Scaling the NetFPGA switch using Aurora over SATA

Ajithkumar Thamarakuzhi, John A. Chandy
Department of Electrical & Computer Engineering
University of Connecticut, Storrs, CT USA
{ajt06010, chandy}@engr.uconn.edu

ABSTRACT

NetFPGA is a powerful platform for building high performance networking systems. The reconfigurable nature of the Field Programmable Gate Array (FPGA) makes it more suitable for certain network applications. The current NetFPGA platform supports four Ethernet ports and it can be configured as a four-port Ethernet router or switch. In this paper, we describe the implementation of an 8-port Ethernet switch using two NetFPGA boards which are interconnected by SATA cables. We use the Xilinx Aurora protocol for the serial data flow through the SATA cables. We also introduce an interconnection method using SATA for scaling the NetFPGA switch. We also demonstrate the effectiveness of the 8-port NetFPGA switch using throughput and packet loss measurements under different traffic conditions.

General Terms

Switch, NetFPGA, SATA, Aurora

1. INTRODUCTION

The NetFPGA platform helps researchers, teachers and students in designing high performance networking hardware, and its reconfigurable nature makes the redesigning process cheaper [1]. Researchers have already built several applications on top of the NetFPGA platform and a number of projects are in progress [5]. A modular design methodology available for the router architecture helps researchers to add new modules easily [6]. The NetFPGA platform support four gigabit Ethernet interfaces and source code is available to use it as a 4-port gigabit router or a 4-port gigabit Ethernet switch. The NetFPGA platform also includes two interfaces with Serial ATA (SATA) connectors that enable multiple NetFPGA boards in a system to exchange traffic directly without use of the PCI bus. In this paper, we demonstrate, how, along with the Ethernet ports, the SATA ports can also be used for interconnection in order to scale the NetFPGA switch or router. The available source code does not include the interface modules required for packet transmission through SATA. We have developed interfacing modules which are required for the packet transmission through SATA and added it to the existing source code for the gigabit Ethernet switch.

We are making use of the Aurora protocol provided by Xilinx, for serial communication through SATA [3]. Aurora is a LogiCORE IP designed by Xilinx to enable easy implementation of the FPGA RocketIO transceivers while providing a light-weight user interface on top of which designers can build a serial link. It is a scalable, lightweight, link-layer protocol for high-speed serial communication. It also supports full duplex operation and flow control. SATA with Aurora can give the maximum throughput if we drive the RocketIO transceivers using a differential clock network [4]. The maximum throughput available with a differential clock network is 3.125 Gb/s which is more than three times that of a gigabit Ethernet port. Unfortunately, the current implementation of the NetFPGA does not provide a differential clock pair, but instead provides a single ended clock generator of 125MHz. Using the Aurora Protocol we can transmit two bytes in a single clock pulse. Thus, the maximum possible data transmission rate through SATA in the present system is 2Gb/s which is still twice that of gigabit Ethernet. Each NetFPGA platform support two SATA port. We have implemented an 8-port gigabit Ethernet switch by interconnecting two NetFPGA boards using two SATA cables where each SATA connection support a data rate of 2Gb/s.

The remainder of the paper is organized as follows. In Section 2 we briefly describe the architecture of the 8-port Ethernet switch. In Section 3 we describe the Aurora protocol and the interfacing modules which generates and receives signals that are used in the Aurora modules. In Section 4 we describe another switching architecture called 2-Dilated flattened butterfly which can be used for scaling NetFPGA switch by making use of the SATA interconnection. In Section 5 we present the implementation results and Section 6 is the conclusion.

2. 8-PORT SWITCH ARCHITECTURE

The 8-port gigabit Ethernet switch is constructed using two NetFPGA boards. Two boards are interconnected using two SATA cables, where the SATA master port of each board is connected to the SATA slave port of the other board as shown in Figure 1. Since each board has four gigabit Ethernet ports the maximum incoming Ethernet traffic of each NetFPGA board is 4Gb/s. Each SATA port supports a data rate of 2Gb/s and it also supports full duplex operation. Thus, the cross-sectional bandwidth between the boards in this 8-port gigabit Ethernet switch matches the input bandwidth, meaning all the incoming packets can be transmitted between boards at line rate even in the worst case traffic.

