SCORES: A Scalable and Parametric Streams-Based Communication Architecture for Modular Reconfigurable Systems

Abelardo Jara-Berrocal and Ann Gordon-Ross
NSF Center for High-Performance Reconfigurable Computing (CHREC)
ECE Department, University of Florida, Gainesville, FL 32611
{berrocal, ann}@chrec.org

Abstract - **Parallel architectures have become an increasingly popular method in which to achieve high performance with low power consumption. In order to leverage these benefits, applications are decomposed into multiple computational modules (tasks) that collectively operate and communicate in parallel. In this paper, we present a scalable and highly parametric streams-based communication architecture for inter-module communication for FPGA-based systems – SCORES. This communication architecture improves on previous methods by providing increased application specialization and heterogeneous module clock frequencies, as well as providing a means for low latency communication and data throughput guarantees.**

I. Introduction and Motivation

One of the most important design considerations in VLSI digital systems is achieving high performance with low power consumption. Parallel architectures provide a popular method in which to achieve these goals [6]. Parallel architectures enable application decomposition into multiple computing modules, operating and communicating in parallel and which collectively encompass entire application behavior.

In order to exploit potential speedup as the number of modules increases, an efficient communication architecture to support inter-module communication is required. Traditional bus-based communication architectures enable all computing modules to communicate over a common shared bus, providing great flexibility for varying levels of inter-module communication requirements. However, bus-based communication architectures scale poorly as the number of computing modules increases [5]. This poor scalability results from shared transmission media contention and long routing wire delays, of which collectively imply long routing delays, reduced maximum clock frequency, and reduced throughput.

In order to alleviate some of the bus-based communication architecture drawbacks, a Network-on-Chip (NoC) architecture was proposed as a paradigm for scalable and parallel communication architectures [3]. NoCs are constructed as a topology of networked nodes connected by physical communication links. Each computing module is attached to this topology via a networked node. Depending on the communication scheme used, networked nodes can be classified as routers (packet switching) or switches (circuit switching).

NoCs appear to be a good alternative to bus-based communication. Network nodes provide breaks in the long routing wires inherent in a bus-based architecture, and therefore typically achieve higher operating frequencies. Furthermore, the main advantage of a NoC is scalability and specialization [11]. NoCs are commonly implemented as parametric HDL soft cores. Architectural parameters such as topology dimensions (width and height in number of routers), width and number of ports per node, and buffer depth enable architectural tuning to a specific target application [11].

NoC design was initially targeted for ASICs, although relatively recent research has extended these architectures to FPGAs [4][7][10][11][12][13]. However, FPGA-based designs commonly show four main limitations: high communication latency, no throughput guarantee, limited number of architectural parameters, and a significant area overhead. Furthermore, there is no standardized connection interface for computing modules, which forces computing module specialization to a specific NoC, reducing design reusability.

High communication latency and lack of throughput guarantee is a problem for some application domains. For example, DSP and image processing applications typically stream data between computing modules. Low latency and guaranteed throughput are required due to these system's real-time constraints. However, it is difficult for NoCs to meet these constraints, because NoCs were primarily designed for best effort delivery and not for quality of service guarantees.

In this paper, we present a novel **S**calable **CO**mmunication architecture for **RE**configurable **S**ystems (SCORES). SCORES utilizes a streams-based approach to transmit data between computing modules through dynamically established non-shared streaming channels. These channels enable low latency and guaranteed throughput. SCORES also features numerous tunable architectural parameters, offering increased application specialization compared to previous work. In addition, we present a simple, scalable, and highly parametric switch, which forms the basis for the SCORES communication architecture. Computation modules connect to switches using FIFO-based module interfaces, which enable heterogeneous module clock frequencies.

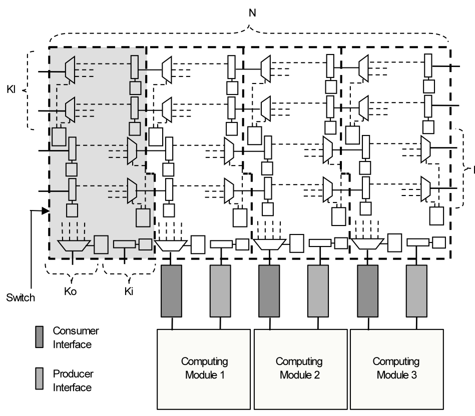

Fig. 1: Top SCORES communication architecture

II. Related Work

Marescaux et al. [7] presented the first working implementation of an FPGA-based NoC communication architecture built from *store and forward* (SF) routers. NoC topology was a 2D mesh and used deterministic XY routing [7]. Each router had five input ports and five output ports with 16 data bits per port. The SF technique introduced high transmission latencies due to packetization and queuing delays. Queuing delays were incurred due to the centralized arbiter inside each router to establish all input and output port connections. Additionally, the routers used one FIFO at each router input port for packet buffering. Because links between routers were time-shared between multiple simultaneous connections and these connections were dynamically established at runtime, throughput could not be guaranteed. A standalone router required 446 slices (4.8% of device usage) on the Xilinx Virtex XCV800.

Zerferino et al. presented SoCIN [12], a 2D mesh NoC built using *wormhole* routers. As opposed to SF routers that sent data messages as one large packet, wormhole routers split data messages into one header flit containing routing information and one or more smaller data flits for message transmission. If an output port was available, the header flit was routed and the remaining data flits followed in a pipelined method. When a message traversed the network, the message flits were located in multiple routers. Flits reduced FIFO memory requirements and transmission latency compared to packets. Modifiable architectural parameters included channel width and buffer depth. To reduce queuing delays, SoCIN replaced a centralized arbitration scheme with a distributed scheme by incorporating one arbiter per each router output port. However, SoCIN had high resource utilization for a standalone router. Each router accounted for as much as 11% of a Xilinx V2PRO V2P30 [10].

Sethuraman et al. designed a 2D NoC using an SF router called LiPaR [10]. Modifiable architectural parameters included channel width and buffer depth. The NoC used the FIFOs' empty flags in order to govern transmission synchronization inside and between the routers, which significantly reduced control logic complexity. However, the router had a complex cross point switch matrix. The router ran at 33.33 MHz and required 352 slices (2.7% of device usage) on a Xilinx Virtex 2 Pro V2P30 for a minimum sized channel width of 8 bits (excluding FIFO buffer resources).

Sedcole et al. presented a bus-based communication architecture called Sonic-on-a-Chip [9] for reconfigurable image processing systems, which leveraged NoC concepts. Dedicated communication streaming channels between computing modules were dynamically established by allocating frame slots inside the time-multiplexed bus. System performance was comparatively slow to NoC-based designs (50 MHz for V2Pro) due to the long routing wires inherent in a bus-based architecture.

Ahmadinia et al. proposed the RMBoC (reconfigurable multiple bus on-chip) architecture [1]. Communication links connected switches in a linear array. Architectural parameters included number and width of communication links. Switches had only one input and one output port for module connections. Switches used a centralized FIFO and arbiter to receive all connection requests. Switches dynamically established dedicated communication paths between modules, although, the architecture did not leverage flow control, which is problematic when a module exhausts buffer memory. Finally, the modules and communication architecture were required to operate at the same clock frequency. RMBoC achieved an operating frequency of 99 MHz and required 3407 slices (10% of device usage) on a Virtex II 6000 for a design with four switches and four communication links between switches.

III. SCORES Architecture

Fig. 1 depicts the top-level design of the SCORES communication architecture. SCORES is composed of a linear array of switches (one switch is highlighted in gray shading). Each switch has a unique X coordinate indicating its horizontal position inside the linear array. Switches communicate with neighboring switches (*Kl* and *Kr*) and computing module interfaces (*Ki* and *Ko*) through bidirectional communication links between their input and output ports.

Computing modules attach to switches through two types of module interfaces. *Consumer Interfaces* connect a computing module's input port to a switch's local output port (*Ko*). *Producer Interfaces* connect a computing module's output port to a switch's local input port (*Ki*).

Dynamically established *Dedicated Streaming Routes* (DSRs) enable data transmission between two computing modules. These dedicated routes provide high throughput and low latency data transmission. For each DSR, we refer to the *producer* as the module sending data and the *consumer* as the module receiving data. A computing module can be both a *producer* and *consumer* simultaneously.

A. Architectural Parameters

SCORES is a highly parametric communication architecture. Our design offers six tunable architectural parameters: *N*, *W*, *Kr*, *Kl*, *Ki*, and *Ko* (Fig. 1). *N* represents the number of switches in the linear array. *W* is the width of the communication links and switch input and output ports. *Ki*, *Ko*, *Kr*, and *Kl* represent the number of local input ports, local output ports, right output and left input ports, and left output and right input ports, respectively, for each switch. Thus, a switch has *Kl* and *Kr* communication links to the left and right neighboring switches, respectively.

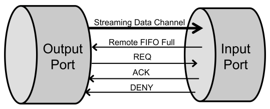

Fig. 2: SCORES communication link

B. Communication Links

Communication links, illustrated in Fig. 2, connect a switch with neighboring switches and computing modules. The link consists of two opposite flowing data channels and three handshaking signals.

The *Streaming Data Channel* (*SDC*) is the main channel and transmits data from a switch or computing module output port to the connected input port. The two most significant bits (MSBs) of the *SDC* are reserved for signaling. The MSB is the *Write Enable* (*WR_EN*) and indicates that the producer is transmitting a word. The second MSB is the *End of Stream* (*EOS*) and indicates that the producer has completed data transmission and that the *DSR* can be released. The remaining W-2 least significant bits (LSBs) of the *SDC* carry data. The *Stream Feedback Channel* (*SFC*) is a single signal, *Remote FIFO Full*, which indicates that the consumer FIFO is full, and therefore the producer must pause data transmission. The handshaking signals (*REQ* and *ACK*) establish and release a DSR. The *DENY* signal will be explained in section IV-IV.C.

C. DSR Addressing and Communication Protocol

When a producer requests DSR establishment with a target consumer interface, the producer writes an *Address Header* to the *SDC* of its producer interface. The *Address Header* is composed of two fields, an X coordinate and a local identifier. The *X coordinate* indicates the horizontal location of the target switch connected to the consumer interface. The *local identifier* indicates the specific local output port to use between the target switch and the consumer interface. Use of a *local identifier* enables computing modules to separate different data stream types to different input ports. *X coordinate* and *local identifier* widths depends on the *N* and *Ko* architectural parameters

After the producer interface receives the *Address Header*, the producer interface writes the *Address Header* to the connected switch's input port and asserts *REQ*. The switch selects an arbitrary left or right (direction determined by the *X coordinate* field) output port that is not already assigned to a DSR and forwards the *Address Header* and asserts *REQ* on this output port. The connection between the input and output port is now reserved for this DSR. This similar process repeats as the *Address Header* propagates through neighboring switches until the *Address Header* reaches the target switch, in which case the *Address Header* is forwarded to the target consumer interface.

When the target consumer interface receives a *REQ*, the consumer interface enters an *Established Connection State* and replies with a positive *ACK*. This *ACK* propagates through the switch array back to the producer interface, traversing the reserved input/output port connections at each switch. When the producer interface receives the asserted *ACK*, a DSR has been established between the producer and consumer interfaces.

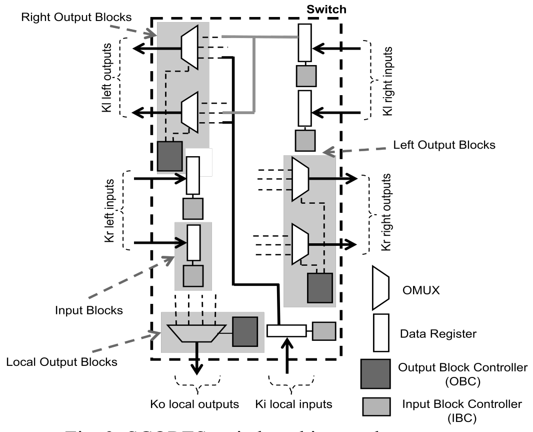

Fig. 3: SCORES switch architectural components

After DSR establishment, data can be transmitted between the producer and the consumer as a continuous low latency pipelined stream because our switch design uses only one register at each input port instead of a large, high latency FIFO. The DSR remains established as long as the producer interface asserts *REQ*. A producer interface deasserts *REQ* when the producer interface detects assertion of the *EOS* flag from the producer module.

