to nodes that are varying hops away. We also measured the performance of the network with various flow control stride settings. Larger Stride settings mean a larger buffer size is required. Flow control offset was set to be half of the stride length. The results can be seen in Figure 8.

When the flow control stride was small, performance of the network was lower when going over a longer network distance. This is because the round trip latency over multiple hops is longer than the time it takes to deplete the send buffer, resulting in idle cycles when no data can safely be sent over the network. With the low network latency of the serial links, maximum bandwidth over 3 network hops could be achieved using a single endpoint when the flow control stride is over 512 packets large.

**Multiple Endpoints Over Multiple Hops :** Since most interesting distributed FPGA applications will have more than one network endpoint, maximum network performance can be achieved even when a single endpoint's stride length is large enough. We measured the aggregate network bandwidth of a varying number of endpoints sending data to a node three network hops away. We also measured the performance with

varying flow control stride lengths. Flow control offset was set to be half of the stride length. The results can be seen in Figure 9. It shows that a collection of smaller sized endpoints can saturate the network by filling in each others' idle cycles.

**Buffer Size and Flow Control Offset :** Endpoints can be characterized not only by its flow control stride length, but also by the flow control offset and buffer size parameters. Having a larger buffer size means being able to reserve space to allocate buffers for more remote nodes. The same amount of buffer space can also be allocated to a different number of nodes with different flow control strides. The size of the flow control offset also effects endpoint characteristics. A larger offset generally requires a larger buffer as space for a new stride has to be allocated while the receive queue is not quite emptied of the previous stride. But setting a smaller offset has the risk of incurring idle time by delivering a flow control packet too late.

We measured the effect of such parameters by having three nodes send a stream of packets to the same remote node. We tested three scenarios, two had the same total buffer size organized into different organizations, and one had a smaller buffer. Table III describe the three scenarios. In the first scenario, the three source nodes will be contending to be scheduled into the two possible stride slots, where in the latter two scenarios they will be contending for one 64 packet stride slot.

| Setting | Description |
|---|---|
| 32*2+16 | Buffer has space for two 32 packet strides, with offset of 16 |
| 64*1+16 | Buffer has space for one 64 packet stride, with offset of 16 |
| 64*1+8 | Buffer has space for one 64 packet stride, with offset of 8 |

TABLE III: Flow Control Parameters

The results can be seen in Figure 10. It shows that even with the same buffer size, having a larger stride with is beneficial to a small buffer configuration. The difference is pronounced enough that even reducing buffer usage further by making the offset smaller results in a better performance

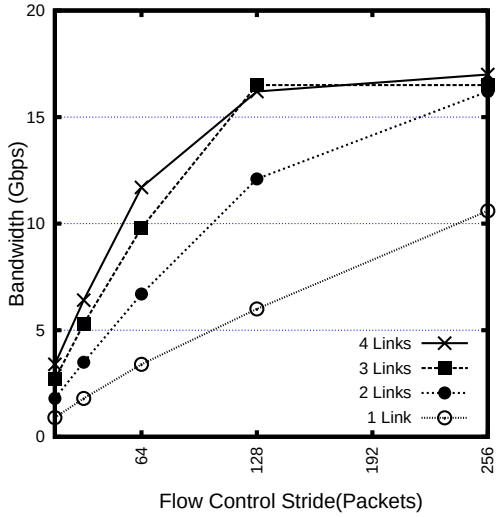


Fig. 9: Network Bandwidth With Variable Number of Channels

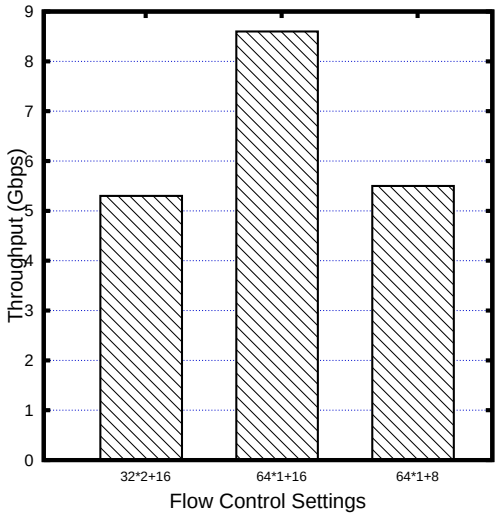


Fig. 10: Network Bandwidth With Different Flow Control Settings

compared to the configuration with smaller stride lengths. These results suggest that for smaller endpoints with small buffer sizes, buffer space can most effectively be used by using it for larger strides, while using a relatively small offset. For large endpoints attempting to exert the most amount of bandwidth, a larger offset may be required to fill in the time between the flow control packet latency.