The NetFPGA research group has already provided the source code for a 4-port gigabit Ethernet Switch. It is a five stage pipeline structure where each module communicates using a simple packet-based synchronous FIFO push interface which

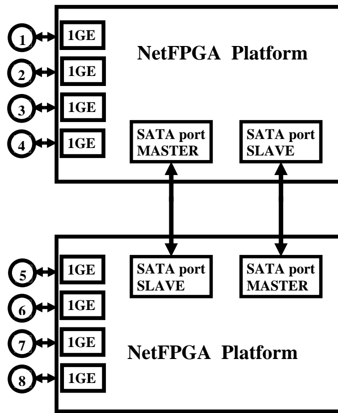

Figure 1: 8-port switch architecture

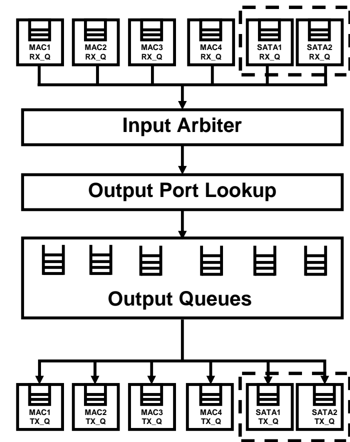

Figure 2: NetFPGA switch architecture

makes it easy to add additional modules to the structure for the purpose of packet processing. The user data path is 64 bits wide and it is driven by a 125 Mhz clock. Therefore, the switch can handle a maximum throughput of 8 Gb/s. In addition to the four Ethernet MAC interfaces provided by the original NetFPGA switch code, we have added two SATA interfaces and the overall structure of the modified gigabit Ethernet switch hardware is shown in Figure 2.

The RX queues accept data from the Ethernet MAC core or Aurora core and convert it to the format required for the user data path. This module also generates an 8-bit control word and the module header. This 64-bit header is prepended to the begining of each packet and the 8-bit control word is generated for each 64 bit word. The module header contains the length of the packet in bytes, the source port as a binary-encoded number and the packet length in 64-bit words. This module header is named the IOQ module header. The Input Arbiter selects an RX queue in which packets are available and pushes a packet into the Output Port Lookup module without modifying the module header. Based on the desired destination port, the Output Port Lookup module decides which output port the packet should be transmitted to, and this information is added as a one-hot-encoded number to the IOQ module header. The Output Queues module looks at the IOQ module header and decides the output queue according to the output port information and it stores the packet into the selected output queue. The output queues are in SRAM and are not in the FPGA. The packet length from the IOQ module header is used to store the packet efficiently. The stored packets in the output queue are removed and pushed into the corresponding destination TX queue. The TX queue module changes the data format according to the requirements of the core (Mac or Aurora) and pushes it to the respective core inputs. The TX queue module also removes the IOQ module header from each packet before transmission and it generates all the control input signals for the core.

3. AURORA AND INTERFACING MODULES

Aurora is a scalable, lightweight, link-layer protocol for high-speed serial communication, developed by Xilinx. We have used LogiCORE Aurora v3.0 for our implementation. The Aurora core can be customized to suit a wide variety of requirements using the CORE Generator software provided by Xilinx.