IV. Switch Architecture

Fig. 3 depicts the block level diagram of the SCORES switch architecture. The switch uses distributed arbitration and contains two main block types: input blocks and output blocks. Input and output blocks enable data to flow into and out of the switch, respectively. These blocks encapsulate and manage a switch's input and output ports. External connections between neighboring switches input and output blocks, and internal connections between input and output blocks collectively enable inter-module communication.

A. Output Blocks

Output blocks are units responsible for controlling switch output ports. Output blocks are classified into three different types: left, right, and local output blocks. Left and right output blocks are responsible for all left and right output port management for the switch, respectively. To enable horizontal data transmission through the linear switch array, a switch's left output blocks are connected to neighboring switch's right input blocks (Section IV-B), and vice versa. To enable internal data transmission through a switch, left output blocks are internally connected to right input blocks (Section IV-B), and vice versa. Each switch has only one left and one right output block, but each block can be connected to multiple output ports. Local output blocks are responsible for local output port management, which connect the switch to computing module interfaces.

Left and right output blocks are composed of three main units: a set of *Remote FIFO Full* registers, a set of *Output Multiplexers* (*OMUX*), and one *Output Block Controller* (*OBC*). Fig. 4 illustrates these units. There is one *Remote FIFO Full* register and one *OMUX* associated with each output port.

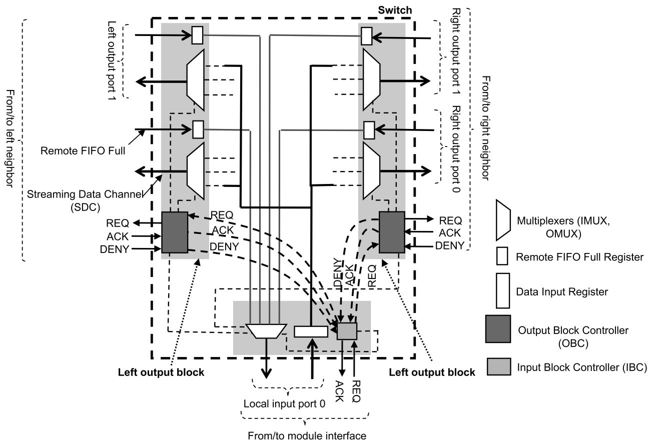

Fig. 4: SCORES switch showing internal connections between a local input block and the left, right, and local output blocks

These registers latch the *Remote FIFO Full* signals coming from the respective output ports. The *OMUX* inputs are the outputs of the *Data Input Registers* (described in Section IV-B) from the internally connected input blocks. *OMUX* outputs drive the output port's *SDC*.

To establish internal connections, the *OBC* is an FSM-based controller that receives requests on input blocks for output port assignment. Requests are serviced using round robin arbitration (RRA). A *Service Table* records state (available or occupied) and assignment (which input block has been assigned which output port) for each output port. When the *OBC* services a request, the *OBC* checks the service table for an available output port. If an output port is available, the *OBC* configures the output port's *OMUX* select lines to set the input data source as the *Data Input Register* from the requesting input block and records this connection in the service table. Finally, the *OBC* asserts *ACK* to the requesting input block. If no output port is available, the *OBC* replies with an asserted *DENY* signal to the requesting *IBC*. We refer to this condition as a *blocking condition.*

Local output blocks are similar to the left and right output blocks, but differ in that there is one local output port per local output block because each output port has a unique local identifier. Thus, each switch may have multiple local output blocks. A service table is not required for local output blocks since the local output blocks manage only one output port.

B. Input Blocks

Input blocks are similar to output blocks in that there are three types of input blocks: left, right, and local input blocks. Each block type serves a similar purpose as the associated output block type described in Section IV-A. In contrast to left and right output blocks, there is one input block for each input port. Depending on the input block type, input blocks are connected with a subset of output blocks. Left input blocks are internally connected to the right output block and to all the local output blocks. Right input blocks are internally connected to the left output block and to all the local output blocks. Finally, each local input block is connected to both the left and right output blocks. Fig. 4 illustrates these connections.

Input blocks are composed of three main units: one *Data Input Register* (*DIR*), one *Input Multiplexer* (*IMUX*), and one *Input Block Controller* (*IBC*). The *DIR* is *W*-bits wide and latches data from the *SDC* connected to the switch's input port on each clock cycle. The *IMUX* selects the source for the *Remote FIFO Full* output at the input port. *IMUX* inputs are the *Remote FIFO Full* registers at the connected output blocks. The *IBC* is an FSM-based controller, which initially waits for assertion of an input port *REQ* from a neighboring switch or module interface. Upon *REQ* assertion, the *IBC* reads the value at the *DIR* (which contains the *Address Header*) and compares it with the X coordinate of the switch to determine an appropriate action.

If the switch's X coordinate is greater than the *Address Header's* target *X Coordinate* field*,* then the *IBC* forwards the *REQ* to the left *OBC*. If the switch's X coordinate is smaller than the *Address Header's* target *X Coordinate* field, then the *IBC* forwards the *REQ* to the right *OBC*. If the switch's X coordinate equals the *Address Header's* target *X Coordinate* field, then the *IBC* sends the request to the *OBC* at the local output port indicated by the *local identifier* in the *Address Header* field. We refer to this routing scheme as *X Routing*.

Once the *IBC* receives an *ACK* signal from the requested *OBC*, the *IBC* sets the control bits of its *IMUX* to select the *Remote FIFO Full* coming from the assigned output port. However, since the left and right *OBCs* manage several output ports, it is necessary to know from which specific output port the *Remote FIFO Full* originates. Therefore, the *IMUX* receives select lines from the *IBC* and the left and right *OBCs*. *IMUXes* do not receive select lines from the local output blocks, because local output blocks only manage one output port.

C. Blocking Conditions

It is possible that the number of input blocks requesting service from an *OBC* exceeds the number of available output ports. In this situation, the *OBC* responds with an asserted *DENY* signal to all the *IBCs* to which it cannot yet assign an output port. The *DENY* signal backward propagates through the partially established communication path to the source producer module from which the *REQ* was generated. Upon receiving a *DENY* signal, the producer module can either hold the asserted *REQ* signal (*persistent request*) or deassert the *REQ* signal for a later retransmission attempt. Deasserting the *REQ* signal also releases the partially established communication path.

If a large number of source producer modules which receive a *DENY* signal do not deassert their *REQs*, excess input blocks are still queued at the saturated *OBC* waiting for an available output port, and thus impose queuing delays. Blocking condition delays are critical to the left and right output blocks since the majority of inter-module communication passes through the left and right output ports. In SCORES, increasing the *Kr* and *Kl* architectural parameters (the number of output ports) can reduce blocking conditions.

V. COMPUTING MODULE INTERFACES

Computing module interfaces connect computing modules to a SCORES switch. These module interfaces are based on dual-clocked FIFOs. These FIFOs buffer data and enable clock domain isolation between the communication architecture and the computing modules. By separating clock domains, each

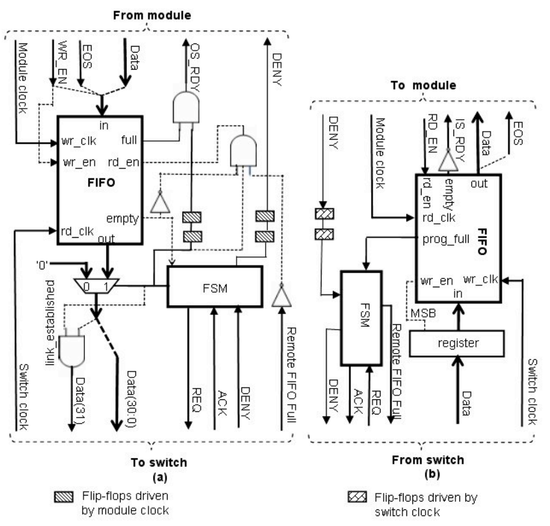

Fig. 5: Module interfaces: (a) producer interface, (b) consumer interface.

computing module can run at an independent and optimized clock frequency.

We created FIFOs using the Xilinx Coregen FIFO Generator 4.3 [15]. This tool enables customization of both FIFO depth and width (of which the width was set to match the switch's *W* architectural parameter). Xilinx Coregen tool allows FIFOs to be implemented using distributed memory or embedded BlockRAMs.

A. Producer Module Interfaces

Fig. 5 (a) illustrates the detailed producer module interface architecture. Signals between the module output port and the producer interface are: *Data_in*, *WR_EN* (Write enable), *OS_RDY* (Output Stream Ready), *EOS* (End of Stream), and *DENY*. The FIFO's data input (*in*) and output (*out*) are the combination of *data_in*, *EOS*, and *WR_EN*. The producer module asserts *WR_EN* to begin data transmission to the producer interface. The producer module asserts *EOS* when data transmission has completed, allowing the producer module interface to release the DSR.

Initially, the producer interface controller (*FSM*) waits for an *Address Header* on *data_in* from the producer module to begin DSR establishment. During this initial state, the FIFO *empty* and *full* signals are asserted and deasserted, respectively. When the producer module transmits the *Address Header*, the FIFO stores *data_in* and deasserts *empty*. The FIFO is synthesized using the first word fall through feature, which enables the *Address Header* to be available on *data_out* without a read operation (eliminating read delay). Upon deassertion of *empty*, the *FSM* asserts *REQ* to the switch's local input port. Upon *ACK* assertion from the switch's local input port, the *FSM* asserts the *link_established* signal, which indicates DSR establishment.

The *FSM* asserts *OS_RDY* when two conditions are satisfied: (1) the FIFO has available space (*full* is not asserted) and (2) the DSR is established (*link_established* is asserted). Since data transmission is not ready until assertion of both *link_established* and *WR_EN* (MSB of *out*), these signals are AND'ed and serve as the producer interface's *data_out* output to the connected switch.

After a DSR is established, the producer interface continues writing data from *out* to the switch's local input port on *data_out* while FIFO is not empty and a *Remote FIFO Full* feedback flag has not been received from the switch. Finally, upon detecting *EOS* assertion from the producer module, the *FSM* deasserts *REQ*, which initiates the release of the DSR.

B. Consumer Module Interfaces

Fig. 5 (b) illustrates the detailed consumer module interface architecture. Signals between a module input port and the consumer interface are: *data_in*, *RD_EN* (Read Enable), *IS_RDY* (Input Stream Ready), and *EOS* (End of Stream). The module asserts *RD_EN* to enable reading from the FIFO. The module interface asserts *IS_RDY* when the FIFO contains data waiting to be read. Since the MSB of *data_in* indicates *WR_EN*, *data_in* coming from the switch's local output port is written into the FIFO if *WR_EN* is asserted.

Consumer interfaces are responsible for asserting *Remote FIFO Full* for DSR flow control. A consumer interface must assert *Remote FIFO Full* before the FIFO reaches maximum capacity due to in-flight data and the *Remote FIFO Full* propagation delay through the switch array. Therefore, the consumer interface must take into consideration its X location (*Xc*) and the producer interface's X location (*Xp*). Thus, the consumer interface asserts *Remote FIFO Full* when the remaining space in the FIFO is equal to $2(N-|Xc-Xp|)$.

VI. RESULTS

A. Experimental Setup

We implemented our SCORES communication architecture, switch, and module interfaces as highly parametric VHDL soft cores, providing the architectural parameters: *N*, *W*, *Kr*, *Kl*, *Ki*, and *Ko*. FIFOs, implemented using one embedded BlockRAM, stored 512 32-bit words. The target device was a Virtex 4 XC4VLX25 [16] and system simulation was performed using Modelsim 6.2 SE [8].

Given the massive configurability of SCORES due to the numerous architectural parameters, we wrote a Perl script to execute Xilinx synthesis and implementation tools (ISE 10.1) [14] for a standalone switch of varying configurations. These configurations enabled architectural parameter impact evaluation on selected performance metrics such as slice utilization and maximum clock frequency. In a real scenario, *Kr*, *Kl*, *Ki* and *Ko* would be specialized to the target application. We measured maximum clock frequency after place-and-route using the Xilinx Trace static timing analysis tool with no clock constraint (*trce –a –u*).

B. Area Usage and Timing Analysis

Fig. 6 shows area usage in slices (top row) and maximum attainable clock frequency (bottom row) versus varying architectural parameters for channel widths W = 8, 16, 32, and 64 bits for a single switch. The first, second, and third columns vary *Kr, Kr* and *Kl,* and *Ki* and *Ko,* respectively. We consider the case in which only *Kr* is varied to account for applications in which most data flow occurs in only one direction such as

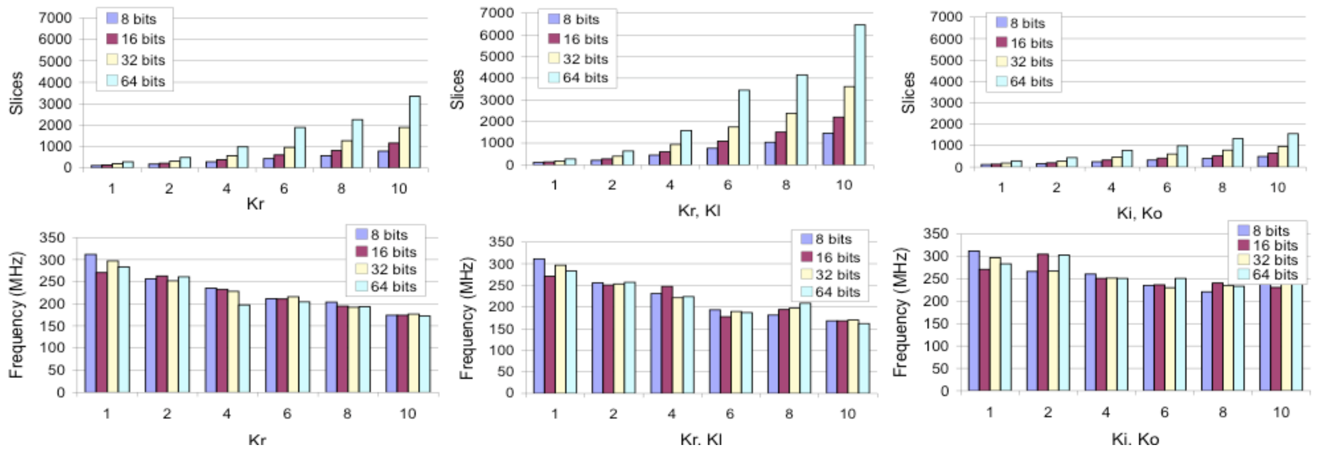

Fig. 6: Area in slices (top row) and maximum clock frequency (bottom row) verses varying architectural parameters for channel widths W = 8, 16, 32, and 64 bits.

DSP or image processing. Fig. 6 (top row) shows low switch area overhead, which scales well due to a small cross point matrix (composed of *OMUXes* and *IMUXes*). For example, the area overhead for a sample system configuration, $W = 32$, $Kr = 2$, $Kl = 2$, $Ki = 1$, and $Ko = 1$, is only 399 slices, which accounts for 1.62% of the XC4VLX25. Doubling the channel width from $W = 32$ to $W = 64$ (with the same system configuration $Kr = 2$, $Kl = 2$, $Ki = 1$, and $Ko = 1$) increases slice usage by only 60% to only 639 slices, revealing a sublinear increase in area verses channel width.

Maximum clock frequency is a very important metric since it determines the maximum data throughput achievable by SCORES. Given a data word of length W, (with the two MSBs reserved for *WR_EN* and *EOS*) the peak data throughput in Gbps for SCORES is:

$$\text{data\_throughput (Gbps)} = (W - 2) * \text{max\_frequency}$$

Fig. 6 (bottom row) shows that for all test configurations, the operating frequency ranges from 161 MHz to 311 MHz. Therefore, the data throughput peaks at 161*(32-2) = 4.8 Gbps for an *SDC* width of W = 32 bits, which is competitive with previous work.

VII. Conclusions and Future Work

In this paper, we introduce a novel highly parametric, **S**calable **CO**mmunication architecture for **RE**configurable **S**ystems (SCORES). SCORES consists of highly parametric VHDL soft-core switches with distributed arbiters, which reduces the inefficiencies associated with centralized arbiters. SCORES enables runtime establishment of simultaneous and dedicated streaming communication links between computing modules. Results show that SCORES is highly scalable in terms of area overhead and can achieve high operating frequencies even for large systems (high number of computing modules).

Future work includes a pipelined implementation of the output block controller (*OBC*) to decrease connection establishment time. We also plan to expand to a 2D mesh in order to reduce blocking conditions and increase design flexibility. In addition, we also plan to develop an optimization technique for automatic computing module placement and architectural parameter sizing based on an application's connectivity graph.

Acknowledgements

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. We gratefully acknowledge tools provided by Xilinx.

References

[1] A. Ahmadinia, C. Bobda, J. Ding, M. Majar, J. Teich, S. Fekete, and J. Veen. A practical approach for circuit routing on dynamic reconfigurable devices. In *RSP* 2005, pages 84–90

[2] Altera Inc., http://www.altera.com, 2008

[3] L. Benini and G. De Micheli. Networks on chips: A new SOC paradigm. In *IEEE Computer*, pages 70-78, 2002

[4] C. Bobda and A. Ahmadinia, Dynamic interconnection of reconfigurable modules on reconfigurable devices. *IEEE Design & Test of Computers*, vol. 22, no. 5, pages 443–451, 2005

[5] W.J. Dally and B. Towles. Route packets, not wires: on-chip inter-connection networks. In *DATE* 2001, pages 684–689

[6] International Technology Roadmap for Semiconductors 2007. In http://public.itrs.net, 2007

[7] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. Interconnection networks enable fine-grain dynamic multi-tasking on FPGAs. In *FPL 2002*, pages 795-805

[8] Mentor Graphics, http://www.mentor.com, 2008

[9] P. Sedcole, P. Y. K. Cheung, G. A. Constantinides and W. Luk, Run-time integration of reconfigurable video processing systems. In *IEEE Transactions on VLSI Systems*, Vol. 15, No. 9, , pp 1003-1016, 2007

[10] B. Sethuraman, P. Bhattacharya, J. Khan, and R. Vemuri. LiPaR: A light-weight parallel router for FPGA-based networks-on-chip. In *GLSVLSI* 2005

[11] D. Wang, H. Matsutani, M. Yoshimi, M. Koibuchi, and H. Amano. A parametric study of scalable interconnects on FPGAs. In *ERSA* 2006, pages 130–135

[12] C.A. Zerferino and A.A. Susin. SoCIN: A parametric and scalable network on chip. In *SBCCI* 2003, pages 169-174

[13] C.A. Zeferino, M. E. Kreutz, and A.A. Susin. RASoC: A router soft-core for networks-on-chip. In *DATE* 2004.

[14] Xilinx Inc., http://www.xilinx.com, 2008

[15] Xilinx. FIFO Generator 4.3 Datasheet. DS317. March 24, 2008

[16] Xilinx Virtex 4 User Guide. UG070 v2.6. December 1, 2008