CHAPTER VII

STREAMING COMPUTATIONS

Resource-rich cluster computers feature a large number of host CPUs and peripheral devices that can be used by applications as a pool of available resources. A challenge in working with such resource models is constructing applications that effectively utilize the cluster's distributed resources. In this chapter we describe support for a pipelined computing model where accelerators available as peripherals in distinct nodes can be configured through GRIM to operate as a single computational pipeline. Such a model can support a wide range of applications, including streaming media and signal processing applications.

Adapting a message layer to support streaming computations requires an examination of how cluster resources can be utilized as elements in a computational pipeline. A streaming computation is visualized as a connection-oriented service, where a number of operations are performed on data that is passed through a connection. This programming abstraction requires two specific features from an implementation. First, individual resources in a connection must be capable of performing a specified computation on an incoming data stream. In GRIM this functionality can be accomplished through the use of GRIM's built-in active message mechanisms. Second, a resource in a connection must be equipped with mechanisms for forwarding computational results to the next resource in the connection. This functionality is implemented in GRIM through both the library's native reliable delivery mechanisms and a programmable *forwarding directory.* This directory allows users to configure the exact functionality of a streaming operation in flexible manner.

As a motivating example the Celoxica RC-1000 FPGA endpoint has been adapted to support streaming computations. In addition to equipping the RC-1000 endpoint with a forwarding directory, several enhancements were made to the endpoint's architecture. These modifications include a virtual memory system that allows on-card memory to be shared by applications and a system for dynamically reconfiguring the FPGA with hardware circuits needed at runtime by applications. This chapter provides implementation details of the streaming computation extensions, as well as performance measurements of the RC-1000 endpoint that relate to the streaming environment.

7.1 An Overview of Streaming Computations

In pipelined implementations a complex computational task is divided into a linear series of subtasks that can be performed by individual resources. Each resource is then configured to function as a pipeline stage, performing a specified computation on incoming data and forwarding the results to the next resource in the pipeline. The benefit of this approach is that when streams of data are injected into the pipeline, it is possible for the pipeline stages to concurrently operate on different portions of the stream. The desired result is that the system is capable of producing output results at the same rate that data is injected into the pipeline.

Multimedia applications provide a strong motivation for systems that are capable of performing high-throughput streaming computations. In a number of these applications raw multimedia data streams must be processed in real time. Unfortunately it is often

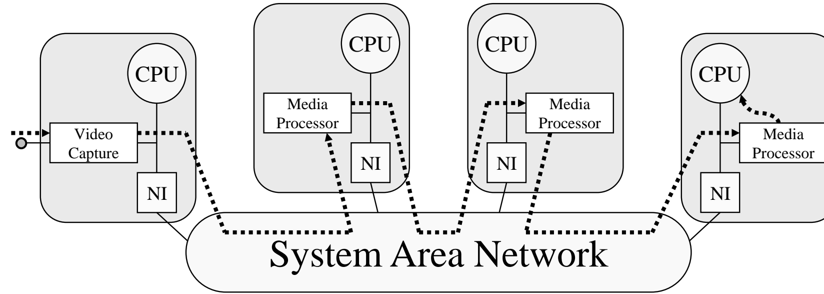

Figure 7.1: A streaming computation example.

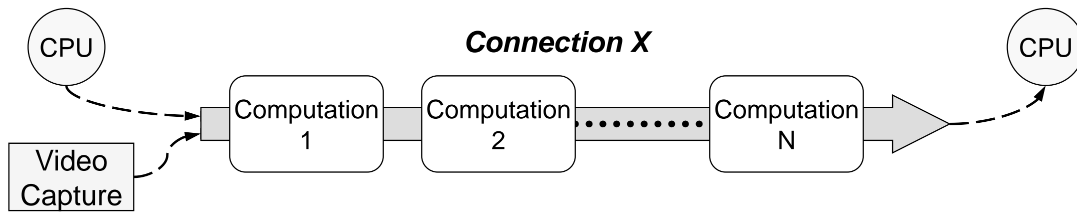

Figure 7.2: An example of a connection-oriented streaming computational pipeline.

infeasible to use a single host computer to perform this processing because of the high data rates that are involved and the computational complexity of the operations that need to be performed. Therefore it is beneficial if a series of resources in a cluster can be utilized as a streaming computational pipeline. An example of such a pipeline is depicted in Figure 7.1. In this system a video capture card generates a video stream that is relayed through multiple peripheral devices distributed throughout a cluster. The devices perform specific operations on the data stream until the data is properly prepared for consumption by a host-level application.

