CHAPTER VI

PERIPHERAL DEVICE EXTENSIONS

A key characteristic of message layers for resource-rich cluster computers is extensibility. From a hardware perspective, it must be easy for users to adapt these message layers to support new and diverse peripheral devices in the cluster. In this effort, peripheral devices are visualized as communication endpoints and added to the cluster's global pool of distributed resources. Therefore, the message layer serves as a general framework for interconnecting both host CPU and peripheral device resources. GRIM is unique in that it is a message layer that is specifically designed to provide this framework. Peripheral devices in GRIM interact directly with other resources in the local host (e.g., the NI or other local endpoints) using efficient PCI transactions. Peripheral devices with sufficient processing capabilities are allowed to operate in an autonomous manner without the guidance of the host CPU. For legacy peripherals that are less capable, GRIM can be configured to utilize host-level software to manage the device's interactions with the message layer.

This chapter focuses on the task of integrating peripheral devices into the cluster environment as communication endpoints. The discussion begins with a generalized description of how new peripheral devices are added to the GRIM environment. In order to construct new peripheral device endpoints, designers must be aware of the manner in which endpoint software is expected to function as well as the methods by which GRIM manages peripheral device resources. As a means of illustrating the integration process, the remaining portion of this chapter provides implementation details for four peripheral devices that have been incorporated into GRIM. These devices include an intelligent server adaptor card, an FPGA accelerator card, a video capture card, and a video display card. These devices offer a diverse range of processing capabilities and help to demonstrate how GRIM can be used in a flexible manner to allow applications to utilize these resources.

6.1 Adapting a Peripheral Device for use with GRIM

GRIM is designed to allow multiple peripheral devices distributed throughout a cluster to be utilized as resources in the virtual parallel-processing machine. The approach taken in GRIM is to allow each peripheral device to function as a communication endpoint that interacts directly with the communication library. Therefore, the process of adapting a peripheral device to operate in the GRIM environment begins by constructing endpoint software for the peripheral device. This software is comprised of a set of device-specific active message function handlers for the device and message-passing mechanisms that allow the device to interact with other resources in the local host. Once endpoint software is available, a designer must construct host-level software to allow the device to be utilized in the GRIM environment. GRIM provides a series of built-in functions that can be used by designers to simplify this task. Finally, end users interact with peripheral device endpoints through a series of resource-management operations provided in GRIM. These functions allow a user to locate, reference, and communicate with a resource that is available in the cluster. The overall organization of the software for a peripheral device endpoint is pictured in Figure 6.1.

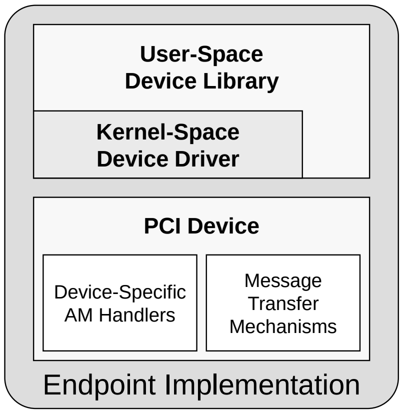


Figure 6.1: The major components of a peripheral device endpoint implementation.

6.1.1 Peripheral Device Endpoint Software

The first task in integrating a peripheral device into the GRIM environment is to construct low-level software that allows the device to function as a communication endpoint. This software is divided into two parts: device-specific active message function handlers and message-passing mechanisms. The active message portion of the software is responsible for serving as a means by which end applications can invoke specific operations at a peripheral device. A designer must therefore construct active message function handlers for a new peripheral device that adequately capture the device's key capabilities. After a device's function handlers have been constructed, a designer can add information about the handlers to a static database that is available in the GRIM library. This database allows a peripheral device's function handlers to be visible to applications in the cluster and removes the need for a peripheral device to register its handlers at runtime.

Low-level endpoint software must also implement message-passing mechanisms for communicating with the NI and other endpoints in the local host. For outgoing messages, an endpoint utilizes PCI DMA operations to transfer data to the message queues of other endpoints. For incoming messages, an endpoint allocates a block of memory for housing message queues that other endpoints can write. An endpoint must periodically poll its incoming message queues to determine if new messages are available. GRIM provides a set of files in C that can be used to simplify the task of implementing this functionality. These files include data structures for messages and message queues, as well as skeleton code for performing key message-passing operations. A designer can use this software by including the files in an endpoint implementation and then defining device-specific functions needed for the message-passing operations, such as a function for initiating a PCI data transfer.

6.1.2 Host-Level Integration

After endpoint software is constructed for a peripheral device, a designer must provide host-level software that allows the card to be utilized by the operating system and the GRIM library. The operating system portion of this software is implemented in a kernel-level device driver. While device drivers are nontrivial to implement, GRIM performs most of its operations in user space. Therefore, device drivers for GRIM can be relatively simple, and require only basic operations such as initializing the peripheral device and providing a memory map of card memory to user-space applications. Once a device driver is available, a designer must construct appropriate user-space software that allows the GRIM library to interact with the card. Typically, this software includes initialization functions and device-specific operations that may be needed by applications.

6.1.3 Library Initialization

The GRIM library must perform a number of initialization functions before the cluster can begin processing an application. The first step in the initialization process is for GRIM to load configuration information for the cluster from a set of user-defined configuration files. In addition to providing basic information such as routing tables, the configuration files specify the peripheral device resources that are available in the cluster. Each host in the cluster uses this information to determine which peripheral devices it is equipped with and how it should initialize its devices. Cluster configuration information is stored in a database that can be accessed by applications at runtime to help locate resource information.

After all devices in the local host have been initialized, GRIM must configure each peripheral device with information that allows the device to interact with the local NI and all of the other endpoints in the host. In this process, GRIM determines how many incoming message queues each endpoint needs and how large each queue should be. GRIM then updates all of the queue pointers for all of the endpoints in the local host. Two sets of pointers must be configured for each message queue, one for the sending endpoint and the second for the receiving endpoint. When configuring these pointers, GRIM must translate all references to the message queue into values an endpoint can utilize. This procedure is complex, and automatically accounts for virtual-to-physical address translations, byte-order differences, and card-specific memory addressing issues. Needless to say, the automatic endpoint configuration functions are the most mind-numbingly complex part of GRIM. However, these operations are designed in such a way that when a new device is added to GRIM, users simply supply basic information to GRIM's configuration function and configuration takes place automatically.

6.1.4 Runtime Management

From an end user's perspective, a communication library for a resource-rich cluster computer must provide basic mechanisms for allowing users to customize their interactions with peripheral device endpoints. After initialization, the GRIM library provides a set of functions for performing such operations. With these functions, a user can query the communication library to locate a specific type of peripheral device. Users can perform these queries in the context of the cluster's global resources, or limit searches to particular hosts. When the communication library successfully locates a desired resource, it returns an integer identifier that can be used to reference the resource. Applications can then invoke operations at the resource simply by injecting active messages that are marked with the reference. Internally, GRIM provides all of the routing that is necessary for the messages to be delivered to the resource.

It is expected that some peripheral device endpoints will require more complex management functions than the current runtime system provides. For example, if an endpoint needs to use a peripheral device to perform a series of computations, it is beneficial if the endpoint can temporarily obtain exclusive ownership of the device so that the computations can take place without interruption. This type of operation can be implemented in GRIM by utilizing the peripheral device's host CPU to manage ownership of the resource. In this

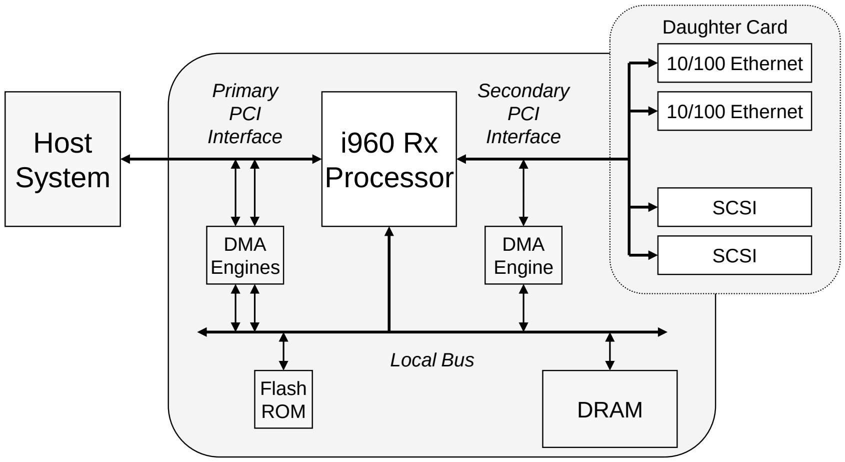


Figure 6.2: Architecture of the Cyclone Microsystems I_2O development card.

approach, the host manages a reservation system for a peripheral device that is manipulated with active messages. When an endpoint needs to obtain ownership of the resource, it transmits the appropriate request to the host and waits for a response before accessing the device. Similar approaches can be used to layer additional functionality on top of the existing GRIM software.

6.2 Cyclone Microsystems Server Adaptor Card

The first peripheral device added to the GRIM communication library was the Cyclone Microsystems server adaptor card [4]. While originally marketed as a general platform for evaluating the Intelligent I/O (I_2O) [51], extensions, this card has become a valuable tool for active disk and active network research efforts [36]. The overall architecture of the card is presented in Figure 6.2. At the core of this architecture is an Intel i960 processor [46] that operates at 66 MHz. This processor includes a built-in 32b/33 MHz PCI unit that features chained DMA engines and PCI doorbell registers. The card is equipped with 4 MB of on-card DRAM that can be expanded to 36 MB by populating a standard SIMM socket. In order to market the development card for different uses, Cyclone Microsystems placed a custom expansion interface on the card for attaching a daughter card. ATM, Ethernet, and SCSI daughter cards were constructed for the development card. The daughter card used in this research effort features two Fast Ethernet ports and two Ultra-wide, Ultra-fast SCSI ports. Communication between the i960 and the daughter-card components physically takes place using a secondary PCI bus.

The Cyclone Microsystems development card uses a custom version of the VxWorks operating system [96] to control the card's hardware resources. VxWorks provides a UNIX-like, multitasking environment for applications. Application developers write programs for this environment in the proprietary Tornado development system [95], and then load the compiled binaries onto the i960 using either a serial download cable or the card's Ethernet controllers. While the VxWorks operating system greatly simplifies the development process with this card, it is important to note that the i960 may be underpowered for operations required by the operating system. The I_2O card was used in multiple research efforts at Georgia Tech, the most notable of which is the QUIC [94] project.

6.2.1 Endpoint Construction

The process of porting the GRIM endpoint software to function on the I_2O card was relatively straightforward due to the card's rich programming environment. The endpoint software was constructed to run as a normal VxWorks process on the I_2O card. At initialization time, the endpoint process allocates a block of memory for housing incoming message queues and shares this information with the host library. After initialization the process monitors the incoming message queues, processes messages, and ejects outgoing messages to the host's NI or endpoints. Message ejections are performed using the card's DMA engines, allowing the I_2O card to operate autonomously without the guidance of the host CPU. With the help of Ivan Ganev, Robert Goldman, and Kelly Norton, a Linux device driver was constructed for the I_2O card that allowed the card to be utilized by host applications.

Multiple active message function handlers were constructed for the I_2O card to provide end users with a means of controlling the card's hardware. Initial development with this card focused on constructing handlers for both network operations and storage operations. However, soon after this development began Cyclone Microsystems announced that the disk controller hardware for this card did not operate in a reliable manner. Therefore the focus of this work shifted to utilizing this card exclusively for its network hardware. Active message handlers were constructed so that the I_2O card could be utilized as a network bridge, relaying messages between the SAN and the LAN when needed. These types of operations are necessary when a cluster is utilized as a large-scale network server that external hosts connect to through a LAN.

Active message handlers were constructed for the I_2O card to allow messages to be transferred between internal cluster resources and external Ethernet-based hosts. In this system an external host connects to the I_2O card using a long-term socket. Once a connection is established, transactions between the internal resources and external hosts can take place using a special active message that performs bridging operations. This message allows a normal active message to be encapsulated as the payload of the bridging message. When the I_2O card receives a bridge message, it extracts the encapsulated active message from the payload and forwards it to the appropriate resource in the cluster. In order for bridging to take place the I_2O card must maintain a table of Ethernet TCP connections so that it can relay data to the proper connection.

6.2.2 Performance Measurements

The I_2O endpoint implementation was constructed for an early version of GRIM. While this card has several shortcomings, including several documented hardware problems, it was sufficient for an initial proof-of-concept demonstration. Simple performance measurements were made of the endpoint implementation and are presented here. The early version of GRIM used in these measurements was less robust than the current version and thus incurred less overhead in host-to-host transmissions. For example, the early version of GRIM featured 13 μs latencies as opposed to 16 μs latencies for host-to-host transmissions. This difference should be taken into account when evaluating the performance of the I_2O card.

A test program was constructed to determine how efficiently a host-level application could interact with an I_2O endpoint. In this test, the time required to have a message sent to and returned from the I_2O card was measured. The resulting communication path is for host-NI-NI- I_2O, and was performed using the LANai 4 version of the NI card. The one-way travel time for a short message was measured to be approximately 21 μs. This latency is much larger than that required for two hosts to communicate. While the I_2O card is situated in close proximity to the NI card, the i960 processor is much slower than the host processor. Additionally, the I_2O card's DMA engines are designed to transfer large blocks of data and therefore have a large overhead for performing small PCI transfers. However, the I_2O card illustrates that a peripheral device can operate in an autonomous manner in the communication library and serves as an example of how intelligent peripheral devices can directly interact with the NI.

6.3 FPGA Accelerator

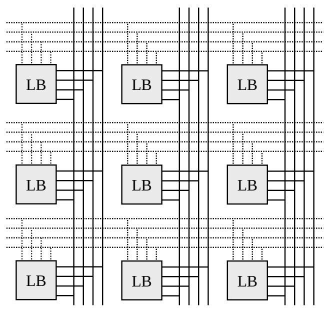
Field-programmable gate arrays (FPGAs) are reconfigurable hardware devices that can be programmed to function as application-specific circuits. In recent years, commercial FPGAs have grown significantly in capacity, and are now capable of emulating large blocks of custom computational circuitry. These circuits can be utilized as a means of accelerating application performance in a cluster, and therefore it is beneficial to consider methods by which FPGAs can be incorporated into a cluster. Currently, there are multiple FPGA cards that are commercially available for accelerating a host's computational performance. One of these cards is the Celoxica RC-1000 [34], which features a modern FPGA and large amounts of on-card SRAM.

In order to better investigate the use of FPGAs as computational resources in the cluster environment, the RC-1000 FPGA card has been adapted to function as a communication endpoint in GRIM. This process was non-trivial, as endpoint software had to be converted into hardware circuitry. This circuitry is referred to as the static frame for the device, and is responsible for managing interactions between the FPGA and other resources in the local host. A second block of circuitry referred to as the circuit canvas is used in the FPGA as a place for housing multiple user-defined computational circuits. These circuits are the hardware equivalent of the active message handlers found in other GRIM endpoints. This section provides basic details of the RC-1000 FPGA endpoint implementation. Additional details are provided in the following chapter as well as in Appendix B.

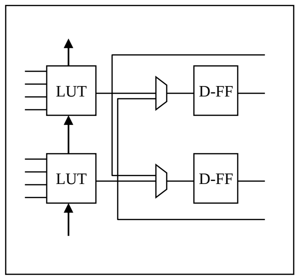
6.3.1 FPGA Overview

In order to use FPGAs in the cluster environment, it is first necessary to understand the basic characteristics of the technology. FPGAs are reconfigurable hardware devices that can be programmed to emulate custom, application-specific circuits. Unlike application-specific integrated circuits (ASICs), which cannot be reprogrammed, FPGAs can be reconfigured at runtime to emulate different circuits that are needed by applications. A high-level architecture of an FPGA is presented in Figure 6.3(a-b). In this architecture, an FPGA is comprised of (a) a two-dimensional grid of (b) programmable logic blocks. Each logic block contains a lookup table that emulates a desired logic function. Logic blocks are connected to implement more complex operations through a programmable interconnection network inside the FPGA. In addition to lookup tables, modern FPGA architectures include complex structures such as blocks of memory, high-speed multiplier arrays, general-purpose CPU cores, and high-speed network transceivers [97]. State-of-the-art FPGAs are advertised as being capable of emulating up to 8 million logic gates at a time [99].

Designs for FPGA devices are generally constructed in hardware-description languages



(a) FPGA Architecture



(b) Logic Block

Figure 6.3: FPGAs are generally (a) large arrays of programmable logic blocks (LBs) that feature (b) lookup tables (LUTs) and D-flip flops.

such as VHDL or Verilog. While these languages can simplify the design process, it is important to note that designing hardware is still significantly more time-consuming than designing software for a general-purpose CPU due to the low-level nature of the work. Once a design is debugged with simulation tools, it is synthesized into a gate-level description that is targeted for a particular family of FPGA devices. This description is then placed and routed for a target FPGA architecture using tools provided by the FPGA vendor. The end result of this process is a configuration file that can be loaded into the FPGA. Depending on the size and complexity of a design, it may take anywhere from tens of minutes to tens of hours for the entire compilation process to complete. Programming an FPGA with a configuration file can take several milliseconds in modern FPGAs.

6.3.2 Celoxica RC-1000 FPGA Card

The FPGA card chosen for integration into the GRIM communication environment is the Celoxica RC-1000 FPGA card. This card features a Xilinx Virtex-1000 FPGA [98], 8 MB of on-card SRAM, and PCI Mezzanine Card (PMC) [5] sockets for connecting two PCI daughter cards. A LANai 4.3 version of the Myrinet NI card was available at Georgia Tech in a PMC form factor, and allowed the NI card to be attached directly to the RC-1000 FPGA card. Figure 6.4 illustrates the overall architecture of our FPGA-enhanced NI card and the major hardware components of the individual cards.

The architecture of the Celoxica RC-1000 card divides on-card memory into four 2 MB banks of SRAM. Each bank is single ported and operates with 32-bit data values. The RC-1000 provides switching hardware and memory arbitration mechanisms to allow the single-ported SRAM banks to be accessed by either the FPGA or the PCI controller. The arbitration mechanisms are implemented in a CPLD through the use of two 4-bit request registers (one for the FPGA the other for the PCI unit), and one 4-bit grant register. In this scheme, exclusive ownership of a bank of SRAM is acquired by updating a request register and polling the grant register. Memory arbitration does not take place automatically for the PCI controller. Instead, the external entity initiating a transfer of data to the RC-1000's memory must first interact with the card's CPLD and obtain ownership of the involved SRAM banks. The PCI controller for the RC-1000 is the PLX-9080 chip [74], which provides a chained DMA engine for PCI transfers. Due to the manner in which the memory arbitration mechanisms are implemented for this card, the FPGA cannot initiate

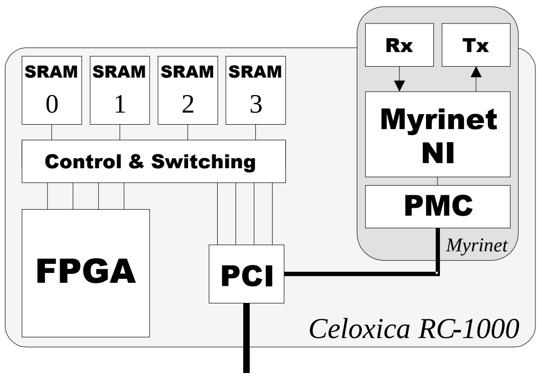


Figure 6.4: Celoxica RC-1000 and Myrinet Peripheral Devices.

its own PCI DMA operations. This implies that the card requires the host's assistance whenever the FPGA exchanges data with the host.

6.3.3 FPGA Endpoint Implementation

Integrating the RC-1000 FPGA into the GRIM environment involved defining a hardware configuration for the FPGA that allows the FPGA to function as a communication endpoint. This hardware configuration must satisfy three design goals. First, it must be capable of managing the card's incoming and outgoing message queues for interactions with the communication library. In addition to injecting and ejecting messages, the FPGA configuration must be capable of parsing an incoming message to determine which hardware circuitry should be used to process the message. Second, the configuration must provide simple mechanisms that allow computational circuits to access the RC-1000's on-card SRAM that is not used for the message queues. This memory can be used to store application data sets in order to improve computational performance. Third, the FPGA configuration must allow multiple user-defined computational circuits to be loaded in the FPGA for use by applications. These circuits are analogous to software-based active message function handlers found in other endpoints.

Based on the preceding requirements, circuitry was designed for the FPGA to allow the RC-1000 to function as a communication endpoint. As presented in Figure 6.5, the design divides the FPGA into two regions, and uses three separate interfaces to allow interactions between hardware units. The majority of the FPGA is used for the *circuit canvas*, a region of the FPGA that houses multiple user-defined computational circuits. These circuits process incoming messages and must adhere to a dynamic circuit API. The other portion of the FPGA is allocated for use as a *frame* for the canvas. The frame is a small region of the FPFA that is utilized to control card-specific interactions between the circuit canvas and the FPGA card's resources. The frame includes state machines for managing interactions with the communication library, the circuit canvas, and user-accessible on-card memory referred to as the scratchpad. Because the frame is designed to insulate the canvas from card-specific features, it is possible to port this work to other FPGA cards by modifying the frame.

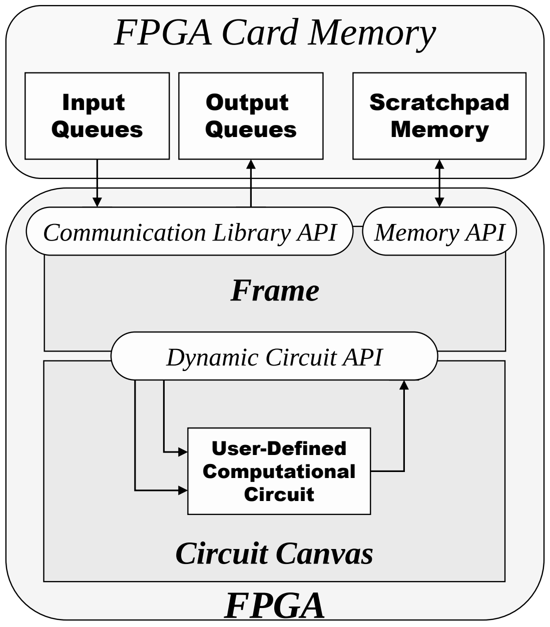


Figure 6.5: FPGA Organization.

6.3.4 User-Defined Circuits

One of the key features of the RC-1000 endpoint implementation is its capability for supporting multiple user-defined computational circuits in a single FPGA. The frame is designed with an interface that allows multiple computational circuits to be physically loaded in the canvas and dynamically utilized to process incoming messages. The frame employs two sets of signals to accomplish this task. First, each computational circuit is connected to the frame by a set of control signals. These signals allow the frame to activate a computational circuit and detect when the circuit has completed its work. The second set of signals routes vector data streams between the frame and the computational circuit. A vector data stream transfers a linear series of 32-bit data values using an asynchronous transfer protocol. Each computational circuit can use up to two input vector data streams and one output vector data stream.

A simplified block diagram of the interface between the frame and computational circuits housed in the canvas is presented in Figure 6.6. Each computational circuit is provided with vector data streams supplied by ports A and B in the frame. Results of the computations are streamed back into the frame through port C. In addition to supporting up to eight user-defined computational circuits, each FPGA configuration is also equipped with a built-in unit for basic ALU operations. This unit provides a set of linear, two-input vector operations (e.g., add, multiply, AND, OR, XOR, min, and max) as well as one-input vector operations (e.g., NOT and a no-operation). The no-operation is beneficial because it can be used to transfer memory from one memory location in the scratchpad to another.

6.3.5 Examples of User-Defined Circuits

Multiple user-defined computational circuits have been constructed for use in the circuit canvas. As a means of illustrating how a modern FPGA can house multiple complex hardware units, an FPGA configuration with a set of cryptography cores was implemented.

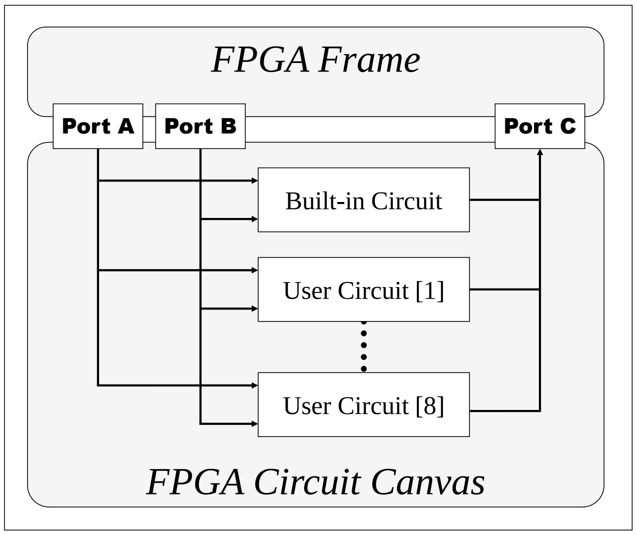


Figure 6.6: The interface for computational circuits in the canvas.

The post-layout design was examined to determine how much space each core occupies in the Virtex-1000 FPGA. These estimates are presented in terms of the percentage of the Virtex-1000's overall resources that are consumed by the circuits. These percentages can be translated to gate counts using the estimation that a Virtex-1000 can emulate approximately one million logic gates. Each core is briefly described as follows:

- **Digital Encryption Standard (DES)** [66] (6%): A publicly available DES core called free-DES [54] was ported to operate as a user-defined core. This unit can either encrypt or decrypt data supplied by vector data port A using a key supplied by vector data port B.

- **RC6** [79] (13%): Chris Wood implemented a version of the RC6 encryption standard for encrypting and decrypting data from vector data port A using a key schedule supplied by vector data port B. The engine operates with up to 1024 rounds (R), at 32-bit data value widths (W), with key lengths (B) up to 1024 bytes.

- **MD5** [78] (26%): The MD5 message-digest algorithm was implemented to generate a 128-bit identifier for data supplied by vector data port A.

- **Built-in ALU Operations** (5%): The frame features a built-in ALU core for basic vector operations.

The frame for this configuration requires approximately 20% of the Virtex-1000's resources. A significant portion of this allocation is for two blocks of the FPGA's internal SRAM so that the frame can buffer messages that are being processed. This design illustrates that multiple useful circuits can be loaded in the FPGA at the same time.

6.3.6 RC-1000 Interactions with GRIM

In order to allow the RC-1000 to function as a communication endpoint in GRIM, low-level message-passing mechanisms were constructed to facilitate data transfers between the RC-1000 and other resources in the local host. The first difficulty in this effort is that the FPGA

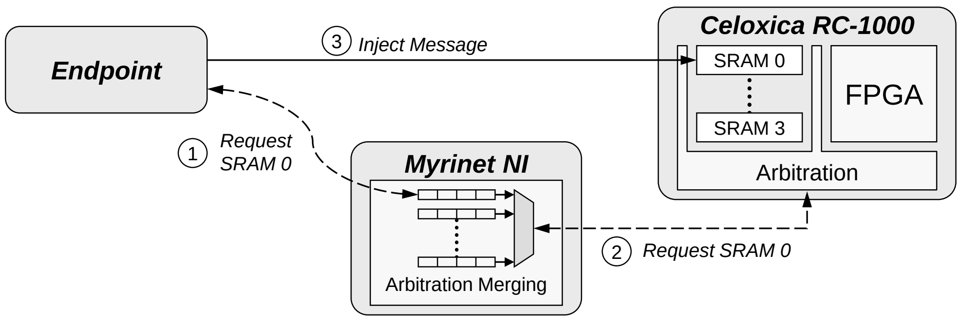


Figure 6.7: The use of the Myrinet NI to merge arbitration requests. An endpoint (1) passes a request to the NI to access RC-1000 memory. The NI merges the request (2) and interacts with the RC-1000 card. After access is granted the endpoint (3) injects the message.

cannot initiate DMA transfers. This problem was resolved through the construction of host-level software that initiates DMA transfers on behalf of the RC-1000 card. This software detects when the card needs to perform a transfer and issues the proper DMA operation. A more challenging issue in implementing the RC-1000's communication mechanisms is dealing with its card-specific memory-arbitration mechanisms. As discussed earlier, the RC-1000's PCI controller and FPGA share access to the card's single-ported SRAM banks through an arbitration scheme. Thus, when an endpoint needs to inject a message into one of the RC-1000's incoming message queues, it must first obtain exclusive ownership of the memory bank that houses the queue. Ownership is acquired by updating the card's request register and then polling a grant register to detect when the card has assigned ownership to the endpoint.

While the RC-1000's memory-arbitration mechanisms are adequate when only the host CPU uses the card, there is the possibility of a hazardous race condition when multiple resources in the host (e.g., the NI and the host CPU) access the arbitration mechanisms at the same time. The problem is that there is only one register for all off-card resources to place memory-arbitration requests. If an endpoint does not have knowledge of the current requests made by other endpoints in the host, it is possible for access to an SRAM bank to be released mistakenly. For example, consider the case where the host CPU and NI are each injecting a message into different queues located in the same SRAM bank of the RC-1000. If the NI finishes before the host CPU, it could mistakenly update the RC-1000's bank request register to indicate that access is no longer needed to the SRAM bank. Because the memory arbiter only observes the most recent update to the request register, it is possible for the arbiter to change ownership of the SRAM bank from the PCI interface to the FPGA. This series of events results in the host utilizing an SRAM bank for which it no longer has access.

A solution to the problem of allowing multiple resources in the host to coherently share access to the memory-arbitration mechanisms of the RC-1000 is to merge access requests at a single location in the host. In the GRIM implementation, this task is delegated to the Myrinet NI due to its proximity to the FPGA card on the PCI bus. As illustrated in Figure 6.7, a series of request queues are implemented in the NI. When the NI detects a new request in a queue, it updates a local set of registers for the queue and compares the sum of all the current requests to the last request issued to the FPGA card. If there is a difference,

Table 6.1: Performance measurements for RC-1000 memory arbitration.

Operation	**Resource Requesting Arbitration**	**Resource Performing Arbitration**	**Time (μs)**
Acquire SRAM	Host	Host	6
	Host	NI	13
	NI	NI	5.5
Release SRAM	Host	Host	2
	Host	NI	7
	NI	NI	3

a new request is sent to the RC-1000's arbitration unit through a PCI transaction. If this update is due to a new request for memory (as opposed to a release), the NI periodically polls the RC-1000's arbitration unit until the request is granted. The endpoints in the host system must poll the NI's arbitration registers to determine when access is granted for each request.

Tests were performed to characterize the RC-1000 memory arbitration mechanisms. In these tests both the host and the NI acquire and release ownership of a bank of RC-1000 SRAM. In the first set of experiments, arbitration is performed directly by the host or the NI. In the second set of experiments, the host utilizes the NI to perform arbitration on behalf of the host. The results of these experiments are presented in Table 6.1. As expected, utilizing the NI to perform arbitration for the host results in a significant performance penalty for the host. The host's indirect arbitration scheme incurs twice as much overhead as a direct approach. These measurements imply that it is beneficial for a host endpoint to invoke arbitration mechanisms infrequently, and that the host software should be designed to bundle multiple transactions with FPGA memory into a single operation whenever possible.

6.3.7 TPIL Performance for the RC-1000 Card

The TPIL software was adapted to operate with the RC-1000 card in order to improve host injection performance. The internal benchmarking features of TPIL that were used to measure the performance of the Myrinet cards in Chapter 5 were utilized to measure the performance of the RC-1000. The results are presented in Figure 6.8 for a P3-550 MHz host that is equipped with a 32b/33-MHz PCI bus. As expected, MMX and SSE PIO based transfers provide the best performance for injections smaller than approximately 3 KB. After this point, zero-copy DMAs become the most profitable transfer method, eventually reaching a performance of approximately 116 MB/s.

An interesting characteristic in these performance measurements is that PIO operations reach a peak value of 42 MB/s for transfers that are 1 KB in size. Performance gradually drops for transfers larger than this size until a steady-state value of approximately 32 MB/s is reached. The RC-1000's PCI chipset is likely to be the cause of this performance drop because the PCI chipset has a limited capacity for buffering incoming data from the PCI bus. PIO transfers can therefore be slowed if the host CPU attempts to inject data faster than the card can empty the buffer. DMA transfers do not experience this performance degradation because the DMA engine can throttle its transfers to match the capacity of the

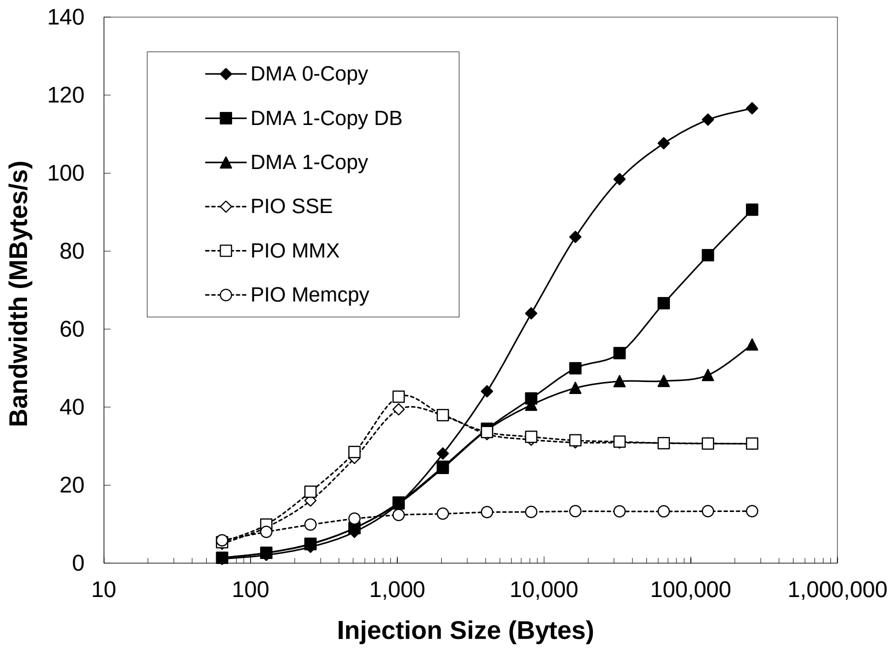


Figure 6.8: RC-1000 PCI injection performance for P3-550 MHz hosts.

buffer.

6.4 BrookTree Video Capture Card

The third peripheral device used as a resource in the GRIM environment is a video capture card based on the popular BrookTree BT8x8 chipset [21]. Many commercial video capture cards utilize this chipset because it is economical, and because device drivers exist for both Windows and Linux operating systems. The chipset integrates a PCI controller, video capture circuitry, and control logic in a single chip. BT8x8 card utilizes a block of host memory to serve as a frame buffer and employ a highly-configurable data-transfer engine to move incoming video data to the host. Unfortunately, this card does not feature a programmable CPU that could be used to implement GRIM endpoint software. Therefore, a host-level library was constructed to allow GRIM applications to access this device. This effort demonstrates that GRIM can be utilized to include even simple peripheral devices in the cluster communication model.

6.4.1 Overview of the BT8x8 Video Capture Card

Many commercial video capture cards are based on the BrookTree BT8x8 chipset. This chipset is popular because it implements all capture hardware in a single chip, can process NTSC, PAL, and SECAM analog video sources, and can perform operations such as scaling, clipping, and pixel-format transformations in hardware. In order to reduce the cost of the chipset, the BT8x8 chipset is designed with minimal buffer space for housing video data. Instead of buffering frames of video data in card memory, the BT8x8 uses a programmable DMA engine to stream captured data directly into a region of the host's memory. A desirable aspect of this approach is that a frame of video data can be streamed to any location in the host, including the on-card frame buffer of a video display card.

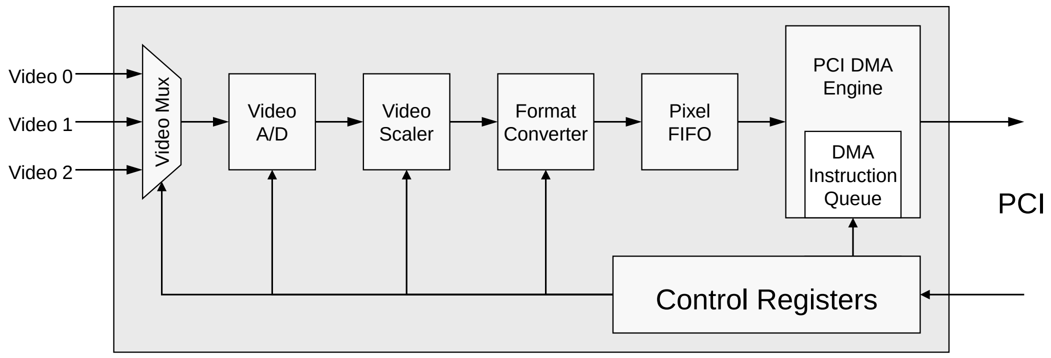


Figure 6.9: The high-level organization of the BT848 chipset.

An overview of the hardware pipeline that transforms analog video into a raw digital data stream in the BT848 chipset is presented in Figure 6.9. The input to the pipeline can be selected from one of three analog video sources using a programmable multiplexer. This input is transformed into a digital data stream using an analog-to-digital converter. Digital data at this point is represented in a 640x480 pixel frame using the 4:2:2 YCrCb format [25]. The next stage in the pipeline is a video scaling unit that can downsample the data stream to user-defined dimensions. The data is then passed through a format-conversion engine that transforms the 4:2:2 YCrCb image into other formats (e.g., RGB565 or RGB32). Video data is then queued in a series of pixel FIFOs, where data is held until a PCI DMA engine transfers the data to host memory. The DMA engine performs data transfers based on a list of transfer instructions that are assembled by the host's device driver for the card.

Open source device drivers are available in the Linux operating system for BT8x8 video capture cards. These drivers are part of the video-4-linux (v4l) effort [37] and provide a basic API for user-space applications to interact with video capture cards. Because the BT8x8 chipset does not employ an on-card frame buffer, the driver must allocate a block of host memory for housing captured data and program the card with a list of DMA operations to store data into the memory. The driver allows users to create two frame buffers for each card so that the card can write data into one buffer while an application reads data from the other buffer. Using this double-buffered approach, it is possible for an application to acquire 640x480 pixel video data at 30 frames per second (NTSC's frame rate).

6.4.2 Endpoint Construction

In the process of examining how the BT8x8 video capture card can be integrated into the GRIM environment, it was observed that adapting endpoint software to run on the card would be infeasible for three reasons. First, while the card does provide a programmable RISC engine, this engine is primarily designed to simply transfer data between the card and host memory. Second, the card does not provide any memory that could be used for implementing on-card message queues. Finally, the fundamental nature of the card makes it impractical for use as an autonomous endpoint. Outside of configuration operations, the card performs the single function of writing data to a memory location in the host. Therefore it was decided that the integration of the BT8x8 card into the GRIM environment should leverage the existing v4l work and utilize the host as a unit for managing cluster interactions with the card.

Table 6.2: Characteristics of BT8x8 video streams.

Frame Size (Pixels)	320x240	640x480
Bytes/Frame (16b Pixels)	150 KB	600 KB
Bytes/Second (@30 Frames/Second)	4.5 MB/s	18 MB/s
Bits/Second (@30 Frames/Second)	36 Mb/s	144 Mb/s

The software that allows the BT8x8 to be used in the GRIM environment is comprised of initialization functions and active message handlers. The initialization functions connect to the v4l device driver and acquire allocations of host memory for housing video frames. After initialization, users can utilize a number of active messages for controlling interactions with a video capture card. In addition to active messages that allow users to configure the video capture software, a special active message is available for requesting that a frame of video data be transmitted to an endpoint. This request message allows users to specify the active message handler that is used in the reply message that carries the video data. This interface allows users to construct their own interfaces for processing incoming data without having to specify the mechanics of how the video capture card obtains the data. If the user does not specify a handler to use in the reply message, a built-in handler is selected that simply transfers the message's payload to a user-specified memory address.

Additional active message handlers were constructed to allow the node with the video capture card to transmit video data to another node in the cluster using remote memory operations. With these functions a node can have a frame of video data transferred directly to a frame buffer or a display device. Given that the dimensions of a frame of captured video may not match the dimensions of the output display, it is necessary to break the remote memory write operation into a series of smaller transfers. For example, if the frame size of the captured video is smaller than the size of the display, the captured data must be transmitted one row at a time in order for the rows to be rendered properly. The GRIM active message function handlers can perform this operation if the user supplies the dimensions of the target display.

6.4.3 Driver Modifications

The V4L device driver was modified to allow captured video data to be stored to a region of memory specified by the user. These extensions allow users to pass a physical memory address to the driver for a block of contiguous memory that is large enough to house a frame of video data. The GRIM software was also extended to allow users to reserve a contiguous block of NI card memory for housing application data. This effort provided the fundamental means by which the capture card could transfer video data directly to the Myrinet NI card in order to reduce the number of PCI transactions required by a system that needed to stream video data from one host to another. Test applications were constructed to demonstrate that the video card could in fact transfer data directly to the proper location of the NI card.

While the basic mechanisms for streaming video data directly into the NI have been constructed, development was halted due to a lack of practicality. Table 6.2 lists the characteristics of two video streams the BT8x8 card is capable of creating. If the video capture card is configured to store data directly to the NI, a buffer large enough to house two frames of data must be allocated from NI card memory. With only a megabyte of SRAM, the LANai 4 NI would only be able to support the 320x240 pixel video stream. While the LANai 9 could support 640x480 pixel frame buffers, doing so significantly reduces the amount of memory that is available for the NI to buffer normal messages. In contrast, the 640x480 data stream requires only 18 MB/s of bandwidth from the host's PCI bus. Because this data rate is only 14% of the available bandwidth of a 32b/33-MHz PCI bus, it was decided that the extra effort required to reliably stream data directly into the NI was not worth the possible performance gain. Instead, software was designed to store captured video data in an intermediate host buffer and then transfer data to the NI as needed by applications. This method is able to transfer 640x480 pixel data streams in 30 frames per second.

6.5 Video Display Cards

Video display devices are another form of multimedia peripheral device that can be utilized in the cluster environment. While there are numerous commercial video display cards, the vast majority of these cards operate on the principle that display data is housed in an on-card block of memory known as the frame buffer. Writing graphical data into this block of memory results in changes in the rendered output. Therefore video display cards can be incorporated into the GRIM environment by making the frame buffer accessible to end applications. GRIM has been extended with functionality to allow the frame buffer in a host to be identified to end applications. End users can then use this information with remote memory write operations to update the display of a remote host in an efficient manner.

6.5.1 Video Display Card Overview

Modern video display cards generally employ a large (2-128 MB) block of memory known as the frame buffer for housing video display data. The frame buffer accelerates system performance by allowing video data to be stored locally on the video card. The output display engine for the graphics card therefore continuously reads from the frame buffer and uses digital-to-analog converters to generate the appropriate VGA signals. A video display card is typically connected to the host system through the accelerated graphics port (AGP) [47]. This port is similar to PCI, but is situated closer to the host's memory system and has asynchronous transfer characteristics (i.e., 'writes' to card memory are faster than 'reads'). While PCI devices can usually store data to an AGP card with write operations, most motherboard chipsets do not allow a PCI device to utilize read operations to fetch data from AGP card memory.

The Linux kernel provides a simple, universal driver that allows user applications to directly access a video display card's frame buffer. This driver provides a means for an application to map the frame buffer into the application's address space, where it can be updated using normal PIO operations. The device driver also provides basic functionality that allows applications to query the frame buffer's settings. User-space applications can use these calls to determine the dimensions of the display and the current pixel depth. The frame-buffer driver operates regardless of whether or not a window manager is running. Therefore, it is possible for user-space applications to update the graphical display even if X windows software [68] is not running.

6.5.2 GRIM Integration

It is generally infeasible to port communication endpoint software to a video display card due to the architecture of these cards. Video display cards simply render graphical data and therefore are designed to function as data sinks. Thus, the approach taken in GRIM to integrate a video display card into the cluster architecture is to simply present the video display card's frame buffer as a block of memory that cluster applications can update. The first task in accomplishing this goal was adapting GRIM to interact with the device driver that controls the frame buffer. At initialization time, GRIM opens the driver and maps the frame buffer into the application's address space. Next, GRIM utilizes a custom-built device driver to transform the virtual address mapping of the frame buffer into a physical address that can be referenced by the NI card. Finally, the physical address of the frame buffer is shared with other endpoints in the system. These endpoints can then use remote memory write operations with the physical address (RM-P) to render changes to the output display.

An example host-level application was constructed to demonstrate how the frame buffer could be used in the distributed cluster environment. In this application, multiple hosts are equipped with video capture cards and configured to capture live video streams. At runtime, each of these hosts is instructed to transmit its captured video stream to a different area of one host's frame buffer. The result is that multiple video streams can be displayed simultaneously on a single output screen. The advantage of this approach is that the incoming video streams can be routed directly to the output display without buffering the data in the display host's memory. This type of operation can be beneficial in other tasks such as distributed rendering systems, where individual workstations perform the task of rendering different portions of an overall scene.

6.6 Summary

Multiple peripheral devices have been integrated into the GRIM communication environment. This work has been simplified by the fact that GRIM implements a common core of its communication functionality in the NI. Peripheral devices in this environment implement mechanisms to facilitate interactions with resources in the local host such as the NI or other endpoints. GRIM is a flexible substrate for this type of work because it can be adapted for use with peripheral devices that have a wide array of characteristics. The I_2O adaptor integration was the least challenging of these efforts because endpoint software could be ported directly to the card. The RC-1000 endpoint was the most challenging because endpoint software had to be implemented as FPGA circuitry. The video capture and display integration work complete the survey of peripheral device work because they represent devices that cannot natively run endpoint software, and therefore require host-level management. All of these examples illustrate that GRIM is extensible in that it can be adapted with device-specific functionality to allow new peripheral devices to be incorporated into the cluster environment.