APPENDIX C

THE GRIM API

GRIM is a communication library that allows designers to easily construct applications that utilize a cluster's distributed resources. GRIM provides a relatively simple but powerful API that feature two separate programming interfaces that can be utilized concurrently by an application. The first of these interfaces is for active messages. In active messages a sender specifies a function handler that is invoked at the receiver when the message arrives. This interfaces allows users to trigger actions at remote endpoints and is particularly well suited for interactions with remote peripheral devices. The second interface provided by GRIM is one for remote memory transactions. These operations allow an endpoint to send or fetch a block of memory from a remote endpoint. Remote memory transactions are useful for efficiently moving large blocks of memory in the cluster with minimal overhead. GRIM is constructed as a linkable C library that can optionally support POSIX threads. Users can tailor GRIM's behavior through the manipulation of configuration files without having to recompile the application. This appendix describes the basic characteristics of the GRIM API.

C.1 Configuration Interface

GRIM utilizes a small number of configuration files to specify the hardware environment for the virtual parallel processing machine. There are two types of files utilized to specify the configuration. First a single application configuration file serves as a top-level means of configuring the system. This file contains basic high-level information such as the number of nodes to use for the application and the configuration of each node. It is expected that end users will commonly adjust this file to meet the needs of the application. The second category of configuration files is for static information that does not frequently change in the cluster. These files define routing tables for the cluster as well as available hardware resources.

C.1.1 Application Configuration

From a user's perspective there is one configuration file that is central to specifying how the GRIM environment is defined. At initialization time GRIM reads a file specified by the GRIM_CONFIG environment variable to determine its configuration. In the current release this configuration file is located in `grim/config/grim_config`. This file contains the following variables:

- **NUM_NODES**: This specifies how many hosts are available in the system.
- **LCP_FILE**: This variable specifies a file that contains the routing information for the Myrinet network.

After these initial constraints users define the configuration for their cluster, including the resources that are to be utilized in each host. A cluster must be defined with a cluster name, a configuration file for the cluster, and a list of resources to be utilized in the cluster. For example a cluster with two hosts (paris and metz) with a Celoxica card could be listed in the configuration file as:

```
#--The French Cluster--
CLUSTER          French_cluster
CLUSTER_RESOURCE French_cluster_config.txt
ROUTING_MYRI     grim_routes_myri.txt

HOST paris
   USE MYRI
   USE CELOXICA

HOST metz
   USE MYRI
}
```

It is possible for multiple cluster configurations to be listed in the configuration file in order to allow users to easily migrate an application between different clusters without adjusting the configuration file. Before GRIM parses this file it determines the name of the host that it is operating on. It then selects the proper cluster to use from the configuration file by selecting the configuration that contains the host the program is running on.

C.1.2 Cluster Resource Configurations

GRIM utilizes multiple configuration files to specify various information for cluster resources. The application configuration file specifies the location of each of these files. GRIM parses the resource configuration files based on whether the resources are utilized by the cluster. The following files are utilized:

- **Myrinet Routing Table**: Because Myrinet uses source routing it is necessary to define all of the routing information for the cluster in advance. Routes must be selected in a deadlock free manner. Each line in the configuration specifies the source node and the paths to all other nodes in the cluster.

- **Myrinet Host Mapping File**: This file contains the listing of hosts utilized in the cluster and the physical ID of each host. The physical ID is the number that is referenced with the Myrinet routing table because it reflects the port number in which the host is physically connected to the Myrinet switches.

- **Celoxica Circuit File**: The software that manages the Celoxica card requires information describing the FPGA configuration images that can be loaded into the FPGA card. This file lists all of the available FPGA configuration files and notes which circuit is loaded in which user-defined computational slot. Incorporating additional user-defined circuits in GRIM requires that the circuits be identified in the grim_handlers.h header file.