### *B. Latency Evaluation*

Network latency was measured by measuring the round-trip latency by sending a packet to nodes of varying distances, where the user logic immediately sends the packet back to the original sender. The results can be seen in Figure 11. We show a consistent latency of less than 0.5us per hop.

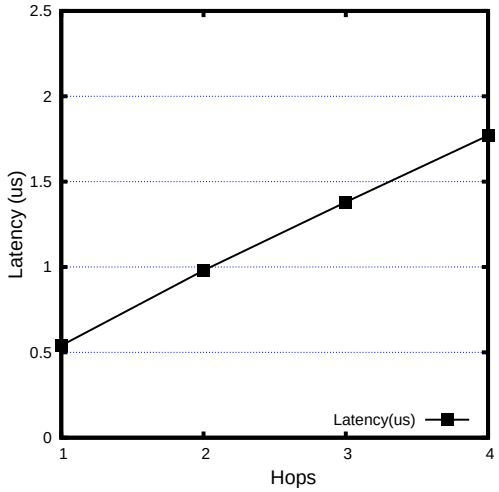


Fig. 11: Network Latency Per Hop

## VI. Conclusion

An efficient and high-performance network is essential for development of an effective distributed FPGA computing platform. Due to high engineering cost and the scarce on-chip memory resource, many existing inter-FPGA network implementations do not provide transport-level network functionality such as end-to-end flow control and virtual channels. Because such systems depend on the user application developer to implement such features and write safe applications, it becomes difficult to create complex distributed FPGA applications which are also deadlock-safe and high performance.

In this paper, we have presented our design of a parameterized, low overhead transport-layer network that provides useful features such as virtual channels and end-to-end flow control. Our network takes advantage of the high reliability of the high-speed serial links, which are integrated in the FPGA fabric, to implement a lossless in-order network layer, which allowed us to simplify the transport layer and use less FPGA resources. The design of the transport layer is parameterized, so that the developer can choose to use less resources while meeting the performance requirements of the individual endpoint. Our prototype implementation demonstrated a high performance in an FPGA cluster setting. We predict that our network will accelerate future research of distributed FPGA applications.

## References


[1] S. Moore, P. Fox, S. Marsh, A. Markettos, and A. Mujumdar, "Bluehive - a field-programable custom computing machine for extreme-scale real-time neural network simulation," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, April 2012, pp. 133–140.

[2] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest, "Maxwell - a 64 fpga supercomputer," in *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, Aug 2007, pp. 287–294.

[3] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, G. Jan, G. Michael, H. S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Yi, and X. D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 13–24, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2678373.2665678

[4] K. H. Tsoi and W. Luk, "Axel: A heterogeneous cluster with fpgas and gpus," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 115–124. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723134

[5] S.-W. Jun, M. Liu, K. E. Fleming, and Arvind, "Scalable multi-access flash store for big data analytics," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 55–64. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554789

[6] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István, "Achieving 10gbps line-rate key-value stores with fpgas," in *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*. Berkeley, CA: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/Blott

[7] I. T. Association, *Infiniband*, 2014 (Accessed November 18, 2014). [Online]. Available: http://www.infinibandta.org

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851192

[9] A. Theodore Markettos, P. Fox, S. Moore, and A. Moore, "Interconnect for commodity fpga clusters: Standardized or customized?" in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014, pp. 1–8.

[10] T. Bunker and S. Swanson, "Latency-optimized networks for clustering fpgas," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 129–136.

[11] A. Patel, C. Madill, M. Saldana, C. Comis, R. Pomes, and P. Chow, "A scalable fpga-based multiprocessor," in *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, April 2006, pp. 111–120.

[12] R. Sass, W. Kritikos, A. Schmidt, S. Beeravolu, and P. Beeraka, "Reconfigurable computing cluster (rcc) project: Investigating the feasibility of fpga-based petascale computing," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, April 2007, pp. 127–140.

[13] K. E. Fleming, M. Adler, M. Pellauer, A. Parashar, A. Mithal, and J. Emer, "Leveraging latency-insensitivity to ease multiple fpga design," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 175–184. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145725