3.1 Customizing Aurora core

One of the important parameters that we should select while generating the core is $LaneWidth$ which decides the number of bytes transmitted during the period of one clock cycle. Aurora supports $LaneWidth$ of 2 and 4. If we set the $LaneWidth$ to 4, the user clock (USER_CLK) should be half of the reference clock (REFCLK), whereas with a $LaneWidth$ of 2, the frequency of USER_CLK and REFCLK should be the same. So, the effective data rate will be same for both cases. Moreover, the clock generation for a $LaneWidth$ of 4 is complicated than for a $LaneWidth$ of 2. So we have selected the default $LaneWidth$ value of 2. There are two types of data path interfaces used for the core which are *Framing* and *Streaming*. The *Streaming* interface is a simple word-based interface with a data valid signal to stream data through the Aurora channel whereas the *Framing* interface is a Local Link interface that allows encapsulation of data frames of any length. Since in our application we have to transmit packets through the SATA link, we selected *Framing* interface. The Aurora core supports two data flow modes, *Simplex* and *Duplex*. We have selected *Duplex* mode because we need to transmit data in both directions. The Aurora core also supports several clock inputs to drive the RocketIO transceivers. BREFCLK and BREFCLK2 are low-jitter differential clock networks that can support line rates up to 3.125 Gb/s. However, the NetFPGA platform does not support differential clock networks, and as a result we have selected the REFCLK, which is a reference clock input for low rate serial connections. The Aurora REFCLK frequency was set to 125Mhz to match the NetFPGA clock - thus giving an Aurora/SATA line rate of 2 Gb/s (125 MHz * 2 byte lane * 8 bits). The Aurora core also supports $UserFlowControl$ and $NativeFlowControl$, but we are not making use of either of these flow controls.

3.2 Clock interface for Aurora

Aurora cores require a low jitter reference clock for generating and recovering high-speed serial clocks in the RocketIO transceivers. We can use either the differential clock pair BREFCLK or the single ended clock REFCLK as the reference clock. Since the differential clock pair is not available in the NetFPGA platform, we have used *gtx_clk*, which is the common TX clock of 125MHz, which we can see in the top level module of the source code, as the source for the REFCLK. Two-byte lane Aurora cores also use a single clock to synchronize all signals between the core and the user application called USER_CLK. All logic that connects to the core must be driven by USER_CLK, which in turn must be the output of a global clock buffer (BUFG). The way in which we generated reference clock and USER_CLK from *gtx_clk* is shown in Figure 3.

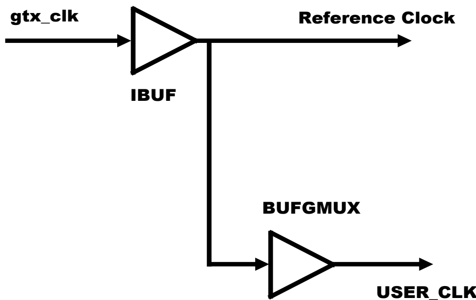

Figure 3: Clocking for 2-Byte Aurora core

3.3 Aurora core and Interfacing modules

The top-level interfacing unit of the Aurora core, which can be directly connected to the other modules of the project is named as the Aurora Local Link Interface. The Aurora Local Link Interface, with LocalLink-compliant ports for TX and RX data is shown in Figure 4 [4].

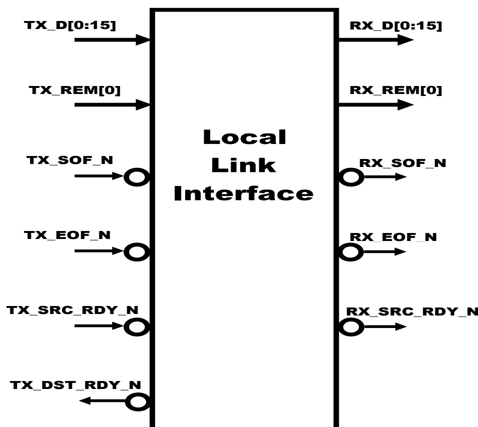

Figure 4: Aurora Core Framing Interface [4]

3.3.1 Transmitting

Table 1 lists port descriptions for Local Link TX data ports. The timing diagram for a simple data transfer at the Aurora transmitting side is shown in Figure 5 [4].

Name	Description
TX_D[0:15]	Outgoing data.
TX_DST_RDY_N	Asserted (Low) during clock edges when signals from the source will be accepted. Deasserted (High) on clock edges when signals from the source will be ignored.
TX_EOF_N	Signals the end of the frame.
TX_REM[0]	Specifies the number of valid bytes in the last data beat; valid only while TX_EOF_N is asserted.
TX_SOF_N	Signals the start of the outgoing channel frame (active-Low).
TX_SRC_RDY_N	Asserted (Low) when LocalLink signals from the source are valid. Deasserted (High) when LocalLink control signals or data from the source should be ignored.

Table 1: LocalLink User I/O Ports (TX)

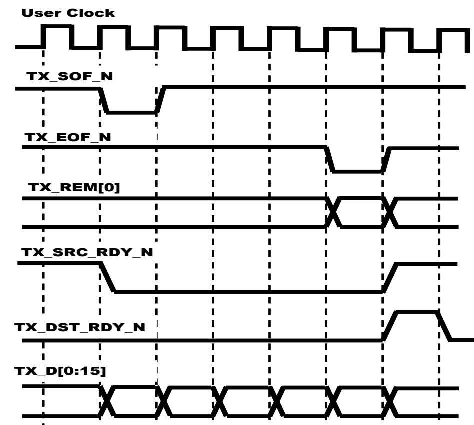

Figure 5: Data transfer in Aurora transmitter [4]

All the signals required to feed the Aurora Local Link transmitter are provided by the module SATA TX_Q, which is shown in Figure 6. The 64-bit data and the 8-bit control, which are coming from the output queue, are used to generate the 72-bit Arranged Data. This 72-bit Arranged Data is stored in a FIFO in the SATA TX_Q module. The data and control bits are arranged in such a way that, the FIFO should output 2 bytes of data and the corresponding two control bits in a single FIFO output read cycle. The writing data width of the FIFO is 72 and the reading data width is 18. The *in_wr* signal, which is asserted whenever the incoming data are valid, is used as the write enable signal for the FIFO. The SATA TX_Q module also generates a signal *in_rdy* which is asserted when ever this module is ready to accept data from Output Queue module. All the other control signals required for the FIFO and for the Aurora

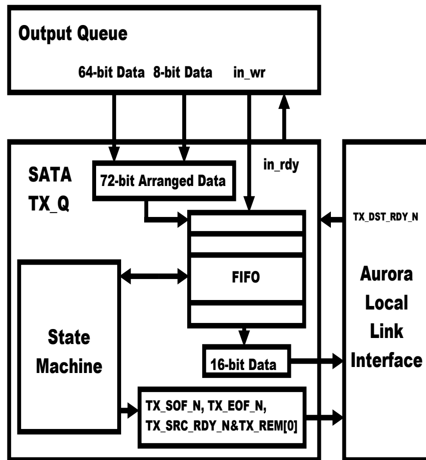

Figure 6: SATA TX_Q module

Local Link Interface transmitter are generated by a State Machine in the SATA TX_Q module. A state machine controls the reading process of the FIFO by generating FIFO read enable signal. Sixteen bits of data are extracted from the 18-bit read data and it is given to the TX_D port of Aurora transmitter. TX_DST_RDY_N is an incoming signal which is asserted by Aurora core and the Aurora core will accept the data only when this signal is asserted. So the state machine should assert FIFO read enable only after checking the TX_DST_RDY_N signal for successful transmission of data. State Machine generates TX_SOF_N, TX_EOF_N and TX_SRC_RDY_N signals by making use of 8-bit control and in_wr signals from Output Queue module. Since Aurora transmits two bytes per clock, it is essential to indicate which of these bytes is the last byte of the frame. This information is passed through the signal TX_REM[0].

In our initial design we tried transmitting frames without deliberately adding any idle cycles between each frame in the Aurora transmitting side. Then, we observed that the switch stopped functioning after few seconds. We first presumed that the internal queues on the NetFPGA were filling up, so we implemented the flow control feature in the Aurora core, and translated that flow control to IP flow control. However, this did not solve the problem, and so we deactivated flow control and attempted to find the cause of the queue overflows. After further study, we observed that the signal which indicates the output queue empty status was indicating that the output queue was empty even though it was not empty. Thus, the *remove_pkt* module in the Output Queue stage was not removing packets from the output queue. During the course of the investigation, we realized that the Aurora frames were being transmitted without any interframe gap, something that Ethernet requires. We, therefore, had the state machine introduce 12 idle clocks between each Aurora frame to serve as an interframe gap. With this slight modification the switch started functioning properly.