C.2 Initialization: grim_enable()

The GRIM communication library is initialized through the `grim_enable()` procedure. This function performs a number of startup operations for the library and should only be called once by an application. This function must be called before any of the library's variables or functions can be accessed. The `grim_enable()` procedure performs the following internal operations:

- **Reset variables**: The GRIM library begins operation by allocating and resetting all variables. These variables include node information as well as various databases.
- **Parse configuration files**: Each node in GRIM must load information from configuration files to determine information about the global cluster environment. This information contains both general information (e.g., routing tables and host names) and application-specific information (e.g., the number of hosts used in application).
- **Construct a local database of resources**: Each node in the cluster utilizes configuration file information to construct a database for resources in the cluster. This database contains both local and global information, and is referenced by end applications to determine the location of requested resources.
- **Initialize local devices**: The next step for a node in the cluster is to initialize the local set of peripheral devices that are to be utilized in the cluster environment. These initializations are device specific, performing operations such as loading a peripheral device with firmware.
- **Allocate host incoming message queues**: Next GRIM interacts with a device driver to open a block of memory that is both pinned and contiguous. The library obtains both a virtual and physical address for the memory so that it can be accessed by the user-space application and the peripheral devices. This memory is utilized to house incoming message queues.
- **Initialize and link message queues**: After all local peripheral devices and host memory is available, GRIM must establish message queues and initialize message queue pointers for the resources in the local host. The size of each message queue is based on the amount of memory available in the resource to house messages and the number of queues that must share this space. Users can request particular sizes for message queues in the configuration files, although this information is ignored if it exceeds queue capacities. After queues are allocated, the library stores pointer information in the appropriate outgoing message queue registers for each resource.
- **Handler library initialization**: Next, GRIM initializes both the local and global handler databases. Once initialized the database is loaded with a set of built-in function handlers that are available at all nodes.
- **NI synchronization**: All of the NIs for the nodes used in the cluster perform synchronization. In this procedure a NI alerts all other NIs that it is reset and waiting to hear from the other nodes. A NI must wait until it receives a reset notification from all other NIs before it can proceed in a normal mode of operation. This process is necessary in order to reset the sequencing information used by NI pairs and to guarantee that a NI will not transmit data to a node that has not been initialized.

- **Peripheral device activation**: The final step in initialization is to notify all peripheral devices that the node is fully initialized and that operation in the cluster is to begin.

Once `grim_enable()` completes, host applications can begin utilizing the library.

C.3 Runtime Information

After initialization, users can obtain basic information about the host an application is running on. Each host in the cluster is assigned both a physical node number (PNN) and a virtual node number (VNN). The PNN is a constant number that is assigned to a particular host. It is utilized internally by the library to manage routing tables. The VNN is a number that is assigned to a host at runtime based on the cluster's configuration file. GRIM assigns VNNs linearly to hosts in the configuration file, starting with VNN 0 for the first host in the file. This approach allows users to easily specify which nodes in the cluster are utilized in the virtual machine, without having to change the files containing physical routing information. End users should always reference host nodes with the VNN in applications for portability.

Table C.1: API for obtaining cluster host information.

`u32 id`	=	`grim_getVNNFromName(string name)`
`string name`	=	`grim_getVNNName(u32 id)`
`u32 id`	=	`grim_getMyVNN()`
`string name`	=	`grim_getMyName()`
		`grim_printVNNConfiguration()`

Users can obtain basic information about the local host an application is running on through the commands listed in Table C.1. The first four of these functions provide either a 32-bit VNN identifier for a host or the host's name. The last command prints out information about the runtime configuration of hosts utilized in the system for an application.

C.3.1 Referencing a Cluster Resource

In addition to referencing hosts in the cluster, users must be able to reference individual communication endpoints. A reference to a communication endpoint is constructed with three pieces of information: the VNN of the endpoint's host, the type of communication device the endpoint is (e.g., host-CPU level, Celoxica card, I_2O card, etc.), and a logical channel identifier to associate transmissions to the endpoint.

Table C.2: Functions for generating a reference to a communication endpoint.

u32	=	**grim_getDestID(**	u32	destination node VNN,	
			u8	destination device,	
			u16	logical channel	)
u32	=	**grim_getResource(**	u32	destination node VNN,	
			u8	device type,	
			u16	logical channel	)

As listed in Table C.2, GRIM provides two functions for generating a reference to a communication endpoint. First, `grim_getDestID()` can be utilized to obtain a 32-bit reference to an endpoint if the location of the endpoint is known in advance by the user. The user must supply the VNN and device type of the endpoint, as well as the desired logical channel for communication with the endpoint. This function is useful for referencing well-known endpoints, such as the host-level endpoint that is available at every node. The second function for generating a reference to an endpoint is `grim_getResource()`. This function is designed to allow users to query the cluster's resource database in order to locate a desired resource. Users can specify a particular VNN with which to restrict the search, or specify that any VNN can be utilized.