7.1.1 Connection-Oriented Streaming Computations

While streaming computations can be implemented in a variety of manners, a particularly useful abstraction is to visualize a streaming computation as a form of a connection-oriented service. In this abstraction data injected into a connection is processed by a series of computational stages that are defined when the connection is established. As Figure 7.2 illustrates, any endpoint in the cluster can inject data into a connection, but computational results are only transmitted to a single endpoint. A new connection can be created by any endpoint in the system. After obtaining a unique identifier for a new connection, an endpoint must configure the individual resources that are to be used in the connection. Configuration information specifies the operation a resource should perform as well as where the results of an operation should be transmitted in the cluster.

There are multiple benefits to implementing a streaming computation as a connection-oriented service. First, connection-oriented communication is well understood by programmers and is therefore a programming abstraction that can be adopted without much difficulty. Second, this approach can be used in a flexible manner to implement a variety of useful computational systems. For example, users can chain multiple connections together

by forwarding the results of one connection to another. Therefore users can construct complex operations by using a set of basic operations as building blocks. Finally, computational connections provide a simple programming abstraction that allows any endpoint to invoke complex operations without having to know the underlying mechanics of the connection. This feature is especially valuable for simple peripheral device endpoints in resource-rich clusters.

7.1.2 An FPGA-based Pipeline Unit

GRIM has been extended with software to support a connection-oriented form of streaming computations. In this effort peripheral devices can be configured to function as the pipeline stages of a streaming computation. Of the peripheral devices that are currently supported in GRIM, the most attractive device for this work is the Celoxica RC-1000 FPGA card discussed in the previous chapter. This card is a natural candidate for use in streaming operations because it is designed to function as a computational accelerator. Therefore the current RC-1000 endpoint implementation has been modified to support streaming computations. These modifications are implemented as extensions to the FPGA's frame, which is the block of logic that implements the endpoint state machines for the RC-1000. While the focus of this chapter is on implementation details for adapting the RC-1000 endpoint for streaming computations, other endpoints can be extended with this functionality in a similar manner.

7.2 Pipeline Computations

The first of two functional requirements for an endpoint to behave as a pipeline stage is for the endpoint to be capable of performing a predefined computation on incoming messages for a data stream. This functionality can be implemented in a relatively straightforward manner using GRIM's active message programming interface. For the RC-1000 FPGA endpoint, the active message function handler is used to select the computational circuit that processes an incoming message for a data stream. Observing that FPGAs have a limited capacity for housing computational circuits, the RC-1000 FPGA endpoint has been extended with software that allows the FPGA to be dynamically reconfigured with different circuitry as needed by applications. The FPGA frame in this approach detects when it does not have the circuitry necessary to process a message and signals a *function fault* to the host. The host software is designed to resolve these faults, allowing the FPGA to be reconfigured on demand as needed.

7.2.1 Using Active Messages to Control Pipeline Computations

An endpoint that functions as a pipeline stage in a streaming computation must be configured to perform a user-specified operation on a data stream's incoming messages. This functionality can be accomplished through GRIM's active message programming interface. In this approach a message arriving at a pipeline stage is labeled with a stream identifier and an active message handler that specifies the operation the endpoint should perform on the message. Since all pipeline processing instructions are included in an incoming message, it is necessary for the endpoint transmitting the message to format the message. While it may seem counterintuitive to have to place an endpoint's processing instructions at the preceding endpoint in the pipeline, doing so simplifies the configuration process. In this system forwarding information (used to transmit results to the next pipeline stage) and

processing instructions (used to specify the operation the next pipeline stage performs) are stored at the same location (the preceding pipeline stage).

Most endpoints can easily be adapted to perform the computational part of streaming operations because this approach relies on the existing active message infrastructure. Similar to other messages, endpoints simply process streaming computation operations by executing the proper active message function handler. For the RC-1000 FPGA endpoint the FPGA frame's current active message interface is sufficient for implementing this functionality. Messages arriving at the RC-1000 endpoint for a streaming computation are examined by the FPGA frame and processed using the user-defined circuit that matches the arguments specified in the message's header.

7.2.2 Dynamic FPGA Circuit Management

One of the difficulties involved in utilizing an FPGA as a computational resource is that each FPGA is only capable of housing a limited amount of user-defined circuitry. While the industry is constantly increasing the gate capacity of commercial FPGAs, it is unlikely that a single FPGA will ever be able to house all of the computational circuits that could be utilized by end applications. This limitation becomes a significant issue as the number of streaming computational pipelines used in a cluster increases. If these pipelines require diverse types of processing, it is likely that the number of computational circuits needed by the pipelines may outnumber the total space available for housing the circuits in the cluster's FPGA resources. What is needed is a system that can dynamically reconfigure the cluster's FPGAs to emulate the hardware operations that are needed by applications at runtime.