The Aurora core supports a feature called clock compensation that allows up to ±100 ppm difference in the reference clock frequencies used on each side of an Aurora channel. This feature is used in systems where a separate reference clock source is used for each device connected by the channel. Since each NetFPGA platform uses its own reference clock we have to make use of this clock compensation feature. A standard clock compensation module is generated with the Aurora core. To perform Aurora-compliant clock compensation, DO_CC must be asserted for several cycles during every clock compensation period. The duration of the DO_CC assertion and the length of time between assertions is determined based on the width of the RocketIO data interface. For a lane width of 2, DO_CC is asserted for 6 USER_CLK cycles after each 5000 USER_CLK cycles.

3.3.2 *Receiving*

Table 2 lists port descriptions for Local Link RX data ports. All the signals coming from the Aurora Local Link receiver, which are shown in Figure 7 [4], are given to the module SATA RX_Q, which is shown in Figure 8.

Name	Description
RX_D[0:15]	Incoming data.
RX_EOF_N	Signals the end of the incoming frame (active-Low).
RX_REM[0]	Specifies the number of valid bytes in the last data beat; valid only when RX_EOF_N is asserted.
RX_SOF_N	Signals the start of the incoming frame (active-Low).
RX_SRC_RDY_N	Asserted (Low) when data and control signals from an Aurora core are valid.

Table 2: LocalLink User I/O Ports (RX)

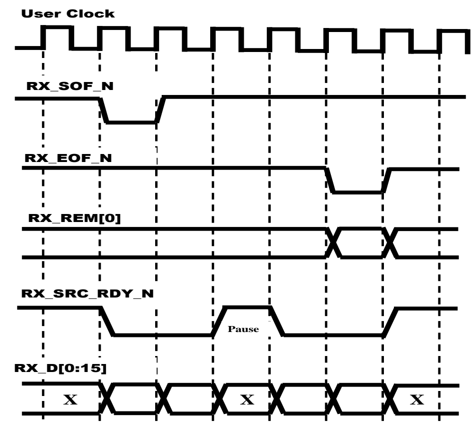

Figure 7: Data transfer in Aurora receiver [4]

The 16-bit data, which is coming from port RX_D[0:15] and corresponding two control bits generated in the SATA RX_Q are written in a FIFO. The writing data width of the FIFO is 18 and the reading data width is 72. We know that the user data path of the NetFPGA switch is 64 bits wide and so in each read cycle, the FIFO should output 64-bits of data as well as the corresponding 8-bit control signal which are given to the Input Arbiter as shown in Figure 8. All the operations in the SATA RX_Q module are controlled and synchronized

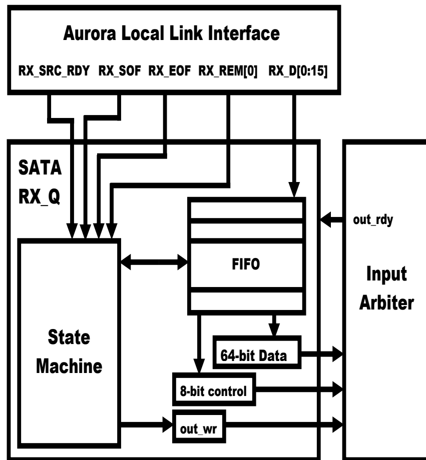

Figure 8: SATA RX_Q module

by state machines. Both the FIFO read enable and FIFO write enable signals are generated by state machines. The FIFO read enable is asserted only when *out_rdy* signal is asserted which is an incoming signal coming from Input Arbiter. *out_rdy* signal is asserted only when Input Arbiter is ready to accept data. RX_SRC_RDY_N is an incoming signal coming from Aurora core which indicates whether the received data is valid or not. This signal is similar to the Ethernet MAC core's *emacclientrxdvld* signal that is asserted at the beginning of each frame and is deasserted only after the last byte of the frame. *emacclientrxdvld* is not deasserted during the transmission of a frame. However, in the Aurora core, RX_SRC_RDY_N can be deasserted during the transmission of a frame because Aurora will stop the transmission for a few clock cycles whenever DO_CC is asserted for clock compensation. Therefore, the signals RX_SOF_N and RX_EOF_N are the only way to identify the starting and ending of a frame. State machine should assert FIFO write enable only when RX_SRC_RDY_N is asserted. Since the Aurora receiver delivers 2 bytes for each clock cycle, State machine should make use of RX_REM[0] to find out the last byte of the frame. The SATA RX_Q module also generates a header which carries information like word length, byte length and the number of input port. This 64-bit header is written at the beginning of each frame. SATA RX_Q also generates 8-bit control signal which carries useful information like starting of a frame, ending of a frame and the position of the header, for each 64-bit data which is passed to the Input Arbiter.