C.4 Active Message Interface

The first of two programming interfaces provided in the GRIM communication library is for active message style interactions with endpoints. In this system the sender includes information in each message that specifies how the message is to be processed at the destination endpoint. Each endpoint contains a set of active message function handlers that are used to process incoming messages. An endpoint must register a handler with the communication library before the handler can be utilized by other endpoints in the cluster. During registration users associate a string identifier with a handler and are returned an integer identifier that can be utilized to reference the handler in subsequent API calls. Endpoints invoke actions in other endpoints simply by sending active messages that are properly encoded. Each endpoint is responsible for periodically invoking a polling operation that extracts messages from incoming message queues and processes the messages accordingly.

C.4.1 Handler Registration

Each communication endpoint in the cluster can be equipped with a different set of active message handlers. Therefore it is necessary to provide functions in the cluster that allow endpoints to register their own unique handlers and publish this information to the global cluster context so that other endpoints can reference and utilize the handlers. This process takes place in three phases with the functions listed in Table C.3.

Table C.3: The functions utilized to register and reference active message function handlers.

		`grim_register_handler(`	`handler_call_t function,` `string name`	`)`
		`grim_syncHandlers()`		
`u32`	`=`	`grim_resolveHandler(`	`string name`	`)`

The first part of handler registration takes place when an endpoint registers its function handlers locally with the `grim_register_handler()` function. With this function an endpoint updates a local table that associates a string identifier with a virtual memory pointer to a function handler. This information does not leave an endpoint until the `grim_syncHandlers()` function is executed. The `grim_syncHandlers()` function transmits the local list of function handlers to an endpoint in the cluster which is responsible for managing a global database of function handlers for the cluster. This node will merge the incoming list of handlers into the global database and transmit a copy of the global list of handlers to the endpoint requesting the synchronization. The requesting endpoint will then update its local tables and assign a global integer identifier to every local function handler. Once equipped with this information, an endpoint can call the `grim_resolveHandler()` function to determine the global integer identifier of a function handler in the cluster. This ID can be utilized in active message transmissions with other endpoints.

C.4.2 Send and Receive Operations

As listed in Table C.4, GRIM provides a `grim_send()` operation for transmitting an active message to a destination and a `grim_poll()` function for receiving and processing incoming messages. In the `grim_send()` function the sender must provide a resource reference id for the destination endpoint, the handler id, four active message arguments, and an optional payload. The GRIM communication library will take this message, transfer it in its entirety (performing segmentation and reassembly if necessary), and execute the function at the receiving endpoint using the information provided.

Table C.4: The functions for sending and receiving active messages.

`grim_send(`	`u32`	`destination resource id,`	
	`u16`	`function handler id,`	
	`s32`	`handler argument 0,`	
	`s32`	`handler argument 1,`	
	`s32`	`handler argument 2,`	
	`s32`	`handler argument 3,`	
	`u16`	`payload length,`	
	`u32*`	`payload starting address`	`)`
`grim_poll()`			

The `grim_poll()` function must be performed regularly at the receiving endpoint in order to facilitate the processing of active messages. This function detects that new messages are available for processing in the endpoint's incoming message queues, extracts the message, and performs the specified computation. When POSIX threads support is enabled in GRIM, it is not necessary for end applications to execute `grim_poll()` operations. Instead a thread is dedicated to periodically performing the poll operation.

C.5 Remote Memory Interface

The second programming interface provided in the GRIM communication library is one for interacting with memory located at a remote node. Because host applications and peripheral devices operate with different address spaces (i.e., virtual and physical), the remote memory API must be specific about the types of addresses that it operates with as well as provide mechanisms for translating between address spaces. In GRIM the primary means of referencing a block of memory for end applications is a virtual memory address. Like many communication libraries that perform remote memory operations, GRIM users are only allowed to utilize virtual address spaces that are created with special allocation function calls. These calls allocate and pin the requested memory regions and supply the NI with memory translation information. The GRIM library also provides mechanisms for remote memory transactions with physical addresses. These mechanisms operate with low overhead because address translation is not necessary, but offer no protection if a user supplies the mechanisms with bad addresses.

C.5.1 Managing Memory

The first part of the remote memory interface is designed to assist users in managing memory that can be accessed by the NI. The first call listed in Table C.5 is the `grim_malloc_pinned()` function. This function interacts with GRIM's pinned memory device driver to allocate a sufficient block of memory that is guaranteed to be pinned. Internally GRIM allocates large blocks of contiguous memory and then allocates memory requests from these blocks in order to simplify the number of virtual to physical memory address translation entries loaded in the NI. The result of this function is a virtual address that an application can use as a regular pointer and pass as a reference to other endpoints in the system. When an application finishes using a block of pinned memory it can call the `grim_free_pinned_va()` call to free the allocation.

