CHAMELEON SHARED MEMORY:
IMPLEMENTING A THIN ARCHITECTURAL LAYER TO SUPPORT SHARED MEMORY ACROSS MULTIPLE BLADES

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Christopher David Dolen

January 2008

ABSTRACT

Highly integrated SMPs can execute a broad range of workloads, but are expensive and monolithic. It is difficult incrementally to add processing power to a highly integrated SMP, which requires the system to be large enough to handle all possible workloads. This can be unnecessarily expensive and wasteful when executing tasks that do not require such computational power. Clusters are cheap and modular, but cannot execute the same workloads, and are more difficult to manage. Although it is easier incrementally to add more processors to a cluster, communication time between processors is much larger, and running applications with high interprocessor communication is not feasible. Clusters must use message passing instead of shared memory, and managing a large cluster can be difficult, due to the extensive system administration required for many individual systems. We present a system that gives us the best of both worlds: modular and scalable systems that are easy to manage and can execute a broad range of applications.

BIOGRAPHICAL SKETCH

Christopher Dolen began his college experience at The University at Buffalo, State University of New York in the year 2000. He graduated Summa Cum Laude in May of 2005 with a B.S. in Computer Engineering. He matriculated at Cornell University in August 2005. He is currently a second year graduate student in the MS/PhD program, working in the Fusion group led by Professor Sally A. McKee. His research interests include processor architecture, memory systems, and implementing hardware designs via FPGAs.

ACKNOWLEDGMENTS

I would like to acknowledge the contributions of many people who helped me in this project, both from Cornell and IBM TJ Watson. First I would like to thank José Moreira, my mentor at IBM, who had the original idea to provide lightweight hardware support for shared memory to blades, and has offered his help and advice every step of the way. My manager at IBM, Xiaowei Shen, was also there to offer his advice and guide my fellow Cornell student, Catherine Trammell, and me through various problems. I thank Catherine Trammell, who worked on the first half of this project with me: we learned how to use the Aurora protocol and the Xilinx tools. I am greatly appreciative of the help from Chuck Haymes, Ken Inoue, and Scott Lekuch, who were all working on the Chameleon blade. Chuck worked on the DVI controller as well as the DDR1 memory controller. Ken is familiar with running Linux on the blades and wrote the demo code for the two-blade and four-blade rings. Scott worked on the connection between the Cell processor and the LX160 FPGA, as well as the connection between the LX160 and the FX100 FPGA's. Scott, Chuck, and Ken also spent much time helping me debug and answering questions about the Chameleon blade. Dan Kuchta is in charge of the optical daughter card we use for the board-to-board communication; I thank Dan for his help with the use of this card and with integrating optics into the design. I also thank IBM design engineer Mickey Tsao. Mickey worked full time with me for the second half of the project. Together we worked through many problems. He taught me much about system design and worked with me on almost every aspect of the design and implementation of the final multi-blade ring. He also worked on the testing of the DDR1 memory and was involved in the routing mechanisms of the ring design. Lastly, I thank my advisor at Cornell, Sally A. McKee. Sally was ultimately responsible for my chance to work on this project, and throughout the entire process has kept me focused and on track. Her continued guidance and experience has helped me get through this project as a whole.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

COMA: Cache Only Memory Architecture

DCM: Digital Clock Manager

DDR: Double Data Rate

DMA: Direct Memory Access

DVI: Digital Video Interface

DVSM: Distributed Virtual Shared Memory

FPGA: Field Programmable Gate Array

Gbps: Gigabits per second

HDL: Hardware Definition Language

HP: Hewlett Packard

IB: InfiniBand

IBM: International Business Machines

I/O: Input/Output

KB: KiloBytes

MGT: Multi-Gigabit Transceiver

NUMA: Non-Uniform Memory Access

OS: Operating System

PCI-E: PCI Express

PID: Processor ID

PLL: Phase Locked Loop

RGB: Red Green Blue

RX: Receive

SDRAM: Synchronous Dynamic Random Access Memory

SMP: Symmetric MultiProcessor

SRAM: Static Random Access Memory

TX: Transmit

VMMC: Virtual Memory Mapped Communication

XDR: eXtreme Data Rate

1 Introduction

Highly integrated SMPs can execute a broad range of workloads, but are expensive and monolithic. It is difficult incrementally to add processing power to a highly integrated SMP, which requires the system to be large enough to handle all possible workloads. This can be unnecessarily expensive and wasteful when executing tasks that do not require such computational power. Clusters are cheap and modular, but cannot execute the same workloads, and are more difficult to manage. Although it is easier incrementally to add more processors to a cluster, communication time between processors is much larger, and running applications with high interprocessor communication is not feasible. Clusters must use message passing instead of shared memory, and managing a large cluster can be difficult, due to the extensive system administration required for many individual systems. We present a system that gives us the best of both worlds: modular and scalable systems that are easy to manage and can execute a broad range of applications.

The goal of this project is to provide a thin architectural layer supporting shared memory across multiple blades. This allows us to achieve a better performance/price ratio than a tightly integrated SMP (Symmetric Multi-Processor), and enables the migration of low-bandwidth applications from SMP systems to blades. Our solution supports efficient use of remote memory at remote blades due to the lower bandwidth available. The use of blades allows us to create a modular server system of variable size, and the shared memory support allows us to use a hypervisor so there is only one machine image, which helps to simplify system management. This is not an SMP or NUMA (Non-Uniform Memory Access) machine. It has limited bandwidth and acceptable latency. Therefore, it is not a solution for commercial applications with high bandwidth cache coherence communication, but is as effective for applications with lower bandwidth requirements, and therefore a cheaper solution than a highly integrated SMP.

One way for an organization to achieve greater computing power is to add more processors to a system. More processors allow for more parallel work. But workloads change over time, and dynamically increasing the computing power of a high performance server is usually not easy or cheap. SMP systems are available, but it can be difficult to guess what level of computing power future applications will need. Buying a system that is too large is a waste of money, but buying a system that is too small means that it may need to be replaced in the near future. Table 1 shows various SMP systems available from IBM [1] and HP [2]. The system costs are obtained from the Transaction Processing Performance Council [3].

Table 1: Prices of various SMP systems from IBM and HP

Model	Range of Cores	Total System Cost (# of processors)
IBM System p5 520	1 to 2	$243,218 (1)
IBM eServer pSeries 660	2 to 8	$1,632,624 (6)
IBM System p5 570	2 to 16	$4,004,491 (8)
IBM eServer pSeries 690 Turbo	16 to 32	$7,574,961 (16)
IBM System p5 595	16 to 64	$11,967,178 (32)
HP ProLiant DL380	2	$81,177 (1)
HP ProLiant ML370	4	$278,114 (4)
HP Integrity rx5670	1 to 4	$556,853 (4)
HP Integrity rx8620	2 to 16	$1,372,435 (16)
HP Integrity Superdome	2 to 128	$8,397,262 (64)

Table 1 shows many currently available systems. Each can support a varying number of cores. The exact number is dependent upon the particular server model. Smaller systems are cheaper, but if the consumer needs to increase the number of processors, a completely new system must be purchased. Larger systems can hold many more processors, but are

much more expensive. SMP systems are not modular. This is a problem for both the designer and consumer. The designer's problem is that it requires providing multiple models of servers to meet the differing needs of the consumer. The consumer's problem is that it limits the maximum size of the system. On the positive side, SMPs only have one OS image to manage.

On the other end of the spectrum, there exist blade servers, e.g., the IBM BladeCenter [1]. The BladeCenter is a much more modular system; more blades are added when greater computing power is needed. A blade is basically a motherboard that contains all necessary components for a functioning server system. It has memory, processors, storage, and I/O capabilities. These blades are plugged into a chassis, which provides power and cooling. Therefore, blades make it simple to scale out by adding a new blade into an open slot in the blade chassis. With multiple processors per blade, and up to 14 blades per BladeCenter, increasing the computing power of a system is simply a matter of adding more blades. But each of these blades functions independently. They each have their own OS, require individual system administration, and have no shared memory support. This requires parallel applications to use message passing, which can be difficult to program. If there are many OS images to manage, system administration will be more difficult. Multiple operating system images are time consuming and more costly to maintain.

SMP systems are designed to have high bandwidth and low latency between processors for fast communication. But the price for this communication is high, and requires higher processor pin counts, larger buses, and higher energy usage. These factors make SMP systems expensive. Benefits of our system are that it provides the required performance at much lower price.

2 Solution

Our solution is to combine the modularity of a blade server with the ease of management and shared memory capabilities of an SMP system. This system is cheaper than a tightly integrated SMP due to lower bandwidth and higher communication latencies between processors. This is not a solution for every application, but there are many commercial, medical, and scientific applications that can benefit from such a system. The ability to scale at the module level to meet differing needs of applications, with a single OS to manage, and shared memory support for parallel applications is beneficial. This addresses many problems: cost, time to solution, engineering development, and inventory.

The solution starts with the BladeCenter, as this system already involves the modularity desired. The next steps are to add shared memory, and get the benefits of an SMP system. These requirements lead us to choose a special blade called a Chameleon, described in Section 2.1. The specific hardware components on this blade are not essential to the project, but since the Chameleon is available for use, is modular, and has programmable FPGAs for custom designs, it is a good starting point. The communications protocol, Aurora [4], is from Xilinx, and is described in Section 2.2. The bi-directional ring network used to connect multiple blades together is described in Section 2.3.

2.1 The Chameleon Blade

Figure 1 shows a simplified view of the Chameleon blade, which is a prototyping blade developed at the IBM TJ Watson Research Center. It has a single Cell [5] processor, two Xilinx Virtex-4 FPGAs [6], memory, and various I/O capabilities, such as a PCI-E connector and an Infiniband connector.

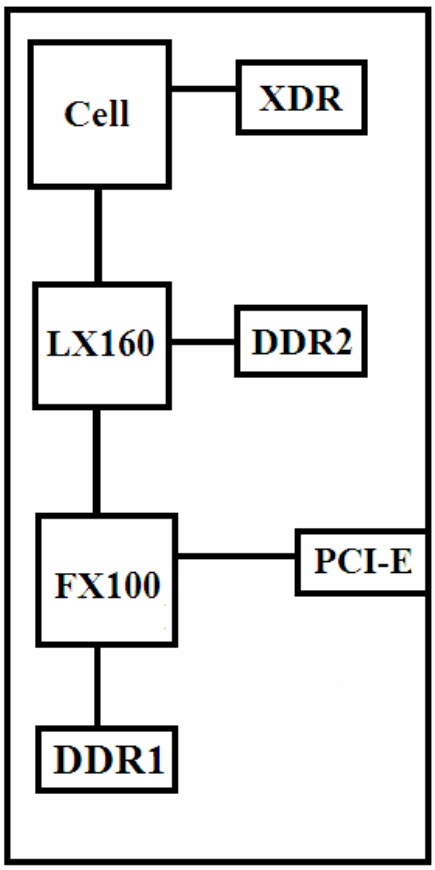

Figure 1: The Chameleon blade

The main components are the Cell processor, which has a connection to XDR (eXtreme Data Rate) DRAM and the Xilinx LX160 FPGA. The LX160 has 960 I/O pins and 152,064 logic cells, and therefore it can hold extensive custom logic. The Xilinx FX100 FPGA has fewer logic cells (94,896) and I/O pins (768), but contains two PowerPC cores. The FX100 has RocketIO Multi-Gigabit Transceivers (MGTs). These MGTs can achieve a baud rate of 6.5 Gbps, and are differential and bidirectional, which makes them useful for board to board communication. The RocketIO MGT-based bus runs from the FX100 to a PCI-Express connector. There are 16 MGTs that run to the PCI-E connector on the board. The FX100 has a connection to DDR1 SDRAM (Double Data Rate Synchronous Dynamic Random Access Memory) on the board, and the LX160 has a connection to a DDR2 DIMM (Dual In-line Memory Module). A DDR1 memory

controller placed into the FX100 accesses up to 128 MB of memory, and a DDR2 memory controller placed into the LX160 can access the DDR2 DIMM.

Although the electrical signals can be carried over wires for board to board communication, we choose an optical transceiver daughter card. Using optical fibers for communication has advantages over using standard copper wires, as described below. This card attaches to the PCI-E connector and converts differential electrical signals to optical signals. There are a total of eight transceivers on the card, allowing us to use up to eight MGTs. Conceptually, a single transceiver looks like the image in Figure 2.

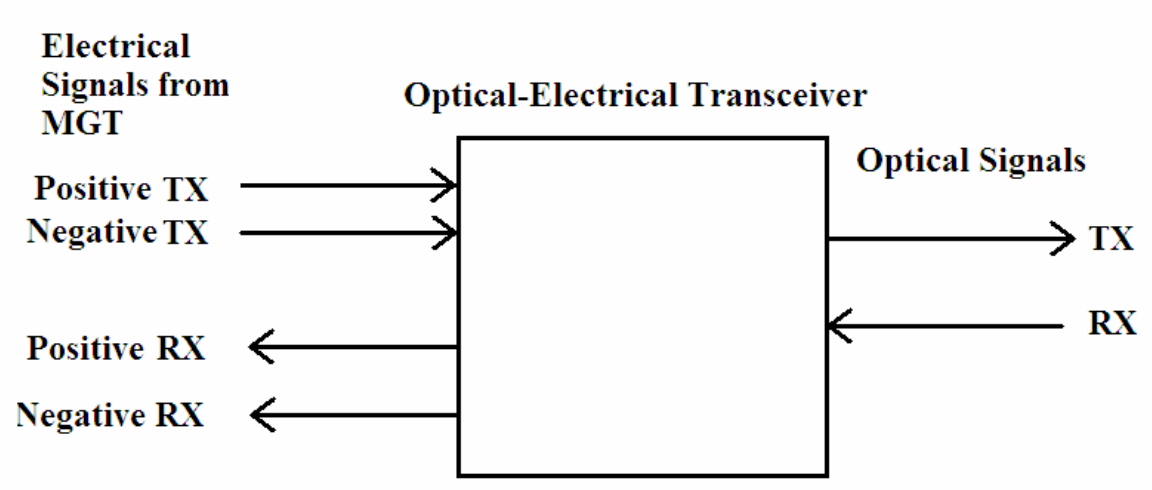

Figure 2: An optical-electrical transceiver

The transceiver is protocol independent. It does not change the signal, just converts it. There are a total of four wires per MGT (a positive and negative input, and a positive and negative output). The four electrical signals are converted to two optical signals, because optical signals are resistant to noise and they degrade much less over distance compared to electrical signals [7]. Using optical transmission allows much longer cables and eliminates many signal integrity issues, and may require less power than electrical transmission. The optical signals travel over eight fiber channels to the corresponding daughter card in a different board, which converts these signals back to differential electrical signals. The board to board connection is shown in Figure 3.

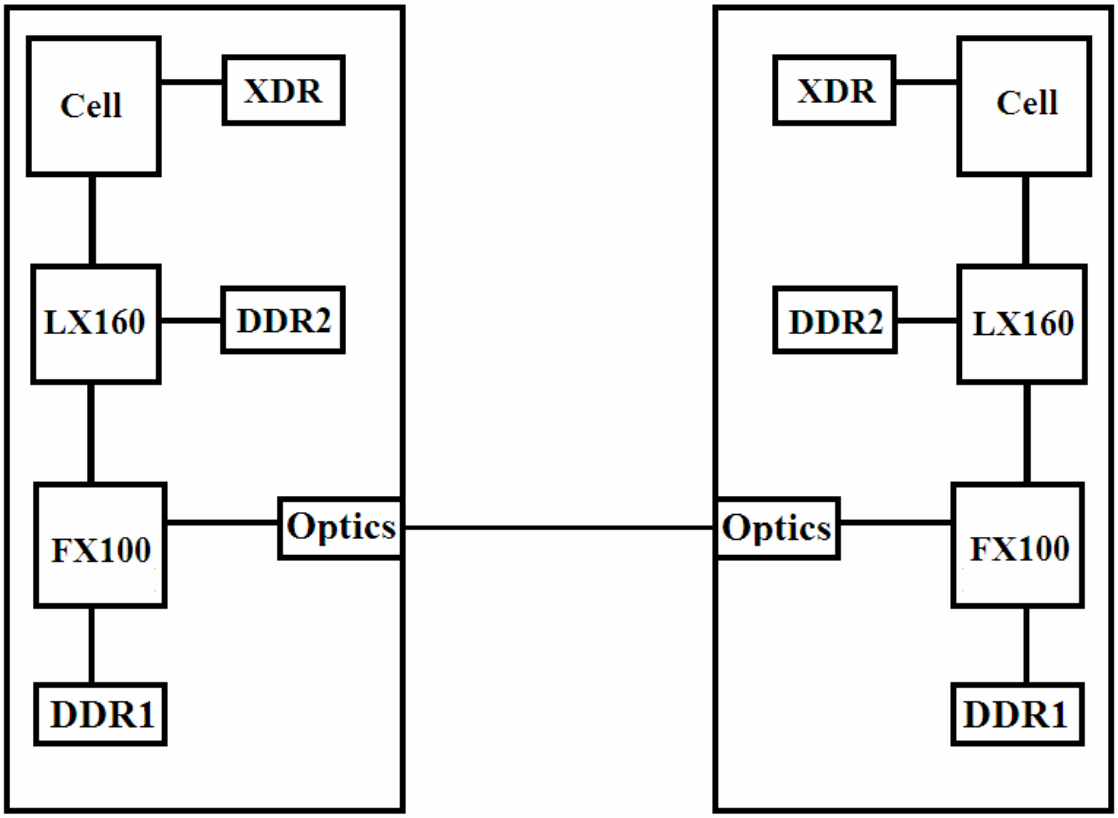

Figure 3: Two Chameleon blades connected via an optical cable

2.2 Aurora

The FX100 FPGAs need a protocol by which to communicate. Our solution uses Aurora, which is an open source, high-speed serial communications protocol. It is scalable and is used for creating point to point links. Aurora has the ability to bond multiple lanes together to form a variable-size channel, giving the ability to add bandwidth if necessary. While other communication protocols are available, Aurora is free from Xilinx, runs on the FPGA available to us, and is completely open source, so that modifications can be made to the protocol. A conceptual view of Aurora is shown in Figure 4 [4].

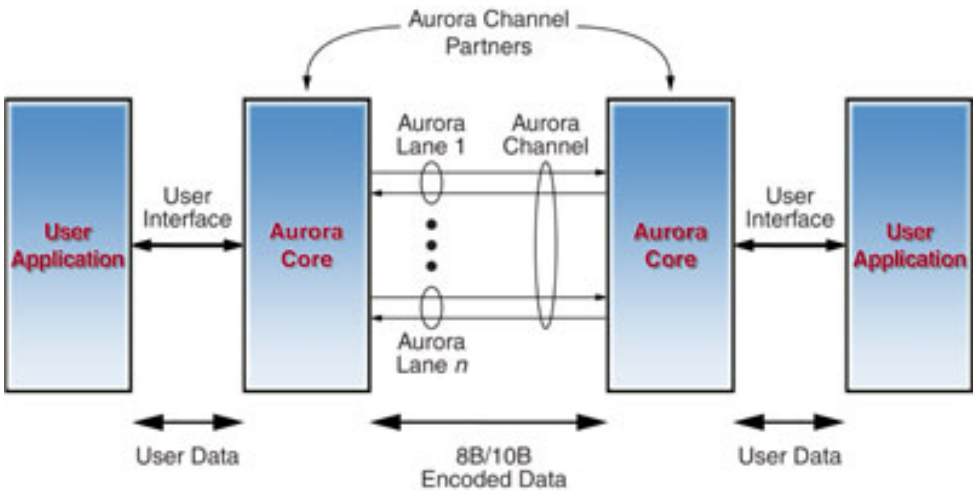

Figure 4: Conceptual view of Aurora

The goal of Aurora is to allow two separate user applications to communicate with each other. These user applications communicate with Aurora via the LocalLink interface [8]. This interface has data and control signals that allow the user application to transmit and receive data through Aurora, which then performs 8B/10B encoding of the data, and transmits this data over the channel. The user interface has 16 bits per lane, and thus Aurora includes a serializer/deserializer, but each lane is serial. There can be as little as one lane, and each lane is bidirectional.

When Aurora is first activated, it begins by training each lane. Data is transmitted between the two channel partners to make sure there are no transmission errors. Once the lanes are working correctly, all available lanes are used to form a single channel. This is called channel bonding. Another initialization procedure takes place to test the channel. Once this completes successfully, the channel is up, and the user application is ready to send and receive data.

The LocalLink interface is shown in Figure 5. This interface is set for framing. A frame is a variable size unit of data to be transmitted. There are five control and data signals on the receiver side, and six on the transmitter side. RX stands for receive, while TX stands for transmit. Note also that some of the control signals have _N at the end of their name. This means they are active low, instead of active high. SRC_RDY_N

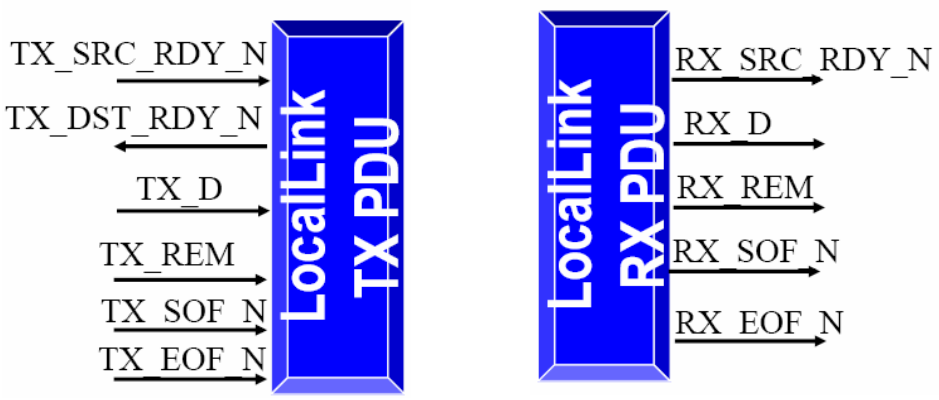

Figure 5: The Xilinx LocalLink interface

stands for source ready, which is controlled by the transmitting side. When this signal is active, it means the transmitter is currently putting valid data on the bus. The receiving side looks at this signal to know if it should be pulling data off the bus. The data bus for the transmitter is TX_D, and the data bus for the receiver is RX_D. TX_DST_RDY_N stands for transmit destination ready, and is a control signal that tells the user application when it is ok to start sending data. If Aurora just finished sending a frame, or if it is performing clock correction, it is not ready to begin sending a new frame immediately. TX_REM stands for transmit remainder, and is only used when sending a partially full cycle of data. For example, a one-lane channel has a 16-bit interface. To send eight bits, the sender uses TX_REM to make the receiver aware that only part of the data in the frame is valid. TX_SOF and TX_EOF are controlled by the transmitting side, and stand for start of frame and end of frame. TX_SOF is asserted low during the first cycle of the frame, and TX_EOF is asserted low during the last cycle of the frame.

1. **Sending a frame:** To send a frame, the destination (receiver) must be ready. The user application determines this by looking at TX_DST_RDY_N, or transmit destination ready. Once the channel is up, there are only two times when the receiver is not ready. One possibility is when the receiver has just finished obtaining a frame. When this occurs, the receiver needs two cycles to process the frame before it can receive another. The second possibility is when Aurora is performing automatic clock correction. There

are two different boards and crystals controlling clocks, so skew is possible. Aurora periodically checks for skew and corrects if necessary. If TX_DST_RDY_N is not asserted, meaning that the receiver is not ready, then the user application must wait before transmitting data.

The next step is to assert TX_SOF_N and TX_SRC_RDY_N. SOF signals the start of frame, and SRC_RDY signals that there is valid data on the bus. This means that data needs to be put on TX_D. TX_D is a bus whose size is a multiple of 16. Every cycle that source ready is active, data needs to be put on this bus. TX_SOF_N is only asserted for a single cycle, but TX_SRC_RDY_N must be asserted every cycle that valid data is on the TX_D bus. If the user application needs to pause, it simply deasserts TX_SRC_RDY_N. Once all the data has been transmitted, TX_EOF_N is asserted. This signals the receiver that the frame has completed. The end of frame signal is required because frame size is variable.

2. Receiving a frame: To receive a frame, the user application on the receiving side must monitor the RX_SOF_N signal, which tells it when a new frame is arriving. Once this signal is asserted, valid data is on the RX_D bus for every cycle that RX_SRC_RDY_N is asserted. Xilinx documentation suggests using a FIFO connected to RX_D so data can be shifted in as it comes. If RX_SRC_RDY_N is deasserted, then the data on RX_D is not valid. Asserting RX_EOF_N marks the end of the frame. Figure 6 displays a waveform showing these transactions.

The two Aurora cores are labeled aurora1 and aurora2. On the first cycle, the first Aurora core asserts TX_SOF_N, TX_SRC_RDY_N, and puts the value 17 on the TX_D bus. TX_SRC_RDY_N is asserted for a total of four cycles, and the values 32, 790, and 541 are also placed on the TX_D bus. Also shown on the first cycle of the waveform is aurora2 receiving this data. RX_SOF_N and RX_SRC_RDY_N are asserted, and RX_SRC_RDY_N is held low for four cycles. The values of 17, 32, 790, and 541

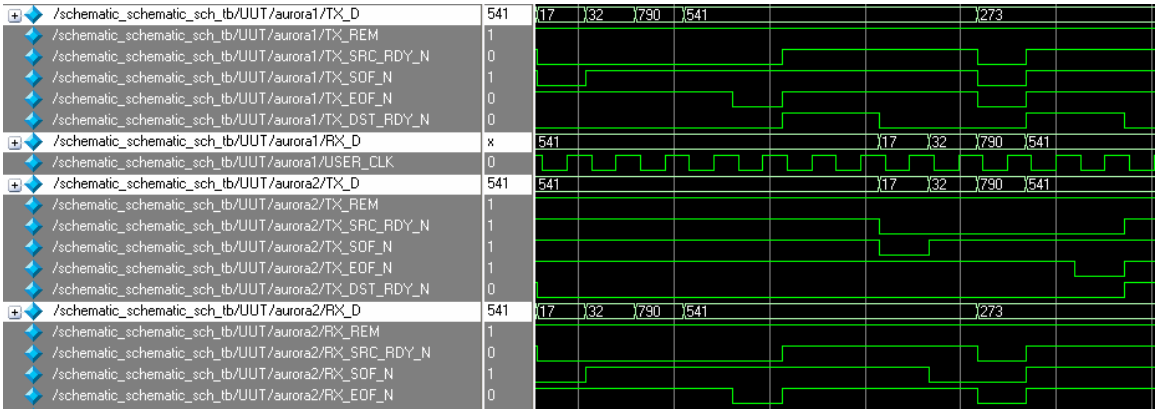

Figure 6: A ModelSim waveform showing a simulation of Aurora

come in the RX_D bus, and on the last cycle, RX_EOF_N is asserted for a single cycle. Obviously, there is latency involved in these transactions. It takes about 38 clock cycles for the data to be received after it is sent. This is due to 8B/10B encoding/decoding, as well as serialization/deserialization of the data. The waveform above was captured from a simulation testbench sending repeating patterns of data, which is why it looks as though the data is received as soon as it is sent.

2.3 Inter-Blade Connection Network

Using Aurora as a connection between two Chameleon blades allows us to begin to create a shared memory system. Aurora makes it possible to have a connection between the two FX100 FPGAs on the two Chameleon blades. Since there is already a bus that connects the FX100 with the LX160, and the LX160 with the Cell, this is now a connection between Cell processors. To show that it is possible to connect more than two Chameleon blades together, we create a bidirectional ring. Each optical daughter card has two connectors, each with four lanes. The ring has four lanes going in either direction. An example of this ring is shown in Figure 7.

Since this design is a ring, routing is required. There is not a direct connection between each blade. Each Chameleon has a unique ID, and packets contain source and

destination IDs. When a Chameleon receives a packet, it decides whether to process it or send it on. More details about the routing and packet format are discussed in Section 3.

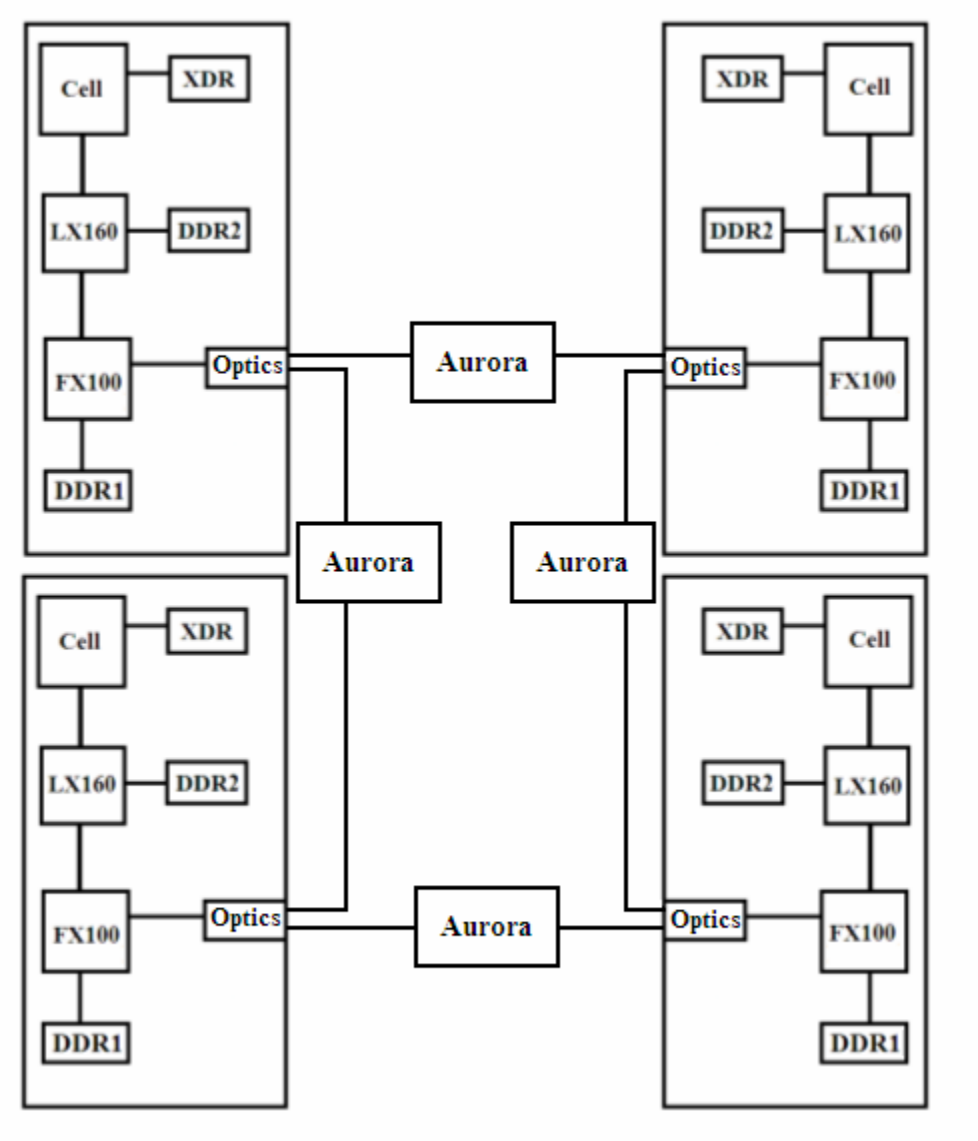

Figure 7: A four-blade ring design

3 Validation

The first step of validation is to understand the Xilinx tools, which includes Aurora. Section 3.1 describes the Xilinx tools we use, as well as a single lane Aurora design for the Virtex-II Pro. Section 3.2 describes the Aurora design for the Chameleon blade, which differs from the Xilinx version due to having multiple lanes and a Virtex-4 FPGA [6]. Section 3.3 describes the shared memory space that each of the blades has access. Section 3.4 describes image transfers between blades, which we use to demonstrate the functionality of the shared memory. Section 3.5 describes the Cell code to access the shared memory. The two-blade demonstration setup is described in Section 3.6. The details of the multi-blade ring design and enhancement over the two-blade design are given in Section 3.7. The four-blade demonstration setup is described in Section 3.8.

3.1 Initial Aurora Design

The FPGAs on the Chameleon are from Xilinx, so we use Xilinx tools. The main design tool used is Xilinx ISE [9]. ISE has a text editor to assist with HDL programming, project wizards to help create and maintain the project hierarchy, and many other integrated tools to help in the design process. We generate the logic for Aurora with the Xilinx Core Generator by specifying the type of FPGA, which Xilinx IP we want to generate, and the customization options. Initially, a single lane Aurora design for the Virtex-II Pro FPGA was generated. This Aurora module is designed to run at 2.5 Gbps. We chose to start with a Virtex-II Pro design because the Chameleon blades were still being put together and tested; they were not available yet for use. We needed technology similar to that on the Chameleons, which have Virtex-4 chips and four-lane Infiniband connectors. Our initial boards therefore used Virtex-II chips and single-lane Infiniband connectors.

Next we verify the functionality of Aurora. MTI's ModelSim SE [10] software is useful for this. The waveform in Figure 6 was obtained using ModelSim. The simulation

shows the two Aurora cores successfully connecting and initializing a channel. It is now possible to send data back and forth between the two cores. The main difference between running on software and hardware is that the FPGA I/O pins must be set correctly. For example, clock inputs and connections to the MGTs need to be defined. Reset signals are required for initialization to function correctly. On hardware, it is necessary to create a reset block that counts clock cycles to trigger the resets at the correct times. An Aurora core is loaded into each FPGA. Single lane InfiniBand cables connect the two boards. ChipScope Pro [11] is a software logic analyzer that allows the user to observe the cores communicating. The signals we would like to scope are defined in a ChipScope analysis file. These signals can be accessed using the programming cable connected to the JTAG interface. JTAG is a commonly used IEEE standard for programming FPGAs. Using a separate JTAG interface means that none of the I/O pins need to be occupied for debugging purposes, and all of the signals can be seen simultaneously on a PC. Figure 8 displays an image of ChipScope.

The left side shows various signal names. The right side shows the corresponding waveforms. ChipScope takes up to 16K samples from up to 255 signals, which are triggered on any state or state change of one or more signals that we choose. In this example, a frame is being sent. The ADDRESS_INPUT bus is holding 0x02BFE, while the DATA_INPUT bus is holding 0xFFFFFF. The control signals are active low. Then the start of frame signal, tx_sof_n_i goes low for one cycle. At the same time, the source ready signal, tx_src_rdy_n_i, goes low and stays that way for four cycles. At the end of the frame, the end of frame signal, tx_eof_n_i goes low for a single cycle. The frame is four cycles, because we have 64 bits to transmit, this is only a single lane, and 16 bits are transmitted per cycle. In the first cycle, the upper 16 bits of the address are sent (0x0000), then the lower 16 bits (0x2BFE), then the upper 16 bits of data (0x00FF), then the lower 16 bits (0xFFFF). Using ChipScope allows verification that the Aurora design is functioning correctly on hardware.

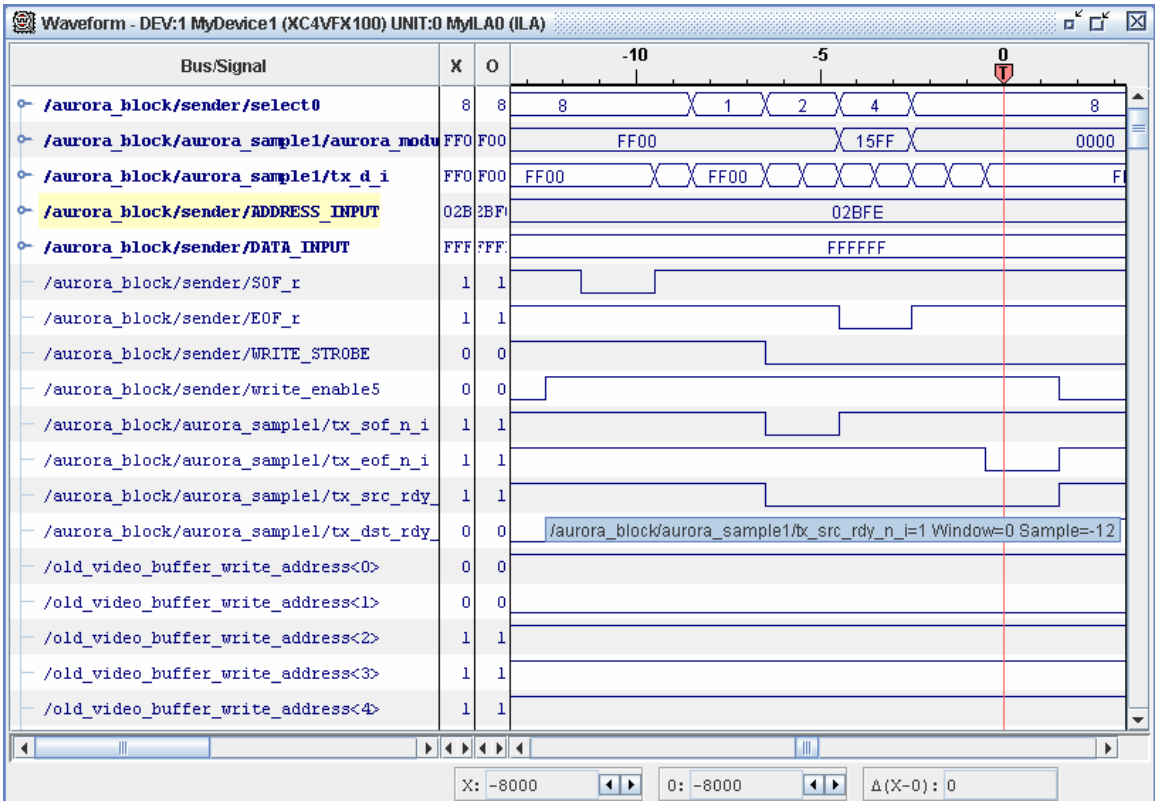

Figure 8: A Xilinx ChipScope screenshot showing Aurora

We test the base Aurora code in these two initial tests. A frame generator creates frames, and a frame checker receives these frames and ensures their correctness. Sending and receiving other data requires that these blocks be placed in two different logic blocks, a sender and a receiver. The sender block is designed to accept 32 bits of address, 32 bits of data, and a write strobe signal. When the write strobe is received, it correctly controls the LocalLink interface signals to send that block of data. The receiver monitors the LocalLink, and looks for the start of frame signal. When the start of frame signal is received, it collects and reassembles the address and data bits and presents them as outputs. A data ready signal signifies a new packet was received and is ready for use.

3.2 Aurora Design for Chameleon

Eventually, a single Chameleon blade became available. The Chameleon has a different FPGA and so we had to recreate the Aurora design. The Virtex-4 has many more options, such as what reference clock to use, which leads to different line rates. However, the greater number of inputs and outputs complicate the design. For example, two new reset lines are added, and they need to be activated a certain amount of time after the design is loaded. If these inputs are ignored or not used correctly, Aurora does not initialize. Another problem is choosing the line rate. At first, we thought it was a good idea to choose a slow line rate of 0.625 Gbps so we could monitor the MGT signals for integrity issues. However, running at such a low line rate required enabling an over-sampling option in the MGT because the lower limit on frequency of the PLL was about 1GHz.

We redesigned Aurora to run at 2.5 Gbps, but this created another problem. The design required a 250 MHZ reference clock, a clock frequency we did not have on the Chameleon. The 100 MHz clock was replaced with a 250 MHz high tolerance oscillator to minimize clock jitter.

To connect a second Chameleon blade, two options were available: cut the IB cable and move the wires around, or try an optical daughter card. The optical card was ready to be tested, so we chose this. All that needed to be done was to change the MGTs on the FX100 and redirect them to the PCI-E connector.

3.3 Shared Memory Space

To use Aurora for communication between distributed shared memory locations, we created a shared memory space. A small block of SRAM was instantiated on the FX100. We do this because a DVI controller is also on the FX100, and it uses the data in this video buffer as input. It outputs DVI signals to a connector on the Chameleon blade, so that image data is displayed on a DVI monitor. The DVI controller is configured to

display a 320x240x24 bits image on the screen. This means the size of the video buffer is 225KB.

To get data into the video buffer, we connected the Cell to the FX100. Firmware in the Cell processor issues read and write requests using index addressing, i.e., it has three registers it uses to access the video buffer, an address register, a data in register, and a data out register. These are connected to the LX160 chip over a bus from the Cell. This bus is currently in debug mode, and is only eight bits wide and runs at 50 MHz. The real bus is 32 bits wide and can be clocked up to 5 GHz. Once the LX160 receives a read or write command, it sends the data and address values down to the FX100 using a 64-bit data bus. The read enable and write enable control signals dictate how the data are handled. There is another 64-bit data bus that goes back to the LX160 from the FX100 for reading, along with another control signal called read data valid.

The data bus to the FX100 contains a 32 bits of address and 32 bits of data. Because of the small video buffer size, only 17 address bits are required (2^17 = 128 K entries, and each entry is three bytes, for a total of 384KB). Each pixel is only 24 bits. The Cell sends writes, pixel by pixel, into the video buffer, and the image appears on the monitor. The main hurdle is that there are now different clock domains in the FX100. An Aurora user clock runs at 125 MHz, a video buffer clock runs at 100 MHz, and an inter-FPGA clock runs at 50 MHz. Crossing these clock boundaries requires latching data and control signals for multiple cycles to ensure signal integrity. The above solution enables the Chameleon to use Aurora to connect two video buffers together, creating a shared memory space.

3.4 Image Transfers Using Shared Memory

Two more steps are necessary to complete the two-board design. The first expands the Aurora design to eight lanes, fully utilizing the optical daughter card. This requires another complete regeneration of Aurora code. At this configuration, each lane has a

bandwidth of 2.5 Gbps, for a total of 20 Gbps for the channel. The Aurora LocalLink interface allows for parallel input of 16 bits per lane (which it later serializes), and for eight lanes, the transmit and receive interfaces are 128 bits wide. Packets are only 64 bits (32 for address, 32 for data), and each frame now takes a single cycle. That means that once the new address and data are available and the write strobe is received, the start of frame, end of frame, and transmit source ready signals are all active for a single cycle, and then deactivated. The receiver grabs the single cycle packet, but only looks at 64 of the 128 bits.

The second step is to add intelligence to the FX100 so that each Cell processor can read and write to local and remote shared memory. We add an 18^{th} address bit used to differentiate the local vs. remote video buffer. If this bit is a 0, it is a local address; if it is a 1, then it is a remote address. There are also status bits to signify a read or a write.

3.5 Accessing Shared Memory

There are four actions that can occur when the shared address space is accessed from the Cell. First is a local write, into the local video buffer. Second is a local read, from the local buffer. Third is a remote write, which sends packets of address and data over the optical link using Aurora. The most difficult is a remote read, where a packet is sent from the local FX100 to the remote FX100, requesting a certain piece datum at a given address. This data must be read from the video buffer and sent back to the requesting FX100. The local side then receives this packet and sends the data up to the Cell. Several state machines are responsible for controlling all of these actions.

Local Write: A local write is the simplest of the four actions. When a local write occurs, the 18 bits of address and 24 bits of data are sent to the LX160 using the data out and address register. The LX160 passes these values to the FX100 along with a write enable signal. When the FX100 detects this write enable, it latches the address and data

values, and sends these to the video buffer, and asserts the write enable signal. Once the write completes, the write enable signal is de-asserted.

Local Read: The Cell attempts to read its local shared space. The Cell sends an address and read pulse to the LX160, which passes these signals on to the FX100. When the FX100 receives the read pulse, it latches the address and puts it on the video buffer's address input bus. The write enable signal stays deasserted, and data at that address are put onto the address output bus. The FX100 latches this data and sends it back to the LX160 over another unidirectional bus, separate from the one used to issue a read or write command. The LX160 sends the read data to the Cell, and the data are put into a register and are ready for use.

Remote Write: Remote writes are similar to local writes, except that they must use Aurora to exploit the optical link to the remote board. When the FX100 receives a write pulse, it analyzes the address to see if the 18^{th} bit is set. If so, it creates a 64 bit packet (32 address bits, 32 data bits) to send to another state machine. Bits 23:20 of the address are set to indicate the type of frame. If these bits are 0000, then it is a remote write, and the packet contains a valid address and data. This state machine latches this frame data and controls the Aurora LocalLink interface appropriately. Data are placed on the Aurora transmit bus, and the start of frame, end of frame, and transmit destination ready signals are asserted for one Aurora user clock cycle. Because the data rate is 2.5 Gbps per lane, the user clock runs at 125 MHz. To calculate this, we take the unencoded bit rate, which is 2.5 Gbps, and multiply by 0.8 to get the actual data rate: 2.0 Gbps. 8B/10B encoding is used, which is why we multiply by 0.8. We divide 2.0 Gbps by 16 bits, the interface width per lane, which yields the user clock frequency of 125 MHz. The remote FX100 has a receiver state machine that detects start of frame and begins capturing data. This is a single cycle frame, so only 128 bits are captured. The 32 bits of address and 32 bits of data are latched into registers, and the upper 64 bits are ignored. Another state machine is signaled when a remote request arrives. Bits 23:20 of the address are checked to

determine the type of frame. The receiver state machine sees 0000 and therefore knows it is a remote write. The address bits are placed onto the video buffer address bus, the data bits are placed onto the video buffer data input bus, and the write enable signal is asserted. No acknowledgement is returned to the initiator of the remote write. There is an infrastructure in place that can use bits 31:24 of the address as a sequence number, so that the receiver can send acknowledgements back to the sender using this as an identifier, but this is not yet implemented.

Remote Read: The FX100 sees the read strobe, the data and address are latched, although at this point the data are not valid. The 18^{th} bit of the address is set, so a frame is created and sent over Aurora. This time, the frame type is set to 1100, signifying a read command. The receiver knows that only the address within the frame is valid. The address goes onto the video buffer address bus, and write enable is unasserted. The address and read data are put into a new frame with packet type 1111. This signifies that this packet is returning read data from a previous read request. The packet is sent back to the initiator. The receiver sees the packet type of 1111 and sends the address and data to the LX160, which sends it to the Cell. The Cell reads this data from the data in register. If the remote read was done because data is being copied from the remote video buffer to the local video buffer, the Cell performs a local write of the new data. The address is the same, except the 18^{th} bit is 0.

To demonstrate that each Cell can access each other's remote memory, we issue memory accesses from the Cell to addresses that map to the remote and local shared memory buffer. The address register, data in register, and data out register are used to perform these memory accesses. We wrote C code which controls these registers in order to execute these reads and writes.

3.6 Two-Blade Demonstration Setup

To prove that the two blades can access their local and remote shared memory, a command is issued that writes a 320x240 image into the local shared memory space (video buffer). The DVI controller displays this image on a monitor. Different images on each blade are used to show that the data in each video buffer is different. From one of the blades, a command called remote2local is issued. This command first sends out a read request to the first address in the remote video buffer. The FX100 performs a remote read and the data is eventually sent back up to the Cell. The Cell then does a local write of that data to the first address in the local video buffer. This is repeated for every pixel in the image. When this is done, the remote image is now on the local screen. To prove it is not reading from the flash that stored the original images, the pictures are modified. After issuing another remote2local instruction, after the pixel by pixel copy, both monitors are displaying the exact same image. To show remote writes, a C program was written that displayed a ball moving about the screen. To change the location of the ball on the screen, it is redrawn at a different address. But, the address space it can move around in is not only the local video buffer space, but also the remote. The local Cell processor is continually issuing writes. Some of those writes are to local space, and some are to remote. The two monitors show the ball moving between each of them. Once the ball leaves the first monitor, it appears on the second monitor and bounces around there until it bounces back to the first monitor. The Cell processor on the remote blade is sitting idle, while the Cell on the local blade is continually issuing writes without any idea that it is writing to a local or remote address space. The combination of the remote2local copy and the bouncing ball proves that these Chameleon blades have the ability to read from and write to their local and remote address spaces.

3.7 Multi-Blade Ring Design

Having validated our Aurora implementation, we established two new milestones: to create the ring network to include more than two systems, and to increase the size of the shared memory space. We placed a DDR1 memory controller in the FX100. This space replaces the SRAM block used in the initial validation. The DDR1 becomes the new video buffer, and it has more storage space.

The ring design requires two major communication changes. First of all, we need two Aurora cores per FX100, and the eight-lane channel must be broken up into two four-lane channels. Second, routing needs to be added so that frames from each board can arrive at the correct destination.

3.7.1 Four-Lane Aurora

Creating the four-lane Aurora core should have been relatively easy. But, new problems arose. The RocketIO MGTs come in pairs, and designs need to use the two MGTs together. Each four-lane optical connection needs to carry one channel. The MGT pairs wired to the PCI-E connector are 110, 112, 113, and 114. Each of these pairs has an A and a B MGT. By tracing the routing from the optical card, the first optical connector is wired to MGTs 110A, 110B, 112A, and 114A. The second optical connector is wired to MGTs 112B, 113A, 113B, and 114B. But we can not split up MGT pairs 112 and 114 like this. The temporary solution is not to use MGTs 114A and 112B. This solves the pairing problem, but reduces the number of lanes to three per channel. Rerouting wires in the optical card avoids this problem in later solutions.

3.7.2 Routing

Adding multiple sources and destinations for the data frames requires routing. For example, the Cell IDs can be assigned as shown in Figure 9.

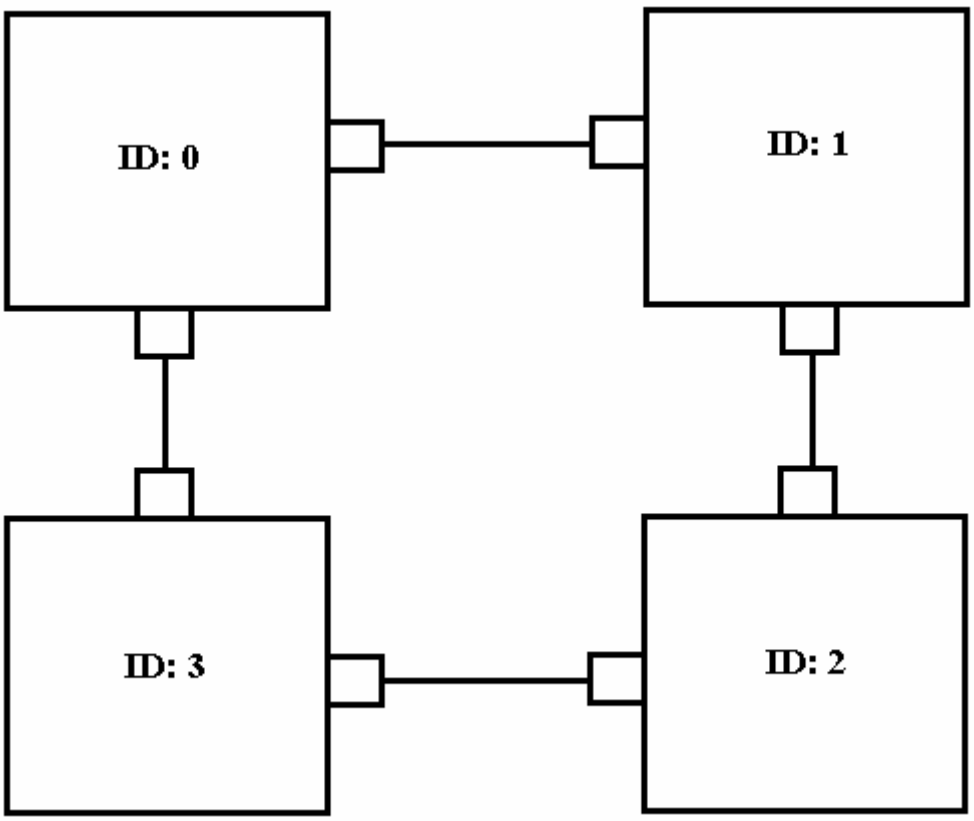

Figure 9: Processor IDs of the four blades in the ring

When a frame is sent, more information is needed than an address and data. We send the command word, the address word, and two data words. Figure 10 displays the contents of these words.

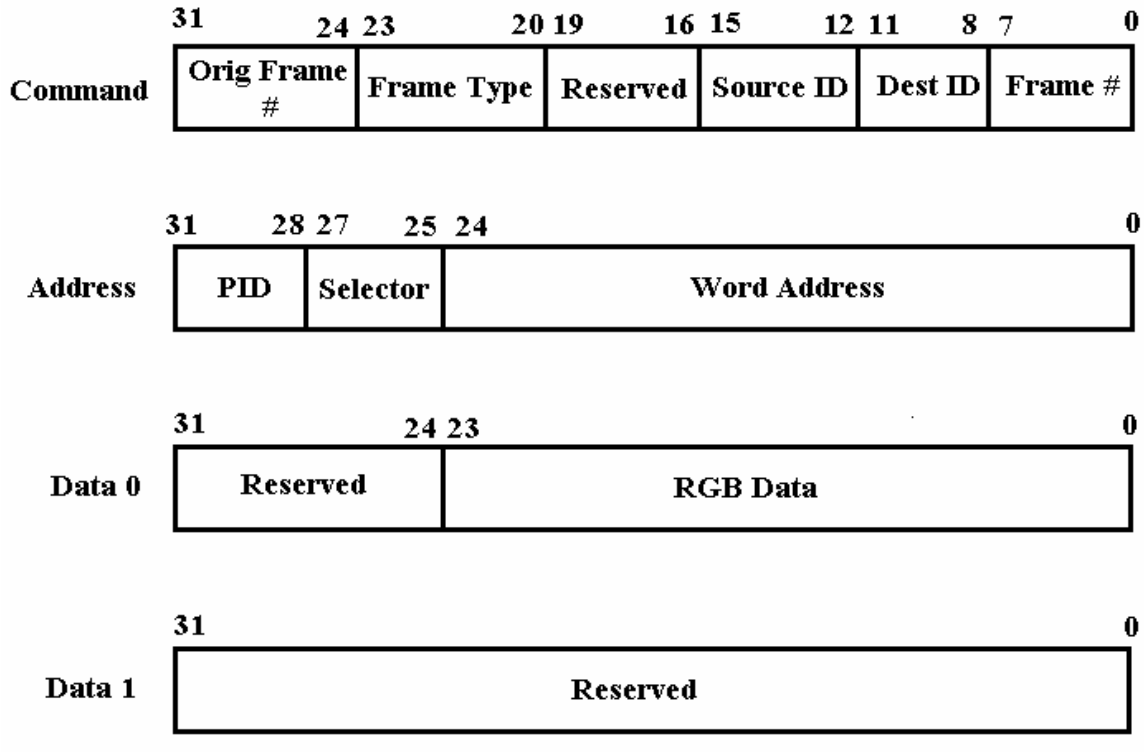

Figure 10: The contents of the four words of each frame

The command word has seven fields. The first is the original frame number. There is another frame number in bits 7 to 0. If a frame is a request frame from one blade to another, the requesting blade assigns a frame number, which is put in bits 7 to 0. After the receiving blade receives this frame and is ready to send it back, it assigns its own frame number in bits 7 to 0, but puts the original frame number in bits 31 to 24. The frame number is not used in the current logic, but is put there for future use, such as to give the blades a framework to send acknowledgments for all frames sent and received. The frame type has five different possibilities, as shown in Table 2.

Table 2: Possible frame types

Frame Type Value	Frame Type Name
0000	Write Request
1100	Read Request
1111	Read Data Being Returned
1001	Load With Reservation
1010	Store Conditional

A blade uses a write request frame to write into another blade's memory. The address word contains the write address, and the data word contains the data to be written. A blade uses a read request frame to read from another blade's memory. The address word specifies where to read, but the data word is ignored. After a read completes, the data is sent back to the requestor blade using the read data being returned frame. The address word specifies where to read, and the data word holds the data that is read. The load with reservation and store conditional frame types are not yet used. They will be used for coherency protocols.

Bits 15 to 12 hold the requestor's source ID. This is needed when a frame needs to be returned to a requestor, such as in the case of a read. Bits 11 to 8 hold the destination blade ID, which is where the frame stops and is serviced.

The address word specifies where to write to or read from. The DDR1 memory is 128 MB. Each addressable block is 32 bits, which means we have 32M words to be addressed. 32M requires 25 bits of address, which is the lower 25 bits of the address word. To have a global shared memory, we also need address bits to distinguish between the different blades. This is what the PID is used for. The PID, or processor ID, distinguishes the different blades from each other. There is also a selector field to determine whether we read from/write to the DDR1 memory or the SRAM block. The

Data 0 word holds the 24-bit RGB value for each pixel. Data 1 is currently unused. This leaves room for expansion if any additional features need to be added.

A routing table decides which direction to send each frame. The design is simple: it chooses the direction yielding the fewest hops between source and destination. Table 3 shows the contents.

Table 3: The routing table

Source ID \ Destination ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	X	1	1	1	1	1	1	0	0	0	0	0	0	0
2	0	X	1	1	1	1	1	1	0	0	0	0	0	0
3	0	0	X	1	1	1	1	1	1	0	0	0	0	0
4	0	0	0	X	1	1	1	1	1	1	0	0	0	0
5	0	0	0	0	X	1	1	1	1	1	1	0	0	0
6	0	0	0	0	0	X	1	1	1	1	1	1	0	0
7	0	0	0	0	0	0	X	1	1	1	1	1	1	0
8	1	0	0	0	0	0	0	X	1	1	1	1	1	1
9	1	1	0	0	0	0	0	0	X	1	1	1	1	1
10	1	1	1	0	0	0	0	0	0	X	1	1	1	1
11	1	1	1	1	0	0	0	0	0	0	X	1	1	1
12	1	1	1	1	1	0	0	0	0	0	0	X	1	1
13	1	1	1	1	1	1	0	0	0	0	0	0	X	1
14	1	1	1	1	1	1	1	0	0	0	0	0	0	X

As an example, the table entry corresponding to source ID 5 and destination ID 9 is 1. That means that when blade 5 writes to blade 9, it uses channel 1 to send the packet. Channel 1 connects blade 5 to blade 6, which is the correct direction for the fewest hops. Since there are an even number of blades, the farthest blade from the sender is seven hops in either direction. When any blade sends to its farthest blade, both directions yield equal-

length paths. We simply choose half such situations to use channel 0, and half to use channel 1.

3.7.3 DDR Access

The last logic block added is an arbiter to the DDR1 memory. There are multiple sources of requests for the DDR: the Cell on the blade performs burst writes directly, requests come from Aurora, and the DVI controller performs reads to display an image. An arbiter ensures that only one unit can access the DDR memory at once. The DVI controller is given the highest priority. To maintain the screen image and keep it updated, the DVI controller must read 640x480 pixels from DDR memory, otherwise the screen goes black. If the DVI controller is interrupted, the screen image becomes distorted, and thus all other accesses to DDR must occur between DVI accesses. With a fast interface, this update can occur during the vertical sync (which is usual). Our slower interface (due to debug mode and narrow bus) requires the arbiter tries to perform accesses between DVI reads. If the DVI is not reading, and if there are no requests coming from Aurora, then any burst writes are allowed to proceed. Once Aurora initiates a read or write request, the arbiter first makes sure the DVI controller is not reading. It then waits for all previously issued reads and writes to complete. The read or write from Aurora is then allowed to proceed. If the operation is a read, the requested data are sent back to the Aurora control logic. Once the read or write is completed, another Aurora request can begin if the DVI controller is still inactive. If there are no further Aurora requests, the arbiter allows burst writes from the Cell to proceed. Figure 11 displays the block diagram of these parts in the FX100.

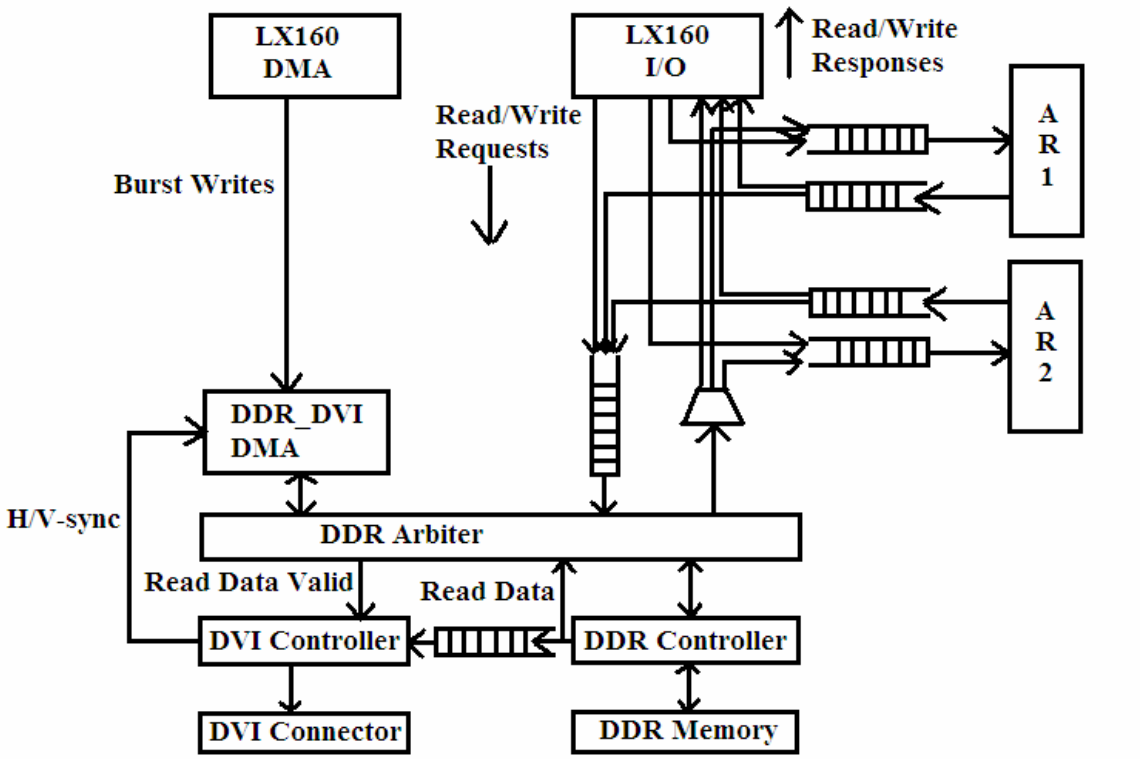

Figure 11: Logic inside the FX100 FPGA

Each processor can only have a single pending request. Requests from other blades must be queued in the FX100. Since there can be up to 14 blades in the ring, FIFO buffers on each blade must be able to hold 14 current requests: 13 from other blades, and 1 from itself. This guarantees that no deadlocks occur.

3.8 Four-Blade Demonstration Setup

The four blades are initialized with an ID (1, 2, 3, and 4). The initial screen image is a set of color bars, to show that the DVI controller and the DDR are working correctly. Two boards are connected to a single dual-input display. Figure 12 shows the initial state of the display. The first image transfer is a local write from board 2, to board 2. Figure 13 shows the result after this is done. The right side of the monitor, driven by board 2, now displays an image. The left side of the monitor, driven by board 3, still has the initial color bars. Figure 14 shows board 2 writing an image to board 3. The left side of the

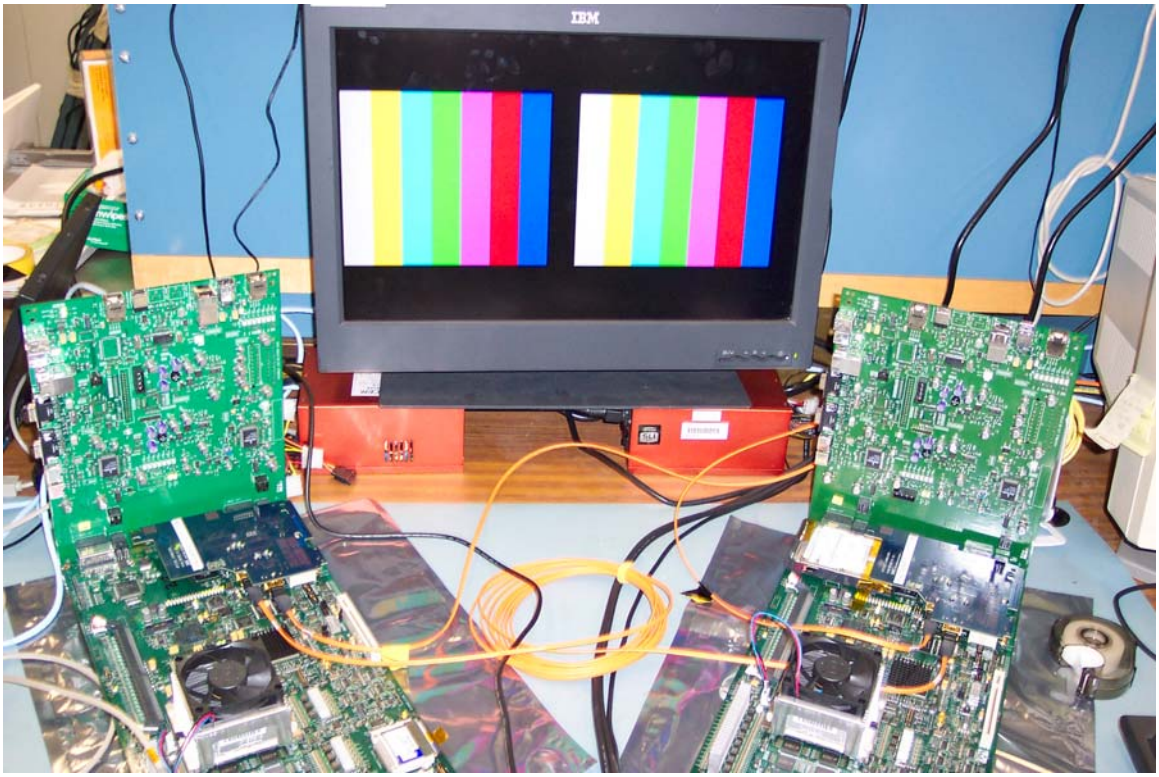

Figure 12: Initial state of the display for the ring demo

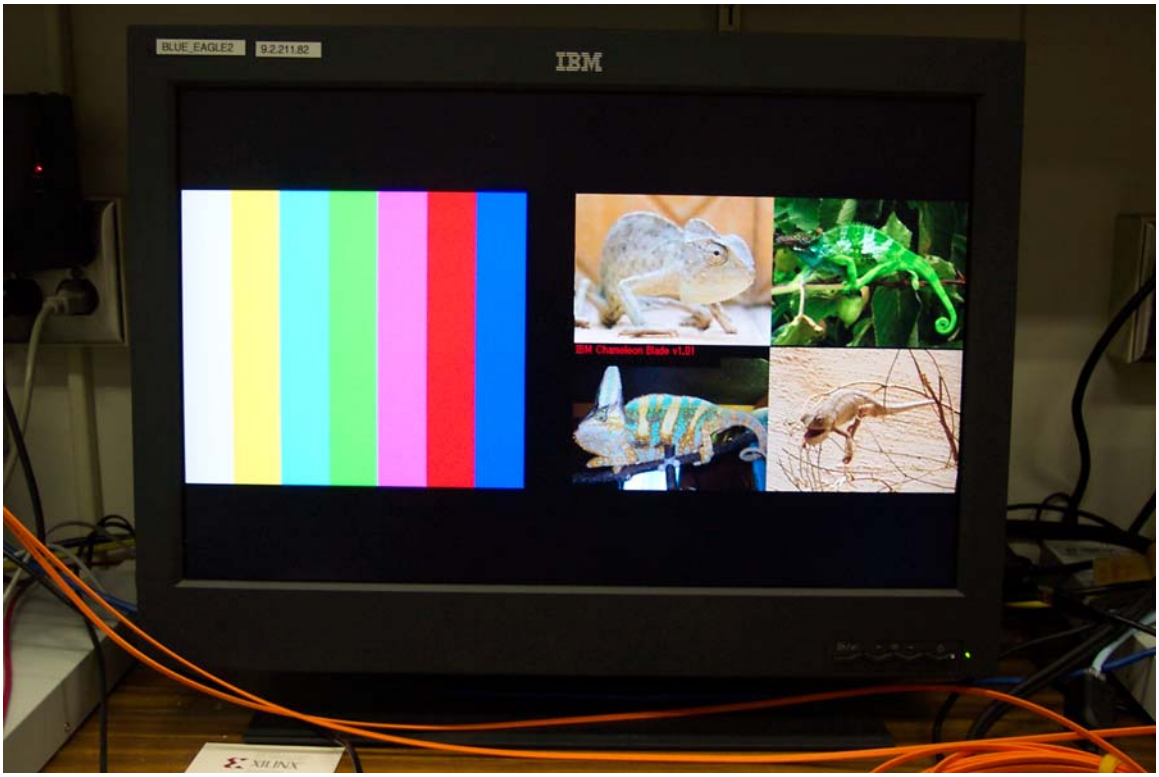

Figure 13: Display after board 2 performs local write (monitor 1)

Figure 14: Display after board 2 performs remote write to board 3 (monitor 1)

Figure 15: Display after board 2 performs remote write to board 4 (monitor 4)

Figure 16: Display after board 2 performs remote write to board 1 (monitor 2)

Figure 17: Display as board 2 reads pictures from board 1 and writes pictures to board 4 (monitor 2)

monitor, controlled by board 3, has a new image. This is accomplished by a pixel-by-pixel write of data from board 2 to board 3. Figure 15 shows what happens when board 2 writes to board 4. Board 4 is two hops away, so the frame is sent to board 3 and passed to board 4. The second monitor displays the result. Its initial state is the same as in Figure 12. Board 4 controls the left side of the monitor, where the new image is displayed. Board 1 controls the right side. Figure 16 shows what happens when board 2 write to board 1. The previous two writes were to boards 3 and 4, causing the frames to be sent the same in the same direction. When board 2 writes to board 1, the frames are sent in the opposite direction from the previous writes. The last part of the demo shows the read capability of the four-blade ring. A C program randomly selects one of the small square pictures currently displayed on board 1, and copies it to a random location on board 4. This program runs on Linux on board 2. This displays the capability for a board to read from one board's memory and then write to another's. Figure 17 shows the display state after the program runs for a short while.

4 Results

A small test block of logic sits in the FX100 FPGA and determines the latency of communication on the ring. This logic performs a read and then a write from a given source blade to a given destination blade. When the read is initiated, a counter is activated. When the read completes, the counter value is recorded and stored. This repeats 64M times. There are 64 buckets for each of the values. The first bucket is 0ns to 159ns. The second bucket is 160ns to 319ns, and so on. The second to last bucket is 9.92µs to 10.079µs. The final bucket is 10.08µs and greater. This covers the range of possible latencies for zero to four hops. Zero hops is a local test, where the request is sent from a logic block inside the FX100, which acts like the LX160, and is handled locally without going over Aurora. We only have four boards, so the four hop test loops around the ring. Latencies for zero to four hops are shown in Figure 18.

The large blue bar at left shows where the majority of the zero hop latencies lie (between 640 and 799ns). The most common latencies for one, two, three, and four hops lie about 2µs apart. Since these latencies account for a two-way trip (send and receive), this means that the one-way latency per hop is about 1µs. To find the exact number and further break down this time, we use ChipScope to find the DDR arbiter latency. Figure 19 shows this.

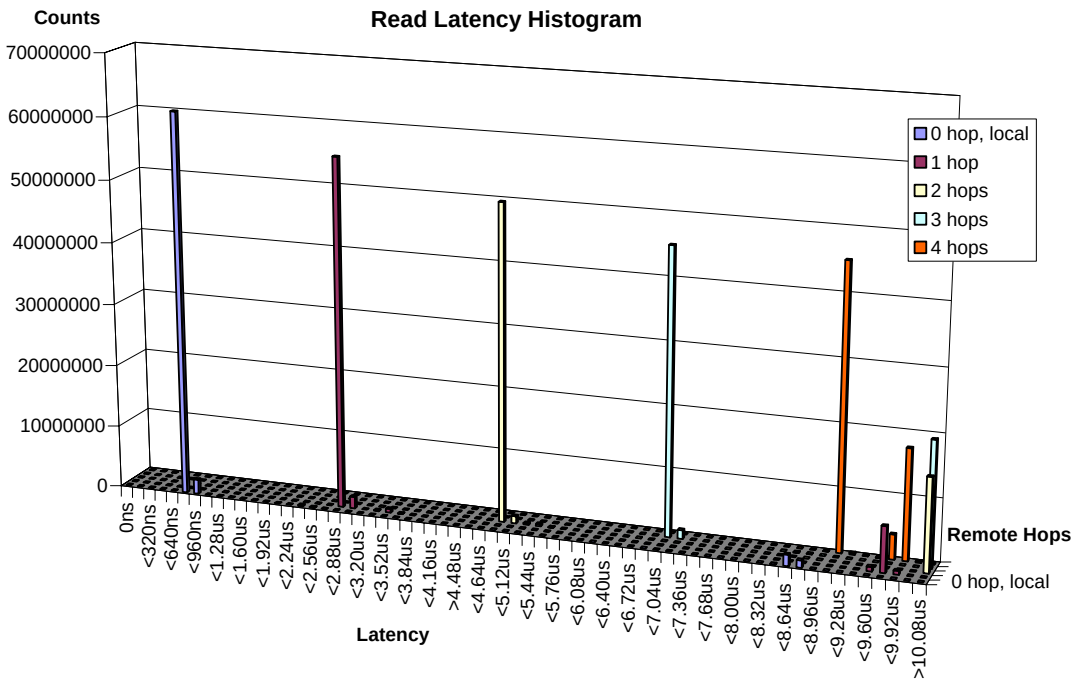

Figure 18: Read latency histogram for 0 to 4 hops

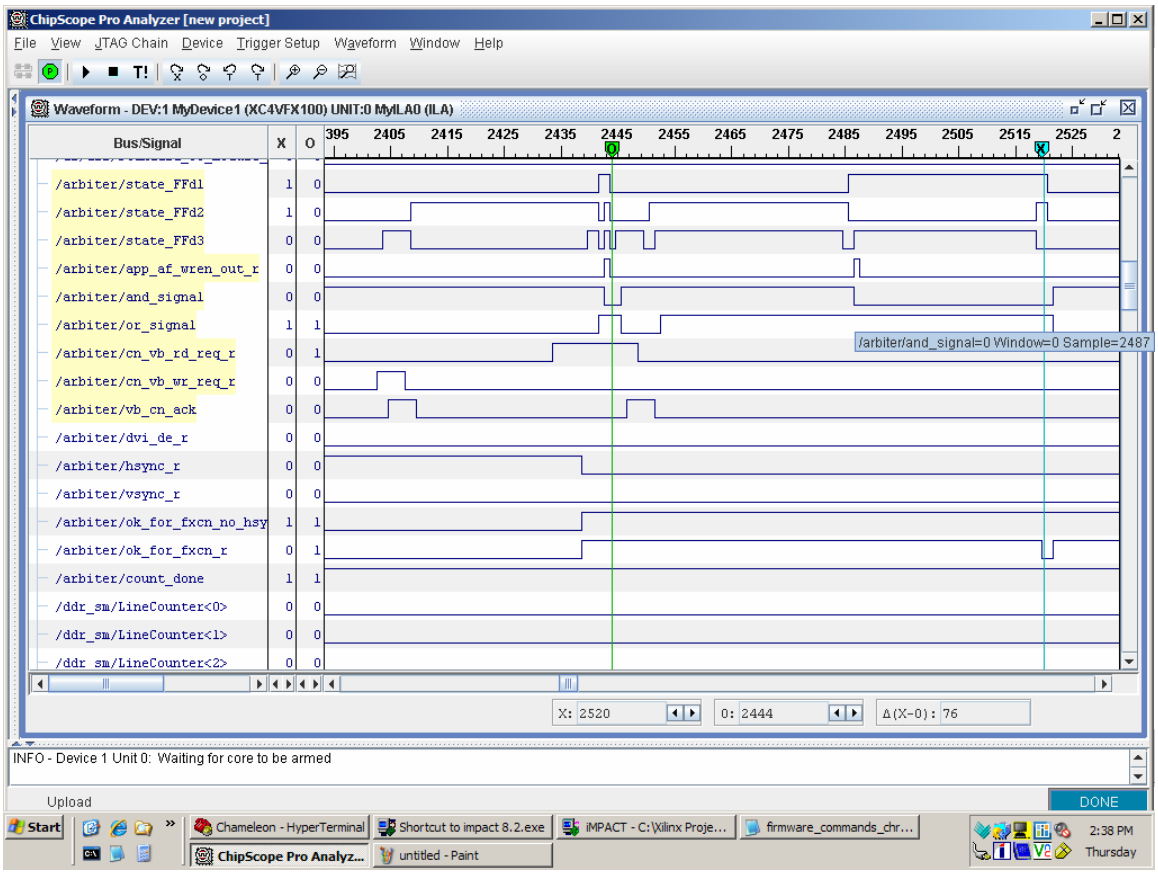

Figure 19: ChipScope showing DDR arbiter latency

This screen shows the progression of arbiter states. The arbiter starts taking action around cycle 2444 and ends around 2520. These cycles have a period of 8ns. Therefore, the arbiter latency is about 608ns. ChipScope can determine the Aurora latency. Figure 20 shows this latency.

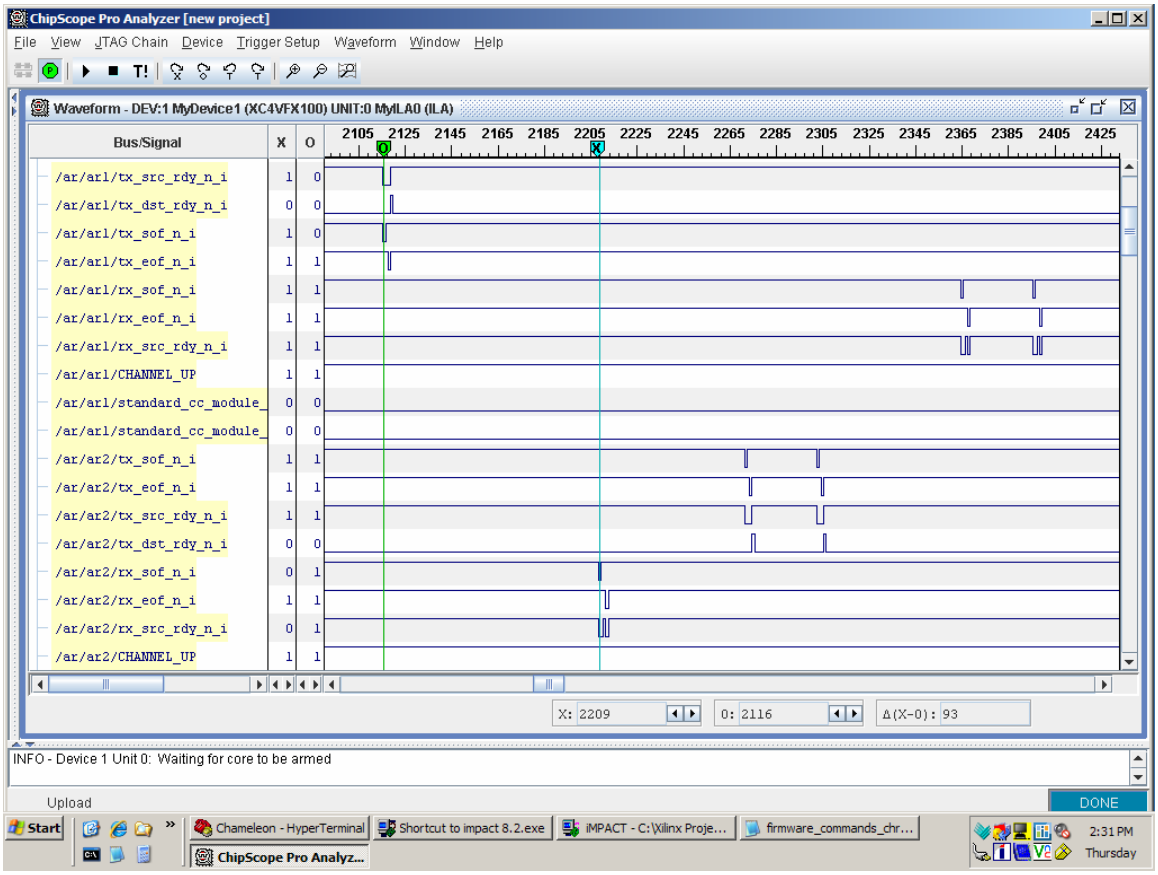

Figure 20: ChipScope showing Aurora latency

In this figure, the X and O markers are set at the beginning and the end of Aurora operation. O marks where one Aurora core begins sending a packet, and X marks where a second Aurora core receives a packet. This is 93 cycles, or 744ns. Adding the Aurora latency (744ns) and arbiter latency (608ns) yields a one way latency of 1352ns. Adding another Aurora latency (608ns) yields a total round trip of 1960ns, or about 2µs.

There are other small bumps in the graph. For the 0 hop case, these bumps are better seen in a logarithmic graph in Figure 21.

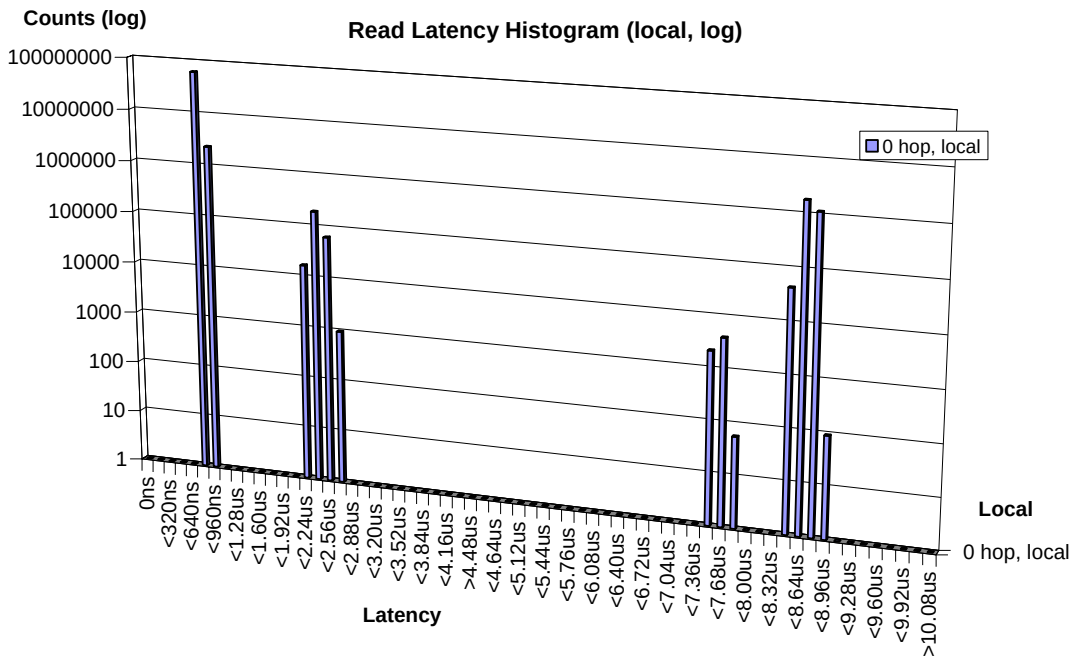

Figure 21: Local (0 hop) DDR read latency

For the 0 hop case, there is small bump at about 2.40µs. This is 1.6µs after the majority of the accesses, and consists of about 0.41%. The largest bar consists of 95.12% of the values. There are various reasons for the access to DDR to be delayed. One is a horizontal sync pulse. Horizontal sync occurs when the DVI controller is ready to read data for the next horizontal line from DDR. During this time, Aurora can not access DDR memory. Sometimes there is a horizontal sync pulse during a vertical sync pulse. Vertical sync occurs when the screen has finished being updated and DVI controller is getting ready to start over at the very top horizontal line. Multiple horizontal sync pulses occur and are not necessarily followed by an access to DDR, but they still block access to the DDR because a DDR access *could* follow. If the horizontal sync pulse drops and no DDR accesses occur, Aurora is free to access DDR. This short pause creates the observed bump.

There is a third bump in the graph at around 7.6µs for the 0 hop case, which comprises 0.0093% of the latencies. This is an extra 6.8µs of latency. More work needs

to be done to explain this. It is such a small percentage of the time that it is not considered important, and has little effect on the project.

The fourth bump in the 0 hop case is around 8.64µs, and comprises 4.459% of the latencies that occur. This extra 7.8µs is added due to a horizontal sync pulse that is followed by a DDR access from the DVI controller. An entire horizontal line is being read from DDR, which takes about 7.8µs. The horizontal sync (hsync) and DDR access (dvi_de_r) are shown in Figure 22.

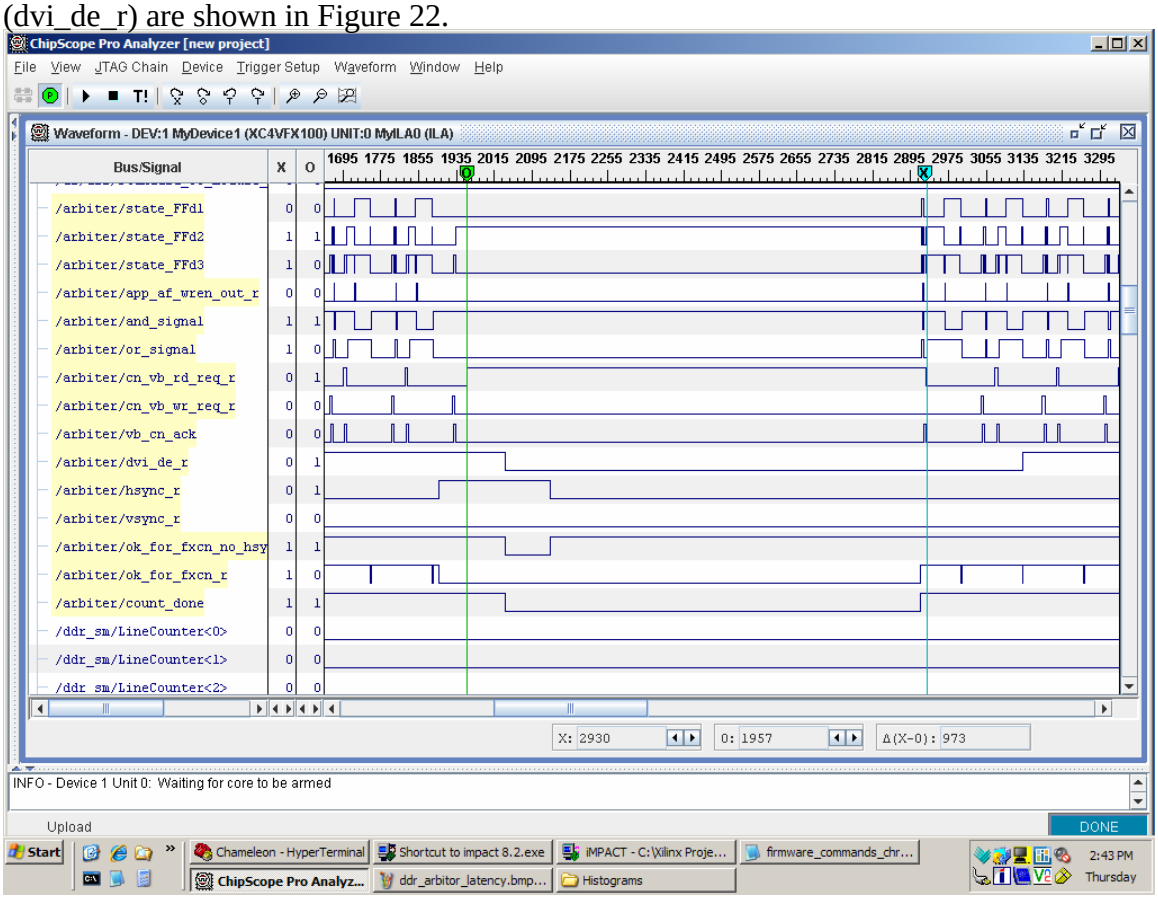

Figure 22: ChipScope showing horizontal sync and DDR access pulses

The horizontal sync pulse is from about 1887 to 2053 cycles, a difference of 166, or 1.33µs. The combination of the horizontal sync and the DDR access freezes the arbiter for about 973 cycles, which is 7.784µs.

The multiple hop graphs have various bumps towards the right end. Describing each one of these would be difficult. Their causes are a combination of reasons, such as these described above. The goal of this project is to build an efficient functioning system, thus these latencies could be optimized and reduced, possibly eliminating some of the bumps we see in the graphs. This is future work for this project.

5 Related Work

This section compares our shared memory system to some previous software and hardware shared memory (DSM) systems, along with some optical interconnect networks and other processor/FPGA systems. We discuss some previous DSM systems for background only: comparing performance of various systems is beyond the scope of this thesis. Also, the main contributions of these projects were not the underlying shared memory communication network, but the policies behind them (for example, coherence policies). Our system does not incorporate these mechanisms yet, so direct comparisons are not yet possible.

5.1 Software Shared-Memory

One of the first software based shared memory systems is IVY [13]. This system is implemented on a ring network of Apollo workstations, and shows how to create a DVSM (Distributed Virtual Shared Memory) system for loosely coupled multiprocessors. The IVY system requires a modification to the OS to implement the virtual memory. When a page fault occurs, the required page might come from disk or from another processor. This leverages the virtual memory system to let the software know when a page is not locally available. An invalidation based strategy provides coherence. One weakness of this scheme is that the smallest block of data that can be transferred is the OS's page size, which can be quite large. This leads to false sharing and extra network traffic.

Munin [14] is another DVSM system that attempts to reduce communication traffic over the network by using release consistency. Release consistency only requires memory to be consistent at synchronization points, resulting in fewer messages and lower overhead. A delayed update queue buffers and merges pending outgoing writes, which allows multiple writes to be sent in a single message.

A hardware/software hybrid shared memory approach, Simple COMA [15], is based on a COMA (Cache Only Memory Architecture) scheme. In COMA systems, data are automatically copied or moved to the physical memory of the node using them. The advantage over a CC-NUMA machine, is that the data are brought closer to the processor using them. One negative aspect is the extra hardware required, which increases the cost. Simple COMA attempts to reduce these costs by moving these extra hardware functions into software. Space allocation happens in software, but the coherence mechanisms are implemented in hardware. This gives Simple COMA the data migration and replication abilities of COMA, but uses a DVSM-like software memory management layer instead of complicated COMA hardware. An advantage over DVSM is that pages are divided into cache line sized blocks, and hardware maintains coherence for these blocks instead of software.

Another hybrid based approach, Cashmere [16], constructs a shared memory system using multiprocessor workstations as nodes and a Memory Channel [17] low latency remote write network. A memory region can be mapped into a process's address space for transmit, receive, or both. Memory Channel guarantees write ordering and local cache coherence. A page-based software scheme maintains coherence between nodes, and hardware maintains coherence within a node. All processors on a node share the same physical frame for a given data page. A form of lazy release consistency is used, and multiple concurrent writes are allowed. There is a home node in which all writes to a certain page write to, and the update in the home node is handled using diffs. Each page has a home node to which all writes get, and diffs control page updates.

5.2 Hardware Shared-Memory

Software shared memory schemes have some advantages, such as it is cheaper to design and create, it takes less time to build, the protocols we use can be more complex, and it is easier to customize. But they come at a cost in performance. Managing coherence in

software causes high processor overheads for handling remote requests, such as invalidations or updates. There is also overhead involved in the communication protocol for coherence messages. Handling a page fault in software can require much longer compared to hardware. When the OS is involved, the application loses execution time, and cached program data can be replaced with OS data.

The DASH [18] (Distributed Architecture for SHared memory) project represents one of the first hardware based distributed shared memory machines. One of the new key ideas is to distribute the shared memory space across different nodes, instead of using an apparently monolithic memory as in previous hardware shared memory systems. DASH utilizes existing hardware shared memory support within an SMP system to expand shared memory support to multiple SMP systems. DASH has hardware support for its directory based cache coherence, and supports release consistency.

Another hardware based shared memory scheme, SHRIMP [19], consists of a network of PC nodes connected by a custom network interface. A thin layer of software communicates with the network to create a Virtual Memory Mapped Communication (VMMC). Intel Mesh Routing Chips and the custom network interface connect each node to the routing backplane, and support the VMMC. VMMC allows applications to exchange data directly using virtual memory addresses. Receive buffers with settable access permissions are created when a process exports a region of its address space for other processes to write. One advantage of this system is that Automatic Updates (AUs) can be created given blocks of memory. If the data in such an address block changes, it is automatically updated in a remote block of memory, as well.

5.3 Interconnect Network

Myrinet [20] is a packet communication and switching technology used for high performance clustering. Myrinet includes flow control and error control, allows variable length packets, and supports many different message passing protocols. Myrinet also has

NICs that execute firmware to offload protocol processing from the processor. The firmware can bypass the OS and communicate directly with application processes for lower latency communication.

SCI [21] (Scalable Coherent Interface) is another interconnect technology that connects up to 64K nodes. The SCI nodes are connected using unidirectional links that create a ring. Switches connect multiple rings together. A node can transmit and receive packets concurrently. Packets contain commands, addresses, and data. Acknowledgments ensure correct packet transmission. SCI uses a virtually addressed distributed shared memory system, so all nodes can read and write the global memory. A distributed directory implements the cache coherence policy. Dolphin Interconnect Solutions currently produces PCI-E cards that contain SCI hardware. Both Myrinet and SCI support optical links for low latency communication.

5.4 FPGA Systems

There are many processor-FPGA boards available today. One of these is the Cray XD1 [22]. Each XD1 chassis (up to 12 in a cabinet) contains multiple processors and Xilinx FPGAs. The FPGAs are connected to the RapidArray fabric (the XD1's interconnect) so that they can be used as accelerators for a target application. Other examples of a processor/FPGA combination are the VMetro Phoenix FPGA/PowerPC systems [23]. For example, the Phoenix VPF1 system contains two PowerPC 7447A cores, two Xilinx Virtex II-Pro FPGAs, and high speed I/O. The board also contains DDR and flash memory, and will soon have the ability to boot Linux.

5.5 Comparison to This Work

There have been many hardware and software approaches to shared memory. While the hardware approaches are generally faster, the custom solutions are also more expensive. An important key to shared memory is the coherence protocol, which has yet to be

determined for the Chameleon Shared Memory project. We instead provide a thin architectural layer supporting shared memory. When comparing our system to the software shared memory systems, there is an obvious difference: ours is not a DVSM system. There are no interrupts involved in accessing shared memory, and the processor and OS are not involved. Our shared memory system is implemented at the word level, not the page level. The coherence protocol and how it will be implemented are future work. Other hardware shared memory systems are more rigid, our design is still customizable. We choose to focus first on the communication protocol, not a coherence policy. Therefore, our system is a flexible FPGA-based platform for experimentation with many different coherence mechanisms.

One of the differences between the Cray XD1 board and the Chameleon is that the interconnect protocol is fixed, but the Chameleon can use any protocol that can be implemented in the FPGA. The VMetro Phoenix system, with its FPGAs, DDR memory, and PowerPCs makes it similar to the Chameleon. In contrast, the Chameleon is a Cell Processor based system with XDRAM, which gives it more processing power. The Virtex-4 FPGAs are larger and have more capabilities than the Virtex-II Pro FPGAs, which allows a greater degree of freedom in our designs.

6 Conclusions

We have shown that it is possible to add hardware shared memory support to a system, independent of, and invisible to the processor. The processor simply accesses memory as if it is all local, and the control logic in the FPGAs and Aurora read from or write to remote memory if necessary. Using blades connected with a ring network allows us to add and remove blades dynamically. Future work involves adding a coherency mechanism to maintain data integrity between the multiple blades. This can be done in hardware so it does not slow down the CPU. There will be a much faster communication bus on the Chameleon soon, so that the connection between the Cell and the FX100 speeds up dramatically. Once this work is done, multithreaded applications will be able to run on the blades. The hypervisor can then be installed to create the single system image, with LibOS running on the slave blades.

The following is an example to illustrate the use of this system. We can have one blade running Linux, and designate this as the master blade. The other blades can run a limited OS, such as LibOS. A hypervisor will then make hardware resources of all blades visible to every blade in the system. The Linux OS image on the master blade is not responsible for managing hardware resources of slave blades, but an application running on this blade can send work to the slave blades and gather results. The hardware-based shared memory interconnect will be the physical communication path. Figure 23 shows a simplified view of this system.

Other improvements can be added as well, such as a network topology that scales better than a ring, and more advanced routing techniques. Although this work only involves four Chameleon blades with a basic network and routing techniques, it shows that with further work the system is capable of achieving its desired goals.

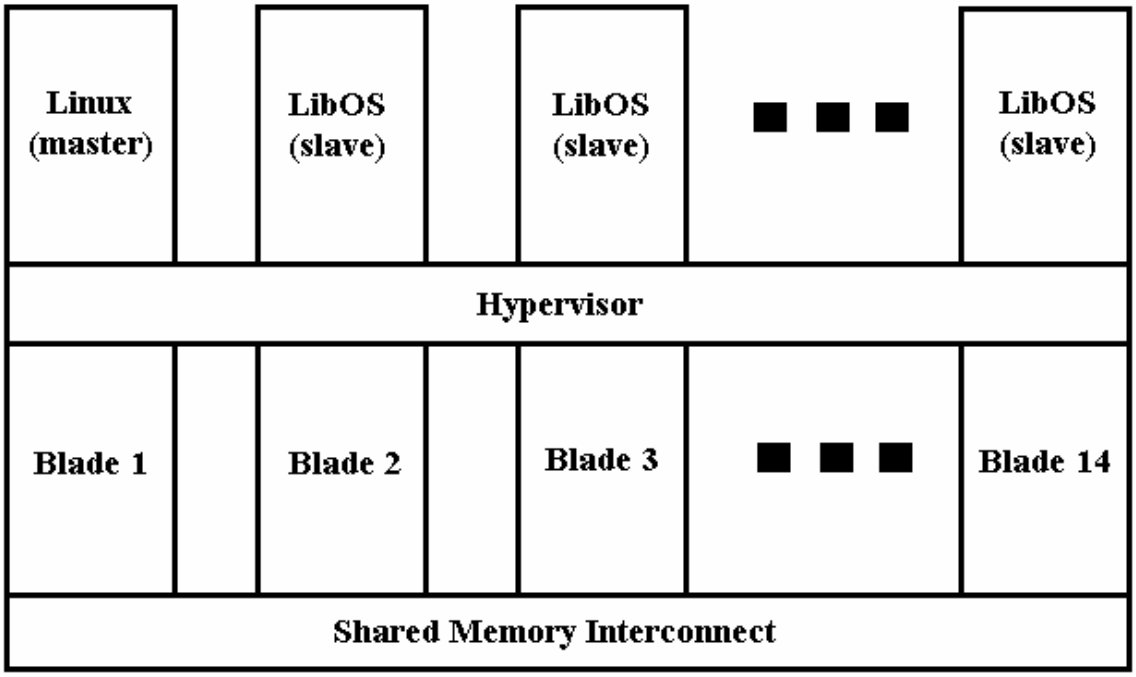

Figure 23: A multi-blade system using a hypervisor and LibOS

REFERENCES

[1] IBM Server Systems. See http://www.ibm.com/products/shop/servers/us/index.html

[2] Hewlett-Packard Server Systems. See http://welcome.hp.com/country/us/en/prodserv/servers.html

[3] Transaction Processing Performance Council. See http://www.tpc.org/

[4] Xilinx Aurora Protocol. See http://www.xilinx.com/products/design_resources/conn_central/grouping/aurora.htm

[5] Thomas Chen, Ram Raghavan, Jason Dale, Eiji Iwata. "Cell Broadband Engine Architecture and its First Implementation". *IBM developerWorks*, 29 Nov 2005. See http://www-128.ibm.com/developerworks/power/library/pa-cellperf/?ca=dnt-648

[6] Xilinx Virtex-4 FPGAs. See http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm

[7] Alan F. Benner, Michael Ignatowski, Jeffery Kash, Daniel Kuchta, Mark Ritter. "Exploitation of optical interconnects in future server architectures". *IBM Journal of Research and Development*, Volume 49, Number 4/5, pages 755-776, 2005.

[8] Xilinx LocalLink Interface. See http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?iLanguageID=1&category=-1212425&sGlobalNavPick=&sSecondaryNavPick=&key=LocalLink_UserInterface

[9] Xilinx ISE. See http://www.xilinx.com/ise/logic_design_prod/foundation.htm

[10] MTI ModelSim SE. See http://www.model.com/products/products_se.asp

[11] Xilinx ChipScope Pro. See http://www.xilinx.com/ise/optional_prod/cspro.htm

[12] FORTH programming language. See http://www.forth.com/forth/index.html

[13] Kai Li and Paul Hudak. “Memory Coherence in Shared Virtual Memory Systems”. *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pages 321-359, November 1989.

[14] John B. Carter, John K. Bennett, and Willy Zwaenepoel. “Implementation and Performance of Munin”. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152-164, October 1991.

[15] Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin. “An Argument for Simple COMA”. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture”*, pages 276-285, January 1995.

[16] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, Michael Scott, “CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network”. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 170-183, October 1997.

[17] Richard Gillett. “Memory Channel Network for PCI”. In *IEEE Micro*, pages 12-18, February 1996.

[18] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. “The DASH Prototype: Implementation and Performance”. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92-103, May 1992.

[19] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandburg. “A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer”. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142-153, April 1994.

[20] Myrinet. See http://www.myricom.com/myrinet/overview/

[21] IEEE Standard for Scalable Coherent Interface (SCI). The Institute of Electrical, Electronics Engineers, Inc. IEEE Std 1596-1992.

[22] Cray XD1. See http://www.cray.com/products/xd1/index.html

[23] VMetro. See http://www.vmetro.com/category1411.html