Modern commercial FPGAs generally provide two forms of reconfiguration that can be utilized by software that dynamically manages an FPGA endpoint's circuits. First, all FPGAs support a form of *full reconfiguration*, where an FPGA is reprogrammed in its entirety. Circuit management software can utilize this operation to reprogram an FPGA at runtime with a configuration that contains a circuit that is required by an application. In this approach multiple FPGA configurations are generated offline and stored in a database that the software manages. Second, some FPGAs support *partial reconfiguration*, where a region of the FPGA can be reprogrammed without affecting the rest of the chip. With this option circuit management software can be designed to replace one computational circuit for another. Unfortunately, partial reconfiguration operations can incur significant overheads due to the amount of effort that is required in rerouting an FPGA's active signals. While the FPGA management techniques described in this section can be applied to both forms of reconfiguration, the focus of this effort is on utilizing full reconfiguration mechanisms.

7.2.3 Supporting Function Faults in the FPGA Frame

In order to support dynamic circuit management the RC-1000 FPGA endpoint had to be modified with functionality for assisting the reconfiguration process. These extensions allow the FPGA frame to detect the need for reconfiguration, and provide a means for the FPGA to save and restore its runtime state information during the reconfiguration process. The extensions operate as follows. Whenever the host system loads new computational circuits into the FPGA it stores a list of function ids for the circuits in the FPGA card's SRAM. After the host activates the FPGA the frame pulls these ids and other runtime state information into the FPGA. The frame uses this information at runtime to determine

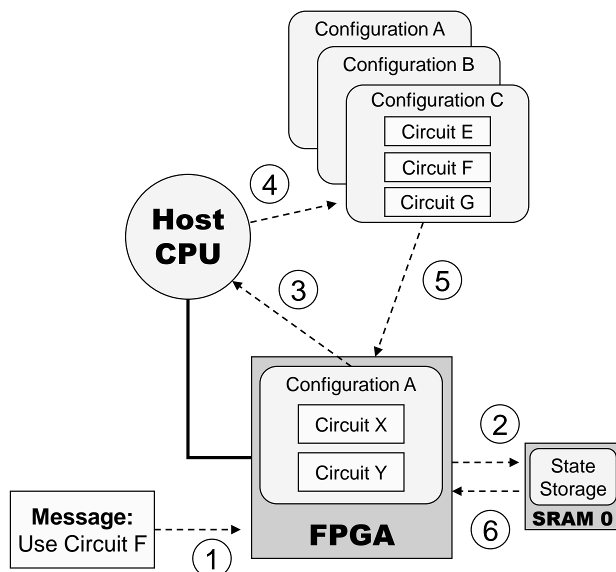

Figure 7.3: The process of reconfiguring an FPGA during a function fault.

if an incoming message can be processed by the FPGA's currently loaded computational circuits. If the FPGA is not equipped with the proper circuits, it initiates a *function fault* that must be resolved by the host's dynamic circuit management software.

Figure 7.3 depicts the steps that are taken during a function fault. After (1) the FPGA detects that it is not equipped with circuitry to process a message it (2) stores its dynamic state information in on-card SRAM. The FPGA then suspends its execution and (3) sends an interrupt request to the host processor. The host software detects the fault, determines which function is needed by the FPGA, and (4) fetches an appropriate configuration from a database. The host (5) loads the FPGA with this configuration and updates the FPGA's list of available circuits. The host then restarts the FPGA which (6) fetches its dynamic state information from SRAM. The FPGA uses this information to begin processing the message that caused the function fault. The message can now be processed because the FPGA is loaded with the computational circuit that is needed by the message.

7.2.4 Function Fault Overhead

Measurements were performed to estimate the amount of overhead that is involved in processing an FPGA function fault. For the FPGA portion of this overhead, the FPGA frame's state machines were examined to determine how many FPGA clock periods are required by the frame to generate and recover from a function fault. Clock periods can be related to wall clock time by dividing the number of clock periods by the FPGA's clock frequency (20 MHz). For the host's portion of a function fault's overhead, instrumentation software was added to the host library to measure the amount of time required to perform fault resolution operations. A P3-550 MHz host was used in these measurements.

The results of the measurements are presented in Table 7.1. While the FPGA operates at a relