A Protocol for Realtime Switched Communication in FPGA Clusters

Richard D. Anderson
Computer Science and Engineering, Box 9637
Mississippi State University
Mississippi State, MS 39762
rda62@msstate.edu

Yoginder S. Dandass*
Computer Science and Engineering, Box 9637
Mississippi State University
Mississippi State, MS 39762
yogi@cse.msstate.edu

Abstract - *Field programmable gate array (FPGA) devices typically have limited resources. This means that networks of FPGAs are required for implementing large-scale applications. Use of realtime communication channels can be used for reducing handshaking overhead in order to create high-performance networks. This paper describes a switched, real-time, link-level protocol and its implementation using Virtex-4 multigigabit transceivers. A prototype 4Gbps implementation of the protocol shows a per-hop latency and jitter of 310ns and 12.5ns, respectively, when endpoints run at a clock frequency of 100MHz. The prototype is also used to demonstrate the implementation of a jitter-free distributed global realtime clock that can be used for scheduling communication and computation of applications implemented using this cluster of FPGAs.*

Keywords: Cluster, FPGA, Real-time, Multi-gigabit transceivers, communication protocol.

1 Introduction

Reconfigurable computing, using field programmable gate arrays (FPGAs), is becoming increasingly popular in high-performance computing applications such as in digital signal processing (DSP) and in bioinformatics. FPGA-based applications exploit the massive parallelism that can be implemented in logic. However, the level of parallelism that can be implemented is often restricted by the area (i.e., resources) available on the chip. Furthermore, large FPGAs are significantly more expensive as compared to smaller chips. Therefore, in many applications it is more cost effective to utilize a number of smaller interconnected FPGAs than using one (or more) large FPGAs.

We had previously introduced the design of a real-time link-level communication protocol and a prototype point-to-point implementation using multi-gigabit transceivers (MGTs) found on Virtex-4 FPGAs [2]. Although this previous implementation had good real-time characteristics, our prototype was limited to point-to-point communication between exactly two FPGAs. This paper extends the ideas introduced previously by developing and characterizing a switch that enables routing of frames in real-time between several FPGAs. Furthermore, our enhanced design improves the real-time performance of the protocol by reducing the jitter of the real-time distributed clock. Reducing this jitter is important for reducing the overheads in a real-time network. The reminder of the paper is organized as follows: Section 2 discusses the motivation and background for this work. Section 3 provides a summary introduction to the protocol. Section 4 describes the experimental setup and results, and Section 5 concludes with a discussion the results and future work.

*Corresponding Author

2 Background and Motivation

Many FPGA-based applications exhibit real-time characteristics because their applications' logic are controlled by finite state machines (FSMs) having well-defined timing properties. When designing a communication protocol for such applications distributed over a cluster of FPGAs, the handshaking required for flow control between devices can be eliminated. This results in a high-performance network in which communication is scheduled according to a globally distributed real-time clock.

Using the protocol presented here, applications typically utilize zero-sided communication as opposed to two-sided communication found in most software applications. In two sided communication, processes at both ends of the communication channel execute communication operations in order to transfer data. In zero-sided communication, neither endpoint process issue explicit data transfer operations. Instead, any data that is available in a buffer at the transmit endpoint is delivered to a buffer at the receiving endpoint according to a predefined schedule.

Modern FPGAs such as Xilinx's Virtex [3], [4] and Altera's Stratix [5] families of FPGAs support several multigigabit transceivers that can be used for chip-to-chip or board-to-board serial communication. Xilinx has designed a general-purpose high-speed serial communication protocol, Aurora, for enabling simplex and full-duplex link-level communication in applications using on-chip MGTs [6]. Aurora supports both frame-oriented and streaming communication with native and application controlled flow control. Xilinx's Aurora implementations are feature-rich and robust. However, we chose to design our own protocol in order to reduce latencies and overheads to the largest extent possible.

A few research projects have also developed and implemented customized point-to-point high-speed communication channels for interconnecting FPGAs using MGTs [1], [7], [8]. The Aurora protocol is also relatively efficient but is a point-to-point protocol. Because of the limited number of MGTs available at each FPGA, creating large clusters requires the use of interconnect switches. Aurora does not have switching capability, and therefore, any frame routing capability needs to be developed as part of the application.

3 Protocol Description

We have adapted and enhanced the implementation of the protocol described in detail in [1], and therefore, do not repeat all the protocol details here in order to conserve space. However, important information on the specification of routing and its implementation in our new version is described in detail. Our protocol is a link-level serial protocol that uses 8b/10b encoding [9] to exchange control and data messages between application endpoint and switch FPGAs. Only frame-oriented communication is supported and endpoint FPGAs are connected to each other using switches over point-to-point physical links. In its present form, the protocol allows a maximum of eight intermediate switches between any two nodes and each switch can have at most eight connections to other endpoints and switches.

Just as in [1], our protocol uses source routing (i.e., the application endpoint determines the routing of each frame and includes the routing information at the beginning of each frame). The switches use the routing information in order to perform cut-through (or worm-hole) routing. In cut-through routing, the switch forwards data words as soon as they are received and the outbound channel is available. This significantly decreases latencies as compared with traditional store-and-forward routing.

3.1 Frame Layout and Routing

Data and control bytes are transmitted in 4-byte words. Frames are delimited by 4-byte *start-of-frame* (SoF) and *end-of-frame* (EoF) control words. When no data is available for transmission, the MGT sends the *idle* control word. Idle control words are continuously transmitted between frames in order to maintain the synchronization between a pair of communicating serial transceivers (i.e., the receiver extracts the clock embedded in the transitions between 0 and 1 signals in the physical data stream being sent by the transmitter). Therefore, the receiver logic on one endpoint FPGA is operating at the same frequency as the transmitter logic on the switch FPGA.

It is also possible for the transmitter to run out of data to send before the frame is completely transmitted. In this case, the transmitter also sends idle control words within the frame; these idle control words are discarded by the receiver. Note: if the real-time schedule is constructed carefully, there should be no need to transmit idle control words within a frame. However, this feature was designed into our protocol to provide some safety against data buffer underflows in case there are small differences in frequencies between the transmitter's, receiver's, and application's clock domains.

The SoF word is immediately followed by a single source routing word. Source routing information is organized into 8 nybbles (i.e., 4-bit sequences) such that each nybble represents a single hop (i.e., an outgoing transmission from a switch). The most significant bit of a hop specification indicates whether the remaining three bits are valid. The remaining three bits specify the outgoing port on the switch that is to be used to forward the frame. Essentially, the value in the three address bits is added to the incoming port number to derive the outgoing port number. The complete route for a frame is specified by the sequence of nybbles starting with the least significant nybble in the word. When the source routing word is received by a switch, it scans the word starting with the least significant nybble looking for a valid hop specification (invalid specifications are skipped). When a switch forwards the frame, it consumes the corresponding nybble by clearing the valid bit in the nybble (see [1] for more details).

For data integrity checks, an optional 4-byte error-checking word (*e.g.* checksum or CRC) can be inserted into the frame immediately before the EoF. The switch can be configured to detect this word, verify the integrity of the frame payload, and invalidate an erroneous frame.

3.2 Implementation Issues

We have developed a prototype implementation of the link layer logic on the Virtex-4 FX family of FPGAs. In our implementation, each Virtex-4 MGT has different clocks for the transmitter and the receiver. An on-board reference oscillator is used for clocking the transmitter whereas the receiver's clock is derived from the received signal. Furthermore, the application on the endpoints has its own independent clock. The MGT can simplify implementation clocking by performing synchronization internally such that the link layer needs to only be aware of one clock domain. However, internal clock synchronization requires the data and control words to traverse through a fairly lengthy internal path [1]. A significant portion of this path can be bypassed by operating the MGT in a "low latency" mode with the tradeoff that the link layer must perform synchronization by itself.

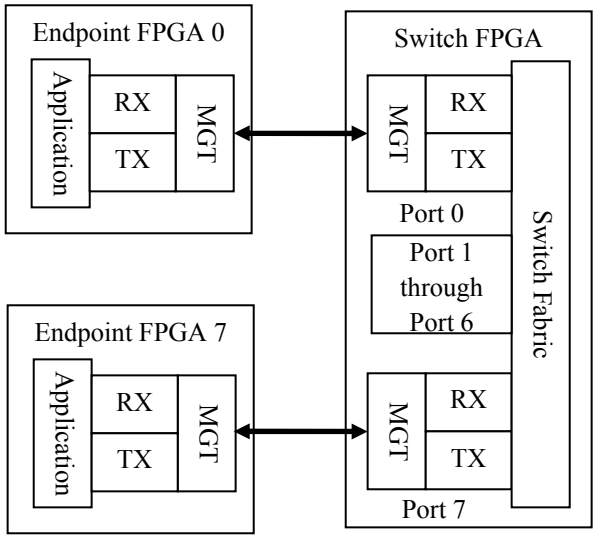

Figure 1: Implementation Architecture [1]

In our prototype implementation, the switch uses eight receivers and eight transmitters. All of the eight transmitters derive their clocks from a common 100MHz reference clock. However, because each of the eight receivers derives its frequency from the transmitter it is connected to, each of the eight receivers in the switch operates in an independent 100MHz clock domain. Similarly, on the endpoints, the application, transmitter, and receiver each operate in independent 100MHz clock domains.

We use FIFOs with independent read and write clocks whenever data needs to cross clock domains. Xilinx supports efficient construction of such FIFOs using block RAMs (BRAMs) in Virtex 4 FPGAs [3]. These FIFOs, however, each add five to six clock cycles of latency in the data path (the variance is caused by the phase relationship of the clocks at FIFO's read and write endpoints. Therefore, significant latency can result when data traverses several clock domains. For example, when data is transmitted from the application block in *Endpoint FPGA 0* to the application block in *Endpoint FPGA 7*, it must traverse three FIFOs (Application-to-TX at Endpoint FPGA 0, $RX_{port\ 0}$-to-$TX_{port\ 7}$ at the switch, and RX-to-Application in FPGA 7.

3.3 Implementation Architecture

Figure 1 depicts the architecture of the link layer. It shows one endpoint FPGA connected to the switch FPGA. The host FPGA uses one MGT to connect to the switch. The switch FPGA uses eight MGTs to connect to other switches or endpoint FPGAs. The internal signal routing resources of the switch FPGA are used to create a fully connected point-to-point topology between the switch's MGTs.

The switch does not provide buffering services greater than the depth of the FIFOs between the ports; it is the application's responsibility to create frames and transmission schedules such that it does not overflow the FIFO. We chose a fully connected topology to simplify switching and to avoid loss of data due to frame *collisions*. A collision occurs at a switch when two (or more) overlapping frames are received that need to be sent out on the same port simultaneously. In the event of a collision, our switch receives frames destined for the same port and buffers them in the separate FIFOs connecting the receiver ports with the transmitter port. It then uses a fixed priority scheme to select the order in which the frames will be forwarded. Although the switch is capable of handling a few arbitrary frame collisions, it is not designed to handle constant simultaneous frame transmissions that are destined for the same port. Therefore, it is also the responsibility of the application to create schedules that avoid collisions.

3.4 Switching and Selection Unit Architecture

Frames received by the switch are handled by two hardware modules, the switching unit and the selection unit. There are eight switching units and eight selection units (i.e., one pair per MGT in the switch). Each switching unit is connected to all eight selection units. This results in a total of 64 connections (including loopback connections). Buffering FIFOs are placed between the switching and selection units. All switching units and selection units are independent of each other and can operate in parallel, therefore, it is possible

for all switching units to route frames and for all selection units to forward data concurrently.

The switching unit is responsible for decoding a received frame's source-routing word and then forwarding the frame to the corresponding destination. The switching unit functions as a one-to-many de-multiplexer, where the decoded routing nybble acts as the selector that connects a single input line from the receiver to one of many output lines. A secondary function of the switching unit is to verify the integrity of the frame data using a verification method such as the cyclic redundancy check (CRC). After decoding the route, the switching unit begins computing the CRC for the incoming frame payload. The frame CRC is stored in the 4-byte word at the end of the payload and before the EoF. Once the switching unit sees the EoF, it will verify its computed CRC with the frame CRC and mark the frame as invalid if there is not a match. To ensure that the switching unit properly detects the source-routing word and the frame CRC, the endpoint applications must follow that rule that no idle control words should be placed between the SoF and the source-routing word, and between the CRC and EoF. These word pairs must not be separated.

The switching unit supports zero-sided communication and does not throttle incoming frames; it consumes and forwards each 4-byte data word received by the MGT receiver every clock cycle. However, since the routing information follows SoF in our frame layout, the SoF is buffered in a two stage pipeline to give the switching unit time to decode, update, and register the routing nybble before forwarding the frame; the pipeline is also used when verifying the CRC.

Figure 2 shows a frame being routed by the switching unit. The data flow across the switching unit begins with the receipt of the SoF from the receiver at clock cycle 0. At clock cycle 1, the SoF has been registered in pipeline stage one and the 4-byte routing information is being presented by the receiver. At this point, SoF is assigned to be registered at pipeline stage two and the least-significant valid routing nybble is decoded and assigned to be registered on the next clock cycle. The updated 4-byte routing information, where the decoded nybble's valid bit is set to zero, is assigned to be registered in pipeline stage one. At clock cycle 2, the registered SoF in pipeline stage two is written to the FIFO input port pointed to by the registered nybble, the updated routing information is assigned to pipeline stage two, and the first 4-byte payload is assigned to pipeline stage one.

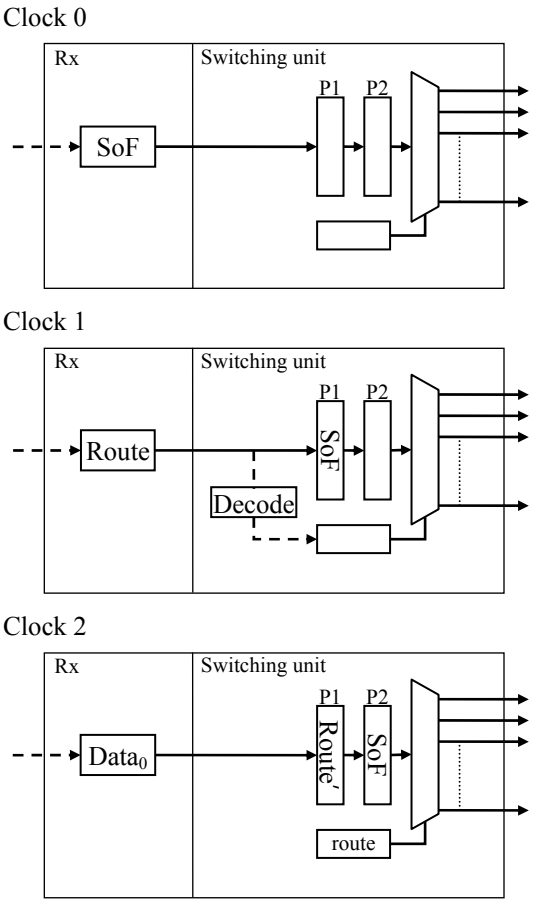

Figure 2: Switching Unit Data Flow

From this point on, each incoming 4-byte payload traverses the pipeline and is inserted into the FIFO until the EoF is presented by the receiver at clock cycle *n*, where *n* is the (*size-of-the-frame* – 1), in words. At this point, if the switch is configured to do error checking, the frame's 4-byte CRC has already been registered in pipeline stage one. Upon detecting the EoF, the switching unit compares its computed CRC with the frame's CRC. If the two do not match, the switching unit marks the frame CRC as a bad CRC and sets its value to be registered in pipeline stage two at clock cycle *n*+1; if the CRCs match, the frame CRC is just shifted to pipeline stage two. At clock cycle *n*+1, EoF has been registered in pipeline stage one and the frame CRC is written to the FIFO from pipeline stage two. At clock cycle n+2, the EoF has been registered in pipeline stage two and written to the FIFO. The registered routing nybble is flagged to be reset on the next clock cycle.

The selection unit (or selector) implements a many-to-one connection. It receives data from eight FIFOs and forwards their data to the transmitter by selecting a non-empty FIFO

and linking its data-out port and read enable signal to the transmitter. The transmitter can then assert the read enable of the FIFO when it is ready to send the contained frame.

A typical selection process begins when at least one *FIFO-is-empty* indicator is deasserted (*e.g.* transitions from high to low); all eight *FIFO-is-empty* indicators are observed in parallel. For this implementation, we have used a simple round-robin priority scheme to select between competing non-empty FIFOs. The FIFO with priority is checked for data (*i.e.* not empty) at this time. If it is not empty, the FIFO is selected over all other non-empty FIFOs and will be linked to the transmitter. The selection unit will also link the *FIFO-is-empty* indicator of the selected FIFO to the transmitter to signal that there is data available to be transmitted. The selector then sets the next sequential FIFO as the new FIFO with priority. For example, if $FIFO_3$ has priority and is not empty, the selection unit will link it to the transmitter and then set $FIFO_4$ as the next FIFO with priority. In the event that the FIFO with priority is empty, the next sequential non-empty FIFO is selected. For example, if $FIFO_3$ is empty and the next non-empty FIFOs are $FIFO_6$ and $FIFO_7$, $FIFO_6$ will be linked to the transmitter and $FIFO_7$ is set as the new priority. When the round robin reaches $FIFO_7$, it will wrap around to $FIFO_0$, the next sequential FIFO.

4 Experimental Setup & Results

A number of experiments were conducted in order to investigate the feasibility of successfully utilizing the protocol and its implementation in a realtime cluster. We have implemented the endpoint and switch logic on two different Virtex-4 FX-100-FF1152 platforms. The HTG-V4-PCIE board from HitechGlobal is used for the endpoints and the ML423 from Xilinx is used as a switch. The HTG-V4-PCIE boards are connected to the ML423 board via SMA connectors. All the MGTs in the virtex-4 FX-100-FF1152 are brought out to SMA connectors on the ML423, making it an ideal platform for implementing the switch.

In our current implementation, the MGTs are configured to operate at a bit rate of 4Gbps resulting in a peak data rate of 400MBps in each direction. The MGT-to-logic interface is implemented to be 32-bit words, therefore, the receiver and transmitter logic operate at 100MHz. The application endpoint is also configured to operate at 100MHz. However, the receiver, transmitter, and application all operate in separate 100MHz clock domains.

4.1 Latency and Jitter

For the first experiment, we connected two endpoints, *A* and *B*, to the switch and measured the latency and the jitter of our implementation using a *ping-pong* test. In the ping-pong test, endpoint *A* sends a frame across the switch to endpoint *B*. Endpoint *B* sends the recieved frame back to endpoint *A*, through the switch, after *B* finishes receiving the frame from *A*. This test is repeated 40,000,000 times and the minimum, and maximum time taken to complete one round-trip is recorded over all of the iterations.

Table 1: Roundtrip Time (100 MHz Cycles).

Payload Size	Min	Max
1	122	126
2	124	128
4	128	132
8	136	140
16	152	156
32	184	188
64	248	252
128	376	380
256	632	636
512	1144	1148

Table 1 shows the results from the ping-pong test. The payload of the frame includes the error-checking word but excludes the source-routing word, SoF and EoF control words. The maximum round-trip jitter is five clock cycles and is the result of crossing clock domains. The total number of clock cycles required to transmit an *n*-word payload is given by equation (1) below:

$$T = 3 + n + T_O, \qquad (1)$$

where n represents the clock cycles required to transmit n payload words and T_O is the overhead. The 3 in the equation represents one cycle each required to transmit the source-routing word, the SoF and EoF. Applying equation (1) to the minimum entry in each row of Table 1 shows that the minimum roundtrip overhead is 114 clock cycles (*i.e.*, one-way overhead is 57 cycles).

When there are no frame collisions, the switch has a latency of 11 clock cycles beginning at the receipt of SoF by the receiver and the transmission of the SoF by the transmitter. This latency is the result of the two stage switching pipeline, one cycle selection, delays internal to the FIFOs, and one cycle each for writing to and reading from the FIFOs.

For the second experiment, we connected four endpoints, *A*, *B*, *C*, and *D* to the switch and measured the latency and jitter

using a four-point-traversal test. In the traversal test, the source-routing word is set to include the following eight hops: hops one and two from endpoint A to endpoint B via the switch; hops three and four from B to C; hops five and six from C to D; and hops seven and eight from D to A, completing the traversal. The switch decodes and updates the source-routing word for each pair of hops. We repeated the traversal test 40,000,000 times and recorded the minimum, and maximum time taken to complete each round-trip over all of the iterations.

Table 2: Roundtrip Time (100 MHz Cycles).

Payload Size	Min	Max
1	244	252
2	249	257
4	257	265
8	272	280
16	304	312
32	368	376
64	496	504
128	753	761
256	1264	1272
512	2291	2298

Table 2 shows the results from the four-point-traversal test. The frame payload includes the error-checking word and excludes the source-routing word, SoF and EoF. The maximum round trip jitter is nine cycles (or average time ± 4 clock cycles). The results show that the minimum round trip overhead is 228 cycles (*i.e.* four times the one-way overhead of 57 cycles). This is consistent with the observations in table 1.

4.2 Clock Synchronization

For the third experiment, we conducted a clock synchronization test between the switch and the endpoints. A distributed global clock is maintained at the switch and at all of the endpoints in the form of 64-bit counters that increment every clock cycle (at a frequency of 100MHz). The switch is responsible for synchronizing the global clock, and therefore, broadcasts its counter value to all endpoints periodically. At the time appointed by the schedule, each transmitter on the switch, retrieves the global clock counter value and transmits this value in a special frame indicted by a special SoF* control word. This frame consists of three words, the SoF*, and the two data words carrying the clock value (there is no CRC and EoF associated with this frame). Upon receiving the global clock from the switch, each receiver immediately updates its own global clock counter (instead of incrementing it). Note that the clock value does not traverse any FIFOs in this process.

At the receiver, the clock is updated using the expression in equation (2):

$$C_{t+1} = C_t + ((R + \lambda - C_t) / 2), \quad (2)$$

where C_t is the current counter value, R is the received counter value, and λ is computed as follows:

$$\lambda = 3 + 19, \quad (3)$$

where λ is the expected one-way latency for transferring the synchronization frame. In (3), 3 represents the total size, in words, of the global clock synchronization frame. The 19 represents the maximum travel time for the clock frame (from the transmitter at the switch to the receiver at the endpoint). The one-way latency computed by equation (3) corresponds to a single switch network with eight connected endpoints. In a multi-switch network, where one or more switches are connected to the broadcasting switch or in succession with the switch, global clock synchronization can become complex.

The additional switches and overall network design can increase latency and introduce additional overheads. Furthermore, our global broadcast procedure may not be the most optimal synchronization method for a multi-switch network. There are other global clock synchronization methods (e.g. NTP and IEEE1588) that use multiple synchronization packages and a three-way handshake to update the global clock [10]-[13]. However, in a single switch FPGA network with predictable communication and synchronization stages due to a predefined schedule, a single frame one-way broadcast is suitable because we can easily compute the one-way latency using equations (1) and (3). For multi-switch networks, additional experimentation with our broadcast synchronization method will be required in order to characterize λ.

Our experiment resulted in a zero jitter in the expected global clock value computed using equation (2) at update frequencies 1Hz, 25Hz, 50Hz, and 100Hz. This is an improvement in the clock jitter over our previous implementation [1] in which we achieved a global clock jitter of 3 and 13 cycles for updates at 100Hz and 1Hz, respectively. Our zero jitter observation is a result of having eliminated the FIFOs in clock value update paths. At the switch, the global clock runs in a clock domain independent of all the transmitters. We implement a FIFO-free mechanism using registers to cross the clock value from the global clock's domain to each of the transmitters' domains (the register

essentially holds the global clock value for several cycles just as the scheduler is indicating to the transmitters that it is time to broadcast the global clock value – this enables the transmitter to read a stable global clock value).

5 Conclusions

The results reported in this paper demonstrate the improvements in performance of our implementation of the real-time link-level communication protocol as compared with our previous design and implementation [1]. Our prototype and experiments now report results that include a switch and exhibit a significant reduction in jitter of the distributed global clock. Our implementation supports a switch, with up to eight endpoint FPGAs.

In future work we plan to optimize our global clock synchronization method for a multi-switch FPGA network. We are also investigating the design and implementation of advanced schedulers for computational and communication activities in the network of FPGAs.

6 Acknowledgements

This work was supported in part by the following National Science Foundation grants: EPS-0903787, EPS-1006983, DUE-0513057.

7 References

[1] R. D. Anderson and Y. S. Dandass, "A Protocol for Realtime Communication for FPGA Clusters using Multigigabit Trancevers," *22nd International Conference on Parallel and Distributed Computing and Communication Systems (PDCCS 2009)*, Louisville, KY, USA, September 24-26, 2009

[2] Xilinx, *Virtex-4 RocketIO Multi-Gigabit Transceiver User Guide UG076 v4.1*, November 2, 2008, http://www.xilinx.com/support/documentation/user_guides/ug076.pdf, accessed March 29, 2011.

[3] Xilinx, *Virtex-4 FPGA User Guide UG070 v2.6*, December 1, 2008, http://www.xilinx.com/support/documentation/user_guides/ug070.pdf, accessed March 29, 2011.

[4] Xilinx, *Virtex-5 FPGA User Guide UG190 v5.3*, May 17, 2010, http://www.xilinx.com/support/documentation/user_guides/ug190.pdf, accessed March 29, 2011.

[5] Altera, *Stratix IV Device Family Overview SIV51001-2.4*, June 2009, http://www.altera.com/literature/hb/stratix-iv/stx4_siv51001.pdf, accessed March 29, 2011.

[6] Xilinx, *Aurora 8B/10B v3.1 for Virtex-4 FX FPGA, DS128*, April 24, 2009, http://www.xilinx.com/support/documentation/ip_documentation/virtex_4fx_aurora_8b10b_ds128.pdf, accessed March 29, 2011.

[7] H. Kristian, O. Berge, and P. Häfliger, "High-Speed Serial AER on FPGA," in *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS 2007)*, New Orleans, LA, USA, May 27-30, 2007.

[8] M. Liu, W. Kuehn, Z. Lu., A. Jantsch, S. Yang, T. Perez, and Z. Liu, "Hardware/Software Co-design of a General-Purpose Computation Platform in Particle Physics," in *Proceedings of the 2007 International Conference on Field-Programmable Technology (ICFPT 2007)*, Kitakyushu, Kyushu, Japan, 2007.

[9] A. X. Widmer and P. A. Franaszek, "A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code," *IBM Journal of Research and Development,* 27(5), 1983.

[10] IEEE, "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," IEEE Std 1588-2008, URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4579757

[11] J. Wu and J. Zhang and Y. Ma and M. Xie, "A Low-jitter Distributed Synchronous Clock Using DAC," *16th IEEE-NPSS Real Time Conference (RT' 09)*, Beijing, China, 2009

[12] Y. Kang and J. Wu and M. Xie and Z. Yu, "A New Design for Precision Clock Synchronization Based on FPGA," *16th IEEE-NPSS Real Time Conference (RT' 09)*, Beijing, China, 2009

[13] M. Zhang and S. Shen and Jian Shi and T. Zhang, "Simple Clock Synchronization for Distributed Real-Time Systems," *IEEE International Conference on Industrial Technology (ICIT 2008)*, Chengdu, China, 2008