Table C.5: Function calls for managing memory used by the remote memory interface.

u8*	=	**grim_malloc_pinned(**	u32	number of bytes)
		grim_free_pinned_va(	u8*	pinned virtual address)
u8*	=	**grim_get_pinned_pa(**	u8*	pinned virtual address)
u8*	=	**grim_get_pinned_va(**	u8*	pinned physical address)

In addition to allocating and freeing pinned memory, the GRIM library provides mechanisms for translating addresses. The `grim_get_pinned_pa()` function translates the virtual address of a block of memory allocated by `grim_malloc_pinned()` into a physical address. This address can be used for physical memory transactions that bypass address translation in the NI. The `grim_get_pinned_va()` similarly can be utilized to translate a physical address reference of a pinned memory block into a virtual address. This function is primarily provided for completeness.

C.5.2 Remote Memory Operations

GRIM is designed to provide both send (see Table C.6) and fetch operations for interactions with a remote endpoint's memory. Both of these operations can be supplied with an optional lock to provide a simple form of notification. A lock is simply the virtual address of a 32-bit integer that is allocated from GRIM's pinned memory. In the `grim_sendMemory()` call the sender specifies the ID of the destination endpoint, the source and target virtual addresses of the block of data to transfer and the size of the block of memory. If a virtual address of a lock is supplied, the receiving NI will DMA the value specified in the call's lock value variable into the address after the entire block of memory has been transferred. The `grim_fetchMemory()` function operates in a similar manner, but transfers data from the remote endpoint to the local endpoint. Once the operation completes the NI of the node initiating the fetch operation will transfer a zero into the local endpoint's lock address, if specified.

Table C.6: Functions for transferring data between an endpoint and a remote endpoint.

grim_sendMemory(	u32	destination resource id,	
	u32⋆	target virtual address,	
	u32⋆	source virtual address,	
	u16	number of bytes	)
grim_sendPhysicalMemory(	u32	destination resource id,	
	u32⋆	target physical address,	
	u32⋆	source virtual address,	
	u16	number of bytes	)

GRIM also supports an interface for transferring data directly to a physical address in the remote endpoint's system. When the NI of the receiving endpoint receives such a message it transfers the data directly without any form of virtual memory translation, as none is needed. While this mechanism allows for the operation to take place without the overhead of translation, users must be aware that there is no memory protection employed with this function. Supply erroneous information to this function can easily cause the NI to write data into an unknown memory address in the system, which is likely to crash the host system. The physical memory interface however is useful for interacting with devices such as a video display device's frame buffer.

C.5.3 Reserving NI Memory

Additional functions are provided in the GRIM library to allow NI memory to be directly utilized by applications. While it is expected that most applications do not need such functionality, it is conceivable that some applications may need a temporary place to store data close to the wire (e.g., supporting a frame buffer in the NI). The first function listed in Table C.7 is `grim_reserveNIMemory()` and is designed to allocate a block of memory in the NI. This function **must** be called **before** the `grim_enable()` function is called. After `grim_enable()` is called the `grim_getReservedNIMemory()` function can be utilized to gain both virtual and physical memory address pointers to the region of memory allocated in the NI.

Table C.7: Functions for reserving memory in the local NI card.

`grim_reserveNIMemory(`	`u32`	`size )`

`grim_getReservedNIMemory(`	`u8**`
	`u8**`
		`virtual memory address,`
		`physical memory address )`

C.6 Multicast

GRIM provides support for multicast and broadcast operations. The current implementation is tree based and performs message replication in the NI cards. A multicast tree is referenced in GRIM by a string and an integer value. Once a multicast tree is defined an endpoint can subscribe or unsubscribe from its data distribution. All endpoints in the cluster are allowed to inject messages into a multicast tree.

C.6.1 Multicast Tree Management

Table C.8 lists the API provided in GRIM for managing the multicast distribution trees. The function `grim_findMulticastTree()` is used to determine a unique integer value to reference a particular multicast tree in the cluster. If the requested tree name has not been referenced, the library constructs a new tree and assigns ownership of the tree to the endpoint that first attempted to locate the tree. After a tree has been identified an endpoint can specify that it whishes to be a part of the tree and subscribe to multicast traffic. An endpoint can use the `grim_subscribeMulticast()` function if a multicast tree has been identified or the `grim_subscribeMulticastByName()` function if the tree id is not yet known. In both cases the use must specify which NI logical channels will be subscribing to the multicast messages. The channel list is a Boolean mask. The `grim_unsubscribeMulticast()` function is utilized to remove the endpoint from the multicast distribution tree.

Table C.8: Functions for managing multicast distributions.

`u32`	`=`	`grim_findMulticastTree(`	`string`	`name )`

`u32`	`=`	`grim_subscribeMulticase(`	`u32`
			`u8`
				`multicast id,`
				`channels )`

`u32`	`=`	`grim_subscribeMulticaseByName(`	`string`
			`u8`
				`name,`
				`channels )`

		`grim_unsubscribeMulticast(`	`u32`
			`u8`
				`multicast id,`
				`channels )`

C.6.2 Sending Multicast and Broadcast Messages

Table C.9 lists the functions utilized to inject messages into the broadcast and multicast trees. These functions are identical to the normal active message send functions except that the `grim_sendMC()` function requires the specification of the multicast tree id instead of the destination endpoint, and the `grim_broadcast()` does not require the specification of a destination endpoint. All endpoints are allowed to utilize these functions, whether they subscribe to a multicast tree or not. Any active message function handler can be utilized with these functions, although users must be aware that the sending of a multicast active message may result in the invocation of the function handler at multiple nodes. Therefore users must be cautious when designing active message function handlers that are intended for multicast operations. In the current implementation, the original sending endpoint identification information is not accurately provided to the endpoint invoking the function handler. Therefore if such information is required it is necessary to include it in one of the active message function arguments that are passed with the message.

Table C.9: Functions for injecting multicast and broadcast messages.

`grim_sendMC(`	`u32`	`multicast id,`
	`u16`	`function handler id,`
	`s32`	`handler argument 0,`
	`s32`	`handler argument 1,`
	`s32`	`handler argument 2,`
	`s32`	`handler argument 3,`
	`u16`	`payload length,`
	`u32*`	`payload starting address )`
`grim_broadcast(`	`u16`	`function handler id,`
	`s32`	`handler argument 0,`
	`s32`	`handler argument 1,`
	`s32`	`handler argument 2,`
	`s32`	`handler argument 3,`
	`u16`	`payload length,`
	`u32*`	`payload starting address )`

C.7 Advanced API Functions: TPIL

As a means of provided accelerated performance for applications injecting data into peripheral devices a library has been constructed for x86 host endpoints named TPIL: the tunable PCI injection library. TPIL provides basic mechanisms for transferring data to a PCI device using hardware units that have evolved in the x86 architecture. When users need to add new peripheral devices to the GRIM communication library, it may be beneficial to utilize the TPIL API in order to enhance performance. The API is summarized in Table C.10

Table C.10: The TPIL API for accelerating injections of data into a peripheral device from a host CPU.

Tdev	=	**tpil_create**(	u32	device file id,	
			u8*	device mmap,	
			u32	mmap size,	
			u32	device ioctl	)
		tpil_h2c(	Tdev*	tpil device,	
			u8*	destination,	
			u8*	source,	
			u32	number bytes	)
Tcfg	=	**tpil_benchmark**(	Tdev*	tpil device	)
		tpil_configure(	Tdev*	tpil device,	
			Tcfg*	tpil configuration	)

The function `tpil_create()` is utilized to initialize TPIL and generate a reference that can be utilized in subsequent API calls. This function provides pointers to the file handler of the device and its memory map, the size of the memory map, and a reference to the ioctl() functions that TPIL can utilize to transfer data with the assistance of a kernel driver. The ioctl() calls are optional and are a means of utilizing a card's DMA engines. The `tpil_h2c()` function is utilized to perform host-to-card transfers of data. The user must supply a pointer to where data is to be written (i.e., a pointer to somewhere in the card's memory map), a pointer to the data that is to be injected, and the size of the transfer. TPIL will utilize internal information to determine which transfer mechanism is the best option.

TPIL provides mechanisms for configuring how transfers are performed. First the `tpil_benchmark()` function can be utilized to examine the characteristics of the host machine. Generally this operation is performed offline as it performs several lengthy tests to determine how long it takes to inject data into a peripheral device. The results of these measurements can be exported for use in other programs. The `tpil_configure()` function is utilize to configure the transfer mechanisms for a particular device. Users can either utilize the information obtained from the benchmarking operations or supply custom settings. These settings are simply the cutoffs in which different transfer mechanisms are employed.