Active SANs: Hardware Support for Integrating Computation and Communication

Craig Ulmer, Chris Wood, and Sudhakar Yalamanchili

Abstract-- **This paper explores the view that the SAN network infrastructure can be an active computational entity capable of supporting certain classes of data intensive computations effectively during communication. The performance is achieved via the use of Field Programmable Gate Arrays (FPGAs) in the network interfaces (NIs). This paper describes the programming model and the design of a prototype hardware/software implementation using commercial FPGA devices coupled with Myrinet. An active messages style of programming is used to support application-transparent, dynamic reconfiguration of the FPGA hardware to accommodate different computations over time. Performance evaluation of this implementation quantifies the overheads and sources of performance improvement.**

I. INTRODUCTION

Traditionally System Area Networks (SANs) have been focused on the cost-effective delivery of bandwidth and latency to cluster computing applications. In recent years we have seen the inclusion of embedded microprocessors in network interfaces (NIs) and as a result the migration of communication functionality to these interfaces. In addition to off-loading the host CPU these intelligent network interfaces have supported an active message style of communication for a variety of remote and local communication services. However, in this model the SAN still largely remains a data transport entity.

The goal of the Active System Area Network (ASAN) project at Georgia Tech is to examine ways in which computation can be "tightly" integrated with communication. A very simple yet motivating example is the process of computing checksums on packets. In most systems this computation is not performed in place but rather during the transfer of packet data from one point in the system to another, for example, during transfer from the NI to the wire. Functions such as data encryption and data compression also naturally fit this model. Many SAN applications transfer large sets or streams of data between endpoints. Algorithmic transformations on such data sets can often be formulated as pipelined or streaming computations. Such streaming computations can be performed during the transfer of data to and from the network in the NI. Many network related services such as firewalls, intrusion detection, or denial of service policies can also be formulated as computations on streams and can be effectively performed close to the wire. Our goal is to study and implement a model of computation where such stream oriented computations can be integrated with communications and performed "in transit", i.e., with minimal impact on latency. The use of such enhanced NIs produce system area networks that we refer to as Active SANs since computations can be performed on the host CPUs or in the NIs during transmission.

The challenges in constructing active SANs are the following.

- The availability of sufficient computing power in the network: Embedded CPUs will be overwhelmed by data intensive computations. We address this with the

C. Ulmer, C. Wood, and S. Yalamanchili are with the Center for Experimental Research in Computer Systems, in the School of Electrical and Computer Engineering at the Georgia Institute of Technology, Atlanta, GA, 30332-0250 USA (e-mail: ulmer@ece.gatech.edu).

placement of field programmable gate arrays (FPGAs) in or near the network interface (NI) as a host for hardware implementations of data intensive, streaming computations.

- Programming model: The challenge is a programming model that enables users to develop applications that can effectively make use of this capability. We are constructing a connection-oriented model of communication where computations can be associated with connections and dynamically placed in the NI.
- Dynamic customization: The hardware implementations that are "placed" or "programmed" into the interface must be changed over short periods of time. A methodology and support infrastructure is necessary for application-specific, or even connection-specific customizations to be created in a demand-driven manner. The interface must support run-time reconfiguration of the FPGAs.

Proof of concept for this work is established by constructing a test bench using commercially available network and FPGA components. The most general system model permits FPGAs and CPUs to be distributed throughout the cluster and be utilized as a pool of resources. This paper focuses on all aspects concerned with the construction of a single active connection. Composition of these connections to utilize network wide resources can be achieved by extending the infrastructure described in this paper.

II. Related Work

Multiple research projects have explored the use of FPGAs in network environments. A significant portion of this work comes from the field of active networks [1] where routers perform computations on in-transit messages. In [2] researchers utilized FPGAs to provide selectable error correction services in an ATM network with lossy links. Researchers at Washington University extended this work with a prototype ATM router that features an FPGA at each router port [3]. This work additionally discusses a potential API and framework for user-defined hardware modules. Another reconfigurable router is prototyped in [4], with stream-based computations performed on IP packets.

While router-based active networking follows the same fundamental concepts as those presented in this paper, there are at least two major differences. First, active networking projects typically operate assuming a LAN context and seek to increase performance or available services for traditional network protocols. Our work is instead oriented on improving cluster computer performance for SANs. Second, router-based processing is a different design space than NI-based processing. The operating environment significantly influences the construction of key operations such as dynamic reconfiguration. In NI-based FPGA processing the host-CPU of a node is a natural candidate for managing reconfiguration. In router-based architectures an external host or on-board CPU must be utilized to manage configurations.

Another group of related research projects deal with FPGA-NI interactions in the local host. In [5] researchers place FPGA, DSP, and ATM cards in a host to enhance node performance. This work describes the utilization of resources but does not focus on using accelerator hardware beyond the local context. Perhaps the most relevant research work is the adaptive computing systems project [6] at Virginia Tech. This work builds a custom computing machine (CCM) from a cluster equipped with various FPGA and network equipment. An API is defined that allows host applications to utilize a series of FPGAs distributed throughout the cluster through calls to the message passing interface (MPI) communication library [7]. While this API creates a usable hardware abstraction, we note that messages are relayed through the host's MPI library before delivery to end FPGAs. Our focus is on making the FPGA a part of the NI and support an abstraction of the network as computational entity wherein peripheral devices have access to the same capabilities as the host CPUs.

III. RECONFIGURABLE HARDWARE

Reconfigurable hardware is a programmable logic device (PLD). An example of such a device is a Field Programmable Gate Array (FPGA). These devices come with a large array of configurable logic blocks (Xilinx terminology) and an extensive configurable routing framework for interconnecting logic blocks. Each logic block contains memories and flip-flops. The memories are configured as look-up tables to implement the truth tables corresponding to small blocks of combinational logic while the flip-flops are used to configure the sequential components of the design. Design tools take gate-level designs and map these circuits to the chip. By configuring the contents of the look-up tables and appropriately configuring the interconnect, the gate-level circuit is realized. It is apparent that these configurations can be changed to implement a variety of hardware designs. While the clock speeds do not compare to those that can be achieved with application-specific integrated circuits (ASICs), one can obtain very high speed implementations of functions relative to that which is feasible with software implementations.

While industrial work with reconfigurable hardware has primarily utilized the technology for verification of and sometimes a replacement for ASICs, researchers have explored the use of FPGAs as a means of improving computational performance in scientific and multimedia applications. The appeal of this technology for researchers is that an FPGA can be dynamically configured with customized computational circuits that provide hardware acceleration for end applications. Unlike single-purpose ASICs, FPGAs can be reconfigured and re-used for a number of diverse computational tasks.

In the last decade a number of research projects have provided details as to what types of applications can benefit from the use of FPGAs. Suitable applications often contain a combination of four characteristics. First, computations based on non-standard (e.g., a non-power of two) or variable bit-widths often perform well because an implementation in FPGA hardware can be tailored to the bit-width. For example, an FPGA circuit can be configured with a 17-bit adder or a 256 bit rotating register. Second, FPGA circuits can be customized to meet the conditions of use. For example, in partial evaluation a priori information such as data constants is used to reduce circuitry for a computation, thereby increasing its speed. Finally, the large amount of programmable logic available in an FPGA provides a substrate where numerous computational engines can operate concurrently in a single FPGA. Although typically more expensive and slower than custom ASICs, the FPGAs can make up ground in their ability to be reconfigured across applications as opposed to the single functionality of ASICs.

Modern commercial FPGA architectures emulate up to 8 million logic gates in a single chip with clock speeds reaching up to 400 MHz [8]. In practice, emerging FPGAs can be expected to house multi-million gate designs running at clock speeds operating in excess of 100 MHz.

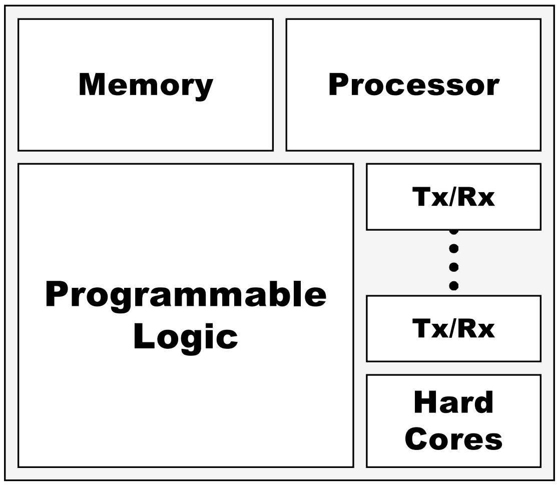

Figure 1: Emerging FPGA Architecture

As illustrated in Figure 1, emerging commercial FPGA architectures are supplementing traditional programmable logic with multiple hardware units for increased design flexibility and computational power. The most attractive feature in these new architectures for the SAN community is the inclusion of high-speed transceivers. Upcoming versions of Xilinx's Virtex II architecture [9] will contain multiple 3.125 Gbps transceivers for use with networks such as InfiniBand [10], 10-Gbps Ethernet, and OC-192c. A second feature of emerging FPGA architectures is the inclusion of powerful embedded processor cores. Altera's Excalibur FPGA includes a dedicated 32-bit 200 MHz ARM core [11]. Likewise Xilinx plans include future Virtex II architectures that contain multiple PowerPC cores. Finally, modern FPGA architectures contain dedicated hardware such as internal SRAM memory and high-speed multiplier array cores to accelerate FPGA computational performance.

Given the recent advances in commercial FPGA architectures, it is reasonable to expect that future high-performance network interface cards can be constructed using a single FPGA chip that features reconfigurable hardware, multi-gigabit-per-second transceivers, and a dedicated embedded processor. A hypothetical future FPGA-based NI card is illustrated in Figure 2, and is comprised of three components. First, input and output message queues are utilized to buffer messages between endpoints and the network in times of heavy traffic. Second, a network interface is implemented through a combination of reconfigurable hardware and software for the FPGA's embedded processor. Third, the remaining reconfigurable hardware space is utilized to house multiple computational circuits for processing in-transit messages.

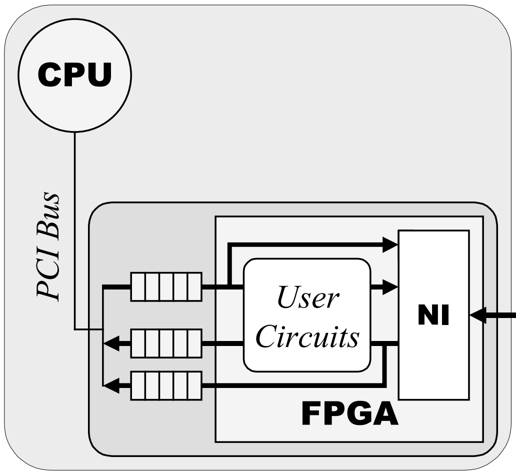

Figure 2: Future Host with FPGA-Based NI

IV. PROGRAMMING MODEL

The basic programming model is an extension of connection-oriented communication between host CPUs in a SAN cluster. We extend this model by enabling the association of a function with each connection. This function is applied to the data stream in transit and is implemented in the FPGA. In general, functions can be associated with the data stream at the sender or receiver. The connection abstraction is not strictly necessary - functions can be simply applied to the data in transit. However there are some practical advantages in doing so. The notion of the state of a connection can be extended to include the FPGA configuration. Setting up and removing connections can be expensive operations as is the relative time required for configuring and reconfiguring of the FPGA

A function applied to a data stream is implemented as a *user-defined circuit* in the FPGA. Questions now arise as to how user-defined circuits are addressed and manipulated and whether they can be composed within and across nodes. Also of interest is when and how user defined circuits are placed in the FPGA. The remainder of this section addresses the current design to support such operations.

A. Active Message Style Programming

The programming model of this architecture is similar to that of active messages [12]. Each message injected into the network has both a destination identifier and parameters specifying how the message should be processed by the destination. Each user-defined computational circuit loaded in the FPGA has a unique identifier that is globally known to all endpoints in the cluster. An endpoint can therefore perform a desired computation in either an FPGA or endpoint by injecting a properly formatted message into the network. Should an FPGA receive a message requiring a user-defined circuit that is not present in the FPGA, an exception is raised and the host CPU intervenes to reconfigure the FPGA.

B. Pipelined Model of Streaming Computations

The model can be naturally extended to include coordinated computations across multiple cluster nodes. FPGA and NI resources in a cluster node are utilized as pipeline stages in an overall computation with the SAN serving as a means of delivering data between stages. Figure 3 illustrates such operation. In this example a host application injects a message into its own FPGA-based NI. The message is received by the FPGA, processed, and forwarded to an FPGA-based NI in a neighboring node. This is repeated until the pipeline is complete and the message is ejected at its intended destination.

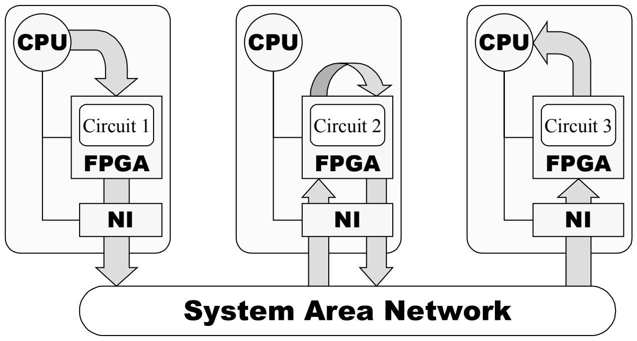

Figure 3: Pipelined Streaming Computations

The streaming model of in-network computation requires mechanisms for forwarding data between pipeline stages. This task can be accomplished by dedicating a region of memory in each FPGA to serve as a *forwarding directory*, and by identifying all streaming messages as belonging to specific pipelines. When a message for a streaming computation arrives at an FPGA, the message's pipeline identifier is used to retrieve routing information from the forwarding directory. This routing information specifies how the results of the message's computation at the FPGA are to be forwarded to the next cluster endpoint in the pipeline. Users may establish new computational pipelines by adjusting values in the forwarding directories of each cluster resource to be utilized in the pipeline. An active message handler is included in the FPGA to specifically facilitate such updates. A resource may house multiple stages of the same pipeline in this system, and multiple pipelines may co-exist in the same resource.

C. Dynamic Customization

With a fixed area FPGA there is a fixed capacity for housing user-defined circuits. Mechanisms must be provided to allow the FPGA to be dynamically reconfigured on receipt of messages. This is the hardware analog of an active message handler that is invoked when a message is received.

The FPGA is a shared resource that must be re-configured at different times. One means of solving this problem is through simple protection mechanisms. In this approach the host CPU manages the FPGA. An application must request that the FPGA be loaded with particular user-defined circuits before any processing can take place, as well as transmit release information when the computations are finished. While adding both startup latency and the potential for resource deadlock, this approach is simple to implement and adequately provides a means of dynamically reconfiguring the FPGA.

An alternative means of sharing access to cluster FPGAs is through demand driven management of FPGA resources. This approach follows the concepts of demand paging in virtual memory, where a resource is loaded as needed by applications. Should an FPGA receive a message that requires a user circuit that is not present, the FPGA flushes its runtime state information to on-card SRAM and generates a *function fault* for the host. The host determines which FPGA configuration best resolves the function fault and then reconfigures the FPGA. When the new FPGA configuration begins operation it restores its state information from SRAM and begins processing the message that caused the function fault. Due to the high overhead of reconfiguring the FPGA, it is expected that user applications will employ resource scheduling or locking mechanisms as previously discussed in order to minimize the number of dynamic function faults.

Dynamic fault management for the FPGA can be expanded through both traditional paging and FPGA-specific optimizations. An example of the latter is extensions to utilize partial reconfiguration, where a portion of the chip can be reconfigured with a user defined circuit as opposed to the entire FPGA. This would require a new generation of tools and is the subject of active research. Another solution is one wherein, the host could be adapted to extract messages that generate function faults and execute the function on the host rather than in the FPGA.

D. Advantages

An active SAN that allows processing to take place close to the network wire is beneficial for a number of reasons. First, the possibility that non-CPU cluster resources may be better suited to solving particular computational problems can result in improved performance simply through the use of accelerated hardware. Second, as with most co-processors, moving computations away from host CPUs potentially reduces the workload of the host CPU, allowing more computationally intensive tasks to take place. Third, placing high-performance computational engines close to the network allows other devices in the cluster to make use of these resources. For example, data capture devices and intelligent storage devices are likely to be lacking the performance capabilities of modern workstation CPUs. Rather than relying on host intervention, these devices can utilize accelerators located within the network.

V. DESIGN

A prototype ASAN NI is constructed using commercial FPGA and NI cards. We have adapted our SAN communication library [15] to use this NI in implementing the programming model described in Section IV. While this implementation differs from the higher performance single-chip solutions described earlier, it suffices to develop the programming infrastructure and study the major elements of this programming model and implementation.

A. Prototype Hardware

For the FPGA portion of the prototype the ASAN NI uses the Celoxica RC-1000 [13]. This card features a Xilinx Virtex-1000 FPGA, 8 MB of on-card SRAM, and PCI Mezzanine Card (PMC) sockets for connecting two daughter cards. The SAN hardware for the prototype is Myricom's Myrinet [14]. This SAN features gigabit links, a switched network fabric, and a NI with on-card memory and processor. A LANai 4.3 version of the Myrinet NI card in the PMC form factor was available at Georgia Tech, allowing the NI card to be directly attached to the RC-1000 FPGA card. Figure 4 illustrates the overall architecture of our FPGA-enhanced NI card and the major hardware components of the individual cards.

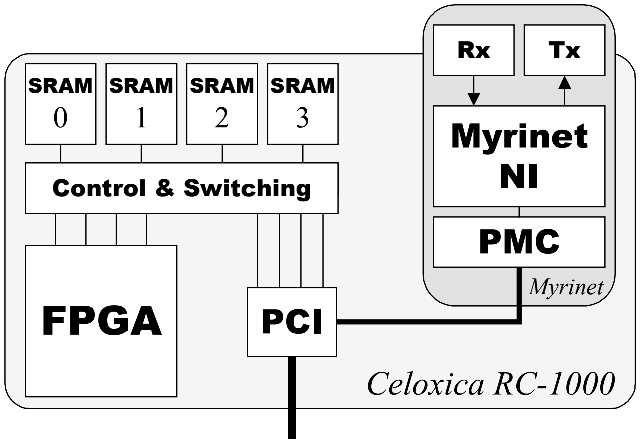

Figure 4: Celoxica RC-1000 and Myrinet Peripheral Devices

The Celoxica memory is divided into four 2 MB banks of SRAM that can be simultaneously accessed by the FPGA. Each memory bank is single ported and provides 32-bit data. A CPLD on the RC-1000 card implements arbitration between the FPGA and the CPU host for shared access to the single ported SRAM banks. Exclusive ownership of an SRAM bank is granted to either the FPGA or the CPU based on the earliest received request. PCI and PMC transactions are managed for this card through a commodity PCI chip. While this chip can perform PCI DMA transactions, DMA operations cannot be initiated directly by the FPGA.

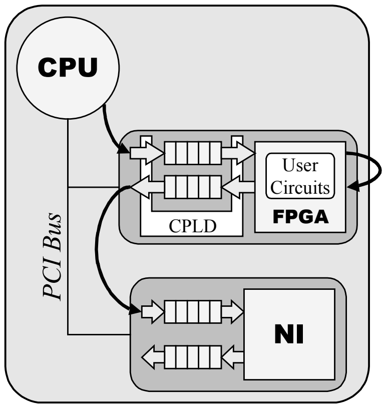

Figure 5: Host with FPGA and NI Cards

The operational model of this prototype is illustrated in Figure 5. Data flow in this active SAN is divided into two contexts: intra-nodal and inter-nodal communications. First, the SAN communication library provides communication between endpoints located in the same node. Since devices in this context are local, this style of communication can be accomplished using queues and PCI transactions. Each endpoint maintains a set of incoming and outgoing queues for all other endpoints in the local host, allowing transfers between two local endpoints to proceed without assistance from the local CPU. The second form of communication is between host computers. This task is managed entirely by the network interfaces of the SAN through a reliable and ordered link delivery protocol [15].

B. SAN Communication Library

The SAN communication library is the General-purpose Reliable In-order Message layer (GRIM) [15]. GRIM is an experimental SAN messaging layer utilized at Georgia Tech that is tailored to the sharing of SAN NIs between host CPUs and a range of intelligent peripheral devices. A key feature of the software is that all device independent communication functions reside in the NI. This design simplifies the amount of work any endpoint must perform to use the network. Peripheral devices interact directly with the NI, moving data to and from inbound and outbound message queues with minimal host intervention. For example, GRIM has been used to integrate the Brooktree BT848 video capture card [16] and Cyclone System's I_2O server adapter card (featuring dual Ethernet and ultra-wide SCSI ports) [17] into a SAN environment. Messages are transmitted reliably and in-order between NIs through a NI-based flow control protocol. Host-to-Host latencies are approximately 13 μs with bandwidths in excess of 100 MB/s. The current version of GRIM operates with the LANai 4 or LANai 9 versions of the Myrinet NI, and can be adapted to work with other network cards that feature a programmable processor and on-card memory.

Other high performance message layers could be utilized in the active SAN environment. GM [18], VIA [19], and LFC [20] all employ NI-based flow control and contain primitives that permit multiple endpoints in a host to share the NI. While non-trivial it is certainly possible to extend host-CPU based message layers to operate with PCI devices, as demonstrated in the adaptation of GM for use with a SCSI controller [21]. In addition to the advantage of familiarity, we utilized GRIM for this work because i) its underlying mechanics were already adapted to work with PCI devices and ii) functionality for auto-configuring PCI-based endpoints was already available.

C. Architectural Organization

To effectively utilize the FPGA we must have simple effective abstractions of the hardware. The abstractions used in the ASAN project are illustrated in Figure 6. Computations are performed in the *User Area* – a portion of the chip reserved as a canvas for configuring user-defined circuits. This is currently approximately 75% of the Virtex-1000 chip and is designed to house up to eight user-defined circuits. The remaining space is the FPGA control block, which manages the data flow and interfaces between user-defined circuits and the on-card SRAM. This unit manages incoming and outgoing message queues for interactions with the communication library, and maintains an interface for *scratchpad memory*. Scratchpad memory is a region of the RC-1000's on-card memory that is allocated for use exclusively by user-defined circuits. The scratchpad allows up to

4 MB of data to be stored close to the FPGA for improved computational performance. The forwarding directory is stored in SRAM bank 0 and consists of 256 entries for pipeline stage forwarding.

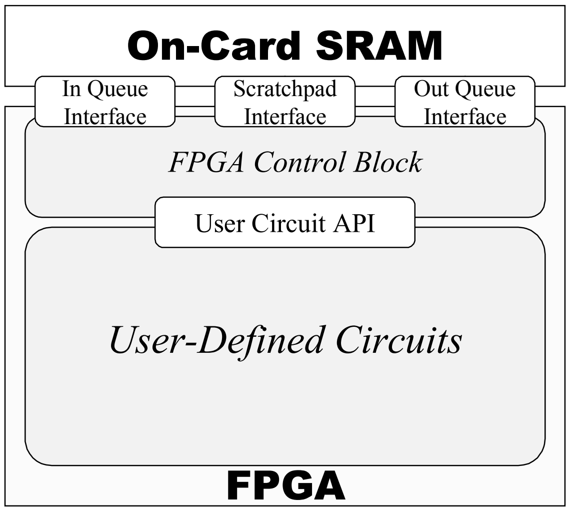

Figure 6: FPGA Organization

FPGA configurations are written in VHDL, simulated in Active-HDL, synthesized with Synplicity's SynplifyPro, and placed and routed with the Xilinx Foundation tools. While automated, the process of synthesizing, placing, and routing can take from thirty minutes to several hours depending on the complexity of the user-defined circuits. Multiple example cores were constructed and are described below along with estimates of the amount of resources required for each core as a percentage of a Virtex-1000 FPGA.

- **DES** [22] (6%): A wrapper for the Free-DES [23] core was constructed allowing both encryption and decryption of 32-bit blocks with a 56-bit key.
- **RC6** [24] (13%): This RC6 encryption/ decryption engine operates with up to 1024 rounds (R), at 32-bit width (W), with key lengths (B) up to 1024 bytes.
- **MD5** [25] (26%): A message digest function that generates a 128-bit identifier for a data stream.
- **ALU Operations** (5%): A set of 32-bit integer operators including add, subtract, multiply, min, max, and logical operators.

These test functions illustrate that the Virtex-1000 contains enough programmable logic to implement complex algorithmic functionality. Newly available Virtex II chips are approaching an order of magnitude increase in area over the Virtex-1000 while the research community has begun to consider designs on the order of tens of millions of gates.

D. Application Programming Interface

User cores must be independently constructed in a manner that can be used by applications. Similarly applications must know how to reference and use any of a set of user-defined circuits. This functionality is achieved via the definition of a standardized interface. Functionality common across user-defined circuits should be moved into this interface, for example, the movement of data to/from memory. This interface must also allow multiple computational circuits to co-exist and provide mechanisms by which users can encapsulate input data for the circuits in a message.

We have defined and implemented an application programming interface (API) that meets these requirements. The API design engenders three specific features in user-defined circuits. First, each core must be uniquely identified. This identifier is equivalent to the active message handler identifier of the GRIM library and is utilized by the interface to determine which computational circuit in the FPGA processes a message. This identifier can be supplemented with a sub-operator identifier that allows a single circuit to perform multiple related functions. Second, a computational circuit may asynchronously read from up to two vector data sets and write a single vector output. Both the addresses and lengths of each vector are specified in the message header and are linearly read or written by the interface as required by the computational circuit. Finally, each message processed by the FPGA contains a forwarding identifier. This identifier is utilized to determine what actions should be taken with the generated results. Results may simply be stored in scratchpad memory, "recycled" in the same

FPGA for further processing, or forwarded to another endpoint in the cluster.

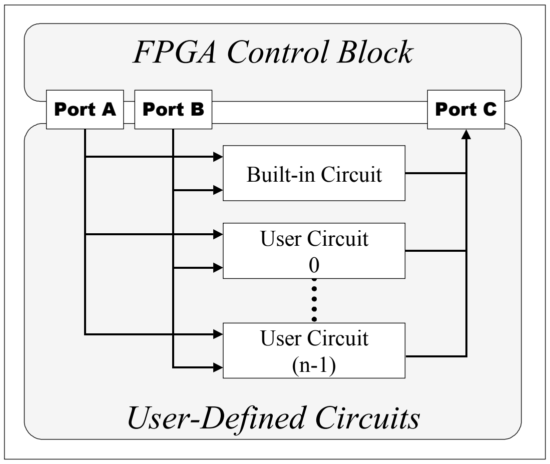

Figure 7: Interface for User-defined Computational Circuits

Figure 7 is a simplified representation of the interface between user-defined circuits and the FPGA control block. Input vectors for the circuits are supplied from vector ports A and B, and results are stored through port C. In addition to multiple user-defined circuits all FPGA configurations contain a built-in computational circuit. This unit provides a series of linear 32-bit ALU operations for vector inputs such as add, multiply, AND, OR, XOR, min, and max. Additionally this unit contains a no-operation function that allows an input vector to be streamed from port A or Port B to port C without modification, allowing memory copies in the scratchpad.

E. FPGA-NI Synchronization

In the prototype the FPGA and NI reside on separate PCI cards. Therefore it is necessary to provide synchronization and transport between these cards. We note that this effort is considerably more complex than would be the case for a NI card constructed entirely using an FPGA such as the Excalibur that provides reconfigurable hardware and a NI in a single chip. Two card-specific characteristics of the Celoxica RC-1000 must be addressed in supporting the FPGA card using the GRIM communication library. First the on-card memory is a shared resource that requires arbitration. This differs from other peripheral devices where on-card memory can be accessed at any time, with contention resolved in a dedicated memory controller. Second, the RC-1000 does not have a general-purpose processor for which existing communication software can be ported. Instead the communication software must be constructed as a portion of the hardware loaded into the FPGA.

Addressing the first challenge of sharing access to RC-1000 memory is resolved through the use of both an allocation scheme and arbitration mechanisms. As a first step in reducing contention, we placed incoming and outgoing message queues for the FPGA in SRAM banks 0 and 3 respectively, and allocate SRAM banks 1 and 2 for exclusive use by the FPGA as a scratchpad. Additionally, access to the banks housing the message queues is requested only when messages are injected or ejected. Other entities in the host (i.e., the host CPU and the Myrinet NI) acquire access to FPGA SRAM banks by writing a request to and polling from the FPGA card’s PCI registers. The difficulty in arbitrating for access to RC-1000 memory is that the card contains only a single PCI register for filing memory requests. This creates a race conditions when multiple communication endpoints concurrently attempt arbitration that could result in access to a memory bank being removed falsely. We resolve this issue by creating an entity that is responsible for combining all host and PCI device requests for FPGA memory. We assigned this task to the Myrinet NI, since the NI is near the FPGA card on the PCI bus and since the NI already runs in a tight loop that can be modified to poll for new memory bank requests.

The second challenge for integrating the RC-1000 FPGA card into the GRIM library is adapting the communication library software to function with FPGA hardware. In our approach the FPGA control block is designed to manage the incoming and outgoing memory queues. This hardware periodically obtains access to the memory banks and examines the queue pointers to observe changes in queue capacity. Additional hardware is necessary to parse incoming message headers, format outgoing messages, and to route data between user circuits and the scratchpad memory banks.

VI. Performance Evaluation

The FPGA-enhanced NI described in the previous section was utilized in a cluster based on 550 MHz Pentium III processors loaded with the Linux 2.4 operating system.

A. FPGA Interactions

Measurements were performed on the prototype to uncover the low-level costs of interactions with the RC-1000 FPGA card. First we examined the amount of time required for a host process to acquire and release individual memory banks on the RC-1000 without the explicit synchronization discussed in the previous section. Using programmed I/O the host required 13 μs to acquire and 2 μs to release memory banks. Next we measured the amount of time required for the host to perform the same operations, but this time using the Myrinet NI for centralized arbitration. This increased the time to 20 μs to acquire and 8 μs to release memory banks. We then measured arbitration from the NI perspective. We observed that the NI could acquire memory in 8 μs and release in 5 μs largely due to its proximity to the FPGA card. These communication penalties are incurred for each message transferred with the FPGA card.

As a means of examining obtainable bandwidth we performed experiments involving the transfer of blocks of memory with the FPGA card using the PCI bus. First we examined movements between the host and the RC-1000. The maximum observed bandwidth for the card is 85 MB/s. When moving a message-sized (4 KB) block of data, writes to the card occurred at 38 MB/s and reads from the card occurred at 59 MB/s. Next, we measured the bandwidth between the Myrinet and RC-1000 cards. For a 4 KB block of data, the Myrinet card's reads and writes both took place at 126 MB/s. The maximum transfer rate between the Myrinet and Celoxica cards was observed at 130 MB/s.

B. FPGA Operations

A fundamental assumption in this work is that FPGAs can rapidly process in-transit messages. Therefore we examined the latency of the prototype system. In Table 1 we present the amount of time required to move an example 4 KB message from either a host application or NI to a local FPGA and then back to the host application. FPGA timings are presented in both number of clocks required for operation as well as in seconds based on a conservative 20 MHz FPGA clock. These numbers assume that the computational unit loaded in the FPGA has a single cycle of latency. While we expect computational units will require multiple execution cycles, pipelined units pay this cost only at startup as the pipeline stages are filled.

Active Unit	Operation	FPGA Clocks	Time (μs)
(Host or NI)	Acquire RAM 0	-	(20, 8)
	Inject Message		(107, 32)
	Release RAM 0		(8, 5)
FPGA	Acquire RAM 0,3	8	0.40
	Fetch Header	7	0.35
	Fetch Forwarding	5	0.25
	Fetch Payload/Data	1024	51.2
	Computation/Store Latency	1	0.05
	Store Header	48	2.4
	Update Queue Pointers	3	0.15
	Release RAM 0,3	1	0.05
Host	Acquire RAM 3	-	20
	Perform DMA		69
	Release RAM 3		8

Table 1: Operational Timing

Several observations can be made from this table. First, even at a low clock rate the FPGA is able to rapidly process an entire message. The FPGA's cycle counts can be utilized to estimate performance of a production-level system. Second, while the fetch, compute, and store operations of the FPGA can be efficiently overlapped in a pipelined fashion, the memory arbitration scheme for the card prevents the overlapping of message injection, computation, and ejection. Finally, the measurements acquired in the previous section can be applied to determine the latency of a remote operation. Given the high cost of arbitration, the low latency of SAN delivery (12 μs), and the fact that the NI can acquire FPGA memory much faster than the host, remote and local interactions with the FPGA card are roughly comparable.

C. *Reconfiguration*

The final area of performance examination is in reconfiguration. We measured the time required for the host to reconfigure the FPGA with a configuration available in host memory to be approximately 96 ms. Reconfiguration is a lengthy process due to the slow reconfiguration speed of the Virtex architecture and the large size of a configuration file. Other tasks required for dynamic reconfiguration in our prototype such as host notification and configuration startup costs are negligible.

VII. CONCLUDING REMARKS

Technology is constantly increasing the power, cost effectiveness, and flexibility of the available silicon. In this paper we have explored a model of computation where the SAN infrastructure is extended to perform data intensive computations during communication. By making the network a compute entity network attached media and storage devices can access this capability altering the flow of data and computation in a typical SAN cluster. The overall goal is the availability of a scalable communications and computation infrastructure for cluster-based data intensive computing.

VIII. ACKNOWLEDGEMENTS

We gratefully acknowledge the support for this research by a grant from the Advanced Networking Infrastructure and Research Division of National Science Foundation under grant ANI-9876573 and a grant from the Xilinx Corporation.

References

[1] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, "A Survey of Active Network Reasearch," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80-86, 1997.

[2] I. Hadzic, J. Smith, and W. Marcus, "On-the-fly Programmable Hardware for Networks," in *Proceedings of GLOBE-COM'*98, 1998.

[3] D. Taylor, J. Turner, and J. Lockwood, "Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers," in *IEEE Proceedings of Open Architectures and Network Programming*, pp. 25-34, 2001.

[4] D. Lee, S. Harper, P. Athanas, and S. Midkiff, "A Stream-based Reconfigurable Router Prototype", in *IEEE International Conference on Communications*, 1999.

[5] T. Harbaum, D. Meier, M. Prinke, and M. Zitterbart, "Design of a Flexible Coprocessor Unit," in *Proceedings of the 25th EUROMICRO Conference*, pp. 335-342, 1999.

[6] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott, "Implementing an API for Distributed Adaptive Computing Systems," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

[7] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," in *International Journal of Supercomputing Applications*, vol. 8, 1994.

[8] Xilinx Corporation, Virtex II 1.5V Field-Programmable Gate Arrays datasheet, 2001.

[9] Xilinx Corporation, "Xillinx and Conexant Announce Licensing Agreement of Skyrail 3.125 GBps Serial Transceiver Technology", Corporate Press Release, 2000.

[10] InfiniBand Trade Association, "InfiniBand Specification 1.0a," 2001.

[11] Altera Corporation, "ARM-Based Embedded Processor Device Overview," product data sheet, 2001.

[12] T. Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *Proceedings of The 19th Annual International Symposium on Computer Architecture*, pp. 256-266, 1992.

[13] Embedded Solutions Ltd., "RC1000-PP Hardware Reference Manual," product data sheet, 1999.

[14] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. "Myrinet: A Gigabit-per-second Local Area Network," in *IEEE Micro*, vol. 15, no.1, 1995.

[15] C. Ulmer and S. Yalamanchili,"A Messaging Layer for Heterogeneous Endpoints in Resource Rich Clusters," in *Proceedings of the First Myrinet User Group Conference*, 2000.

[16] Conexant Systems, Inc., "Bt848/848A/849A Single-Chip Video Capture for PCI," data sheet, 1997.

[17] Cyclone Microsystems Website: http://www.cyclone.com

[18] Myricom, Inc., "The GM message layer," http://www.myri.com

[19] Virtual Interface Architecture Organization, "Virtual Interface Architecture Specification Version 1.0," 1997.

[20] R. Rhoedjang, T. Ruhl, and H. Bal, "LFC: A Communication Substrate for Myrinet," in *Fourth Annual Conference of the Advanced School for Computing and Imaging*, 1998.

[21] P. Geoffray, "OPIOM: off-processor IO with Myrinet," in *Proceedings of First International Symposium of Cluster Computing and the Grid*, pp.261-268, 2001.

[22] National Bureau of Standards, "Data Encryption Standard," Federal Information Processing Standards Pub. 46, 1977.

[23] David Kessner and the Free IP Project, http://www.free-ip.com/DES/index.html

[24] R. Rivest and M. Robshaw and R. Sidney and Y. Yin, "The RC6 Block Cipher," 1998.

[25] R. Rivest. The md5 message digest algorithm, RFC 1321, 1992.