4. 2-DILATED FLATTENED BUTTERFLY

The SATA/Aurora interconnect that we have described allows us to expand the NetFPGA switch to a 8-port nonblocking switch using 2 NetFPGA boards. In this section, we briefly introduce a switching architecture using NetFPGA boards interconnected by SATA connections to build larger-scale switching networks. We have developed a switching architecture called the 2-Dilated flattened butterfly switch architecture [8], which is basically derived from the flattened butterfly structure [7]. A 2-Dilated flattened butterfly is constructed from a flattened butterfly by replacing all the interconnecting channels between switches with channels that have double bandwidth or by duplicating each channel. The 2-Dilated flattened butterfly is a nonblocking switching topology and is well suited for scaling NetFPGA switches using SATA connections. The nonblocking property of 2-Dilated flattened butterfly is proved in [8]. The advantage of using a 2-Dilated flattened butterfly topology is that we can implement a nonblocking switching system with fewer NetFPGA boards compared to other nonblocking switching topologies like Clos. The implementation of a 16-port 2-ary 2-Dilated flattened butterfly topology using NetFPGA is shown in Figure 9.

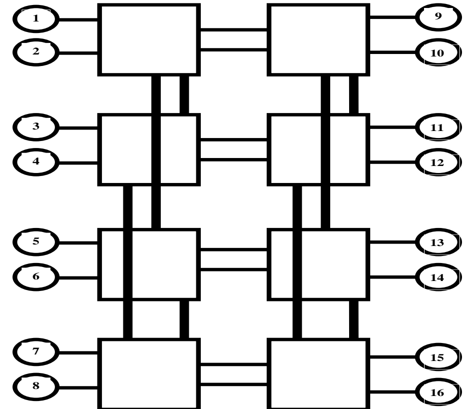

Figure 9: 16-port 2-Dilated flattened butterfly switch

In Figure 9, each box represents a NetFPGA board, each horizontal line represents an Ethernet connection and each vertical line represents a SATA connection. As we can see in Figure 9, we need only 8 NetFPGA boards to implement a 16-port switch whereas the corresponding Clos network would require 12 NetFPGA boards. We can also notice that the average number of intermediate switches in a 2-Dilated flattened butterfly will be always lesser than corresponding Clos structure and therefore the 2-Dilated flattened butterfly implementation will have lesser latency compared to corresponding Clos implementation. Thus, a 2-Dilated flattened butterfly implementation outperforms the corresponding Clos implementation in terms of cost as well as latency. A 2-Dilated flattened butterfly topology can be used to scale the switch to any larger size. Scaling the 2-Dilated flattened butterfly topology to larger networks is similar to that required for the flattened butterfly structure, as explained in [7].

5. PERFORMANCE RESULTS

We have implemented the 8-port gigabit Ethernet switch as shown in Figure 1. We have observed the throughput performance of this network using IPERF [2], which is a network testing tool that can create TCP and UDP data streams and measure the throughput of a network that is carrying them. We have used the UDP data stream for the throughput measurement because in UDP mode we can specify the input data rate and we will get the exact statistics of packet loss. Throughput measurements are done for different traffic patterns as described in Table 3. We are

using 8 separate end-terminal machines and each machine is connected to corresponding ethernet port of the NetFPGA board. Packets are generated by IPERF in which we can vary the packet size. A packet size of 1300 bytes is used for the performance measurement. Each traffic pattern represents pairs of machines in the switch. All the machines are numbered as in Figure 1. In all cases, the source terminal is transmitting a UDP data stream with a rate of 1Gb/s. The average throughput is taken for all the traffic patterns.

Set1: 4-port NetFPGA switch(1 board)	
Pattern A	**(1,3),(2,4)**
Pattern B	**(1,2),(3,4)**
Pattern C	**(1,4),(2,3)**
Set2: 4-port NetFPGA switch(2 board)	
Pattern A	**(1,5),(4,8)**
Pattern B	**(1,2),(3,4)**
Pattern C	**(1,8),(2,7)**
Set3: 8-port NetFPGA switch(2 board)	
Pattern A	**(1,5),(2,6),(3,7),(4,8)**
Pattern B	**(1,6),(2,8),(3,5),(4,7)**
Pattern C	**(1,8),(2,7),(3,6),(4,5)**

Table 3: Traffic pattern

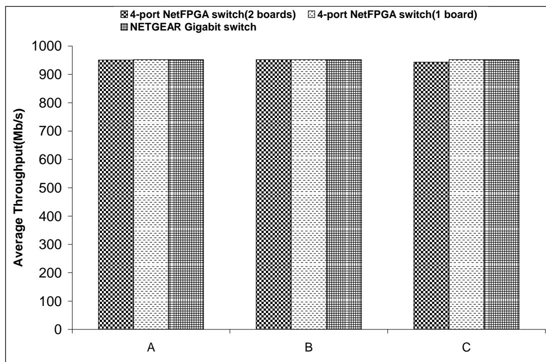

Figure 10: Throughput comparison of 4-port NetFPGA switches

Set1 of Table 3 shows the traffic patterns that we have used for testing the throughput of 4-port NetFPGA switch using a single NetFPGA board. Set2 contains the traffic patterns that we have used to measure the performance of a 4-port netFPGA switch which uses 2 boards. In this case, we are selecting four ports of the 8-port NetFPGA switch such that some of the pairs span the SATA connection. Set3 denotes the traffic patterns that we are using to measure the performance of the 8-port NetFPGA switch.

The throughput of a 4-port NetFPGA switch using a single board is compared with the throughput of 4-port NetFPGA switch using two boards connected using SATA and NETGEAR Gigabit switch and is shown in the Figure 10 for corresponding traffic patterns set1 and set2. For comparison

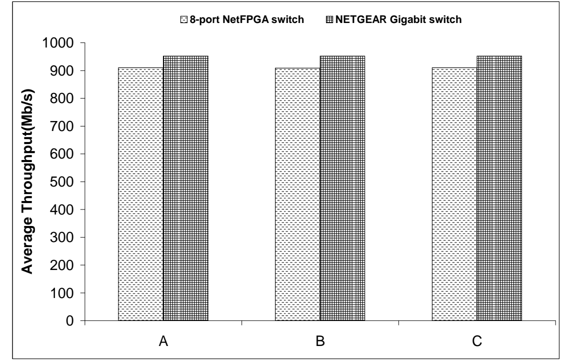

Figure 11: Throughput comparison of 8-port switches

we have selected the first 4 ports of a 16-port NETGEAR Gigabit switch (GS516T) and used set1 traffic patterns. In the pattern B of set2, all the ports are connected to a single NetFPGA. Because of this, the B pattern throughput of 4-port NetFPGA switch using two boards is same to that of other two switches. According to our static routing, packets destined for port5 and port6 are routed through one SATA port and for the port7 and port8 are routed through the other SATA port. Therefore, for the pattern A of set2, the maximum data rate through each SATA port will be 1 Gb/s. From the Figure 10, we can observe that the throughput of all the switches are same for pattern A also. If we use the pattern$C$ of set2, the traffic from two source terminal share the same SATA connection. Though the maximum data rate through each SATA port is 2Gb/s and should support the traffic of two data streams, we see that there is 0.89% packet loss in the 4-port NetFPGA switch using two boards for pattern C.

Figure 11 shows the throughput comparison between 8-port NetFPGA switch and a NETGEAR Gigabit switch. We have selected the first 8 ports of the 16-port NETGEAR Gigabit switch and used the same traffic patterns (set3) for both the switches. We can see that for all the traffic patterns, the throughput of the 8-port NetFPGA switch is close to the maximum achievable throughput. For the 8-port NetFPGA switch, packet loss is around 4% for all traffic patterns.

We have also observed the percentage of packet loss for different load conditions for 8-port NetFPGA switch and 4-port NetFPGA switch using 2 boards for the traffic pattern C of set3 and set2 respectively. We have observed the percentage of packet loss by varying all end-terminal data rate from 0Gb/s to 1Gb/s. This percentage of packet loss is shown in Figure 12. We can observe that the 4-port NetFPGA switch using 2 boards shows 0% packet loss up to 900 Mb/s and for 1Gb/s the packet loss is 0.89% where as 8-port NetFPGA switch using 2 boards shows 0% packet loss up to 800 Mb/s and for 1Gb/s the packet loss is 4.3%

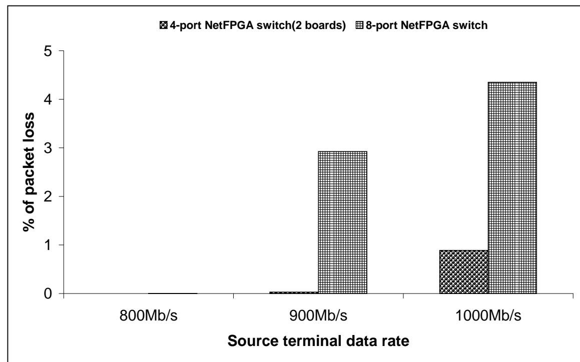

Figure 12: Packet loss for different load conditions

We believe that the small amount of packet loss that we observe in Figure 12 and as compared to the NETGEAR switch is caused by the interframe gap that was introduced between frames. We are still trying to determine why the output queue empty signal is asserted even though the queue is not empty with out using Inter Frame Gap. In the current switch design we are using static routing for routing packets. We are also doing research for designing better routing algorithms.

6. CONCLUSION

In this paper we have described the implementation of an 8-port Ethernet switch. We have used two SATA interfaces to do the communication between two NetFPGA platforms. In order to do this communication, we have made use of the Aurora protocol which is developed by Xilinx. In this paper, we explain how we modified the existing NetFPGA Ethernet switch to incorporate two Aurora cores. We have also introduced a switching topology called 2-Dilated Flattened butterfly which is suitable for scaling NetFPGA switches with reduced cost and latency. Finally, we have demonstrated the throughput and packet loss of our 8-port Ethernet switch implementation. Our experiments reveals the fact that SATA interconnections are suitable for scaling NetFPGA switches to larger sizes with lesser cost and better performance.

7. REFERENCES

[1] http://www.netfpga.org.

[2] OpenSS7 IPERF utility installation and reference manual. http://www.openss7.org/iperf_manual.html.

[3] Xilinx: Aurora link-layer protocol. http://www.xilinx.com/products/design_resources/conn_central/grouping/aurora.htm.

[4] LogiCore IP Aurora 8B/10B v5.1. http://www.xilinx.com/support/documentation/ipcommunicationnetwork_serialinterface_aurora8b10b.htm/aurora_8b10b_ug353-2.pdf, Dec. 2009. User Guide.

[5] J.Naous, D.Erickson, G.A.Covington, G.Appenzeller, and N.McKeown. Implementing an openflow switch on the netfpga platform. In *Symposium On Architecture For Networking And Communications Systems*, 2008.

[6] J.W.Lockwood, N.McKeown, G.Watson, G.Gibb, P.Hartke, J.Naous, R.Raghuraman, and J.Luo. Netfpga: An open platform for gigabit-rate network switching and routing. In *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, 2007.

[7] J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: A cost-efficient topology for high-radix networks. In *In Proc. of the International Symposium on Computer Architecture (ISCA)*, pages 126–137, June 2007.

[8] A. Thamarakuzhi and J. A. Chandy. 2-dilated flattened butterfly: A nonblocking switching network. In *International Conference on High Performance Switching and Routing (HPSR 2010)*, June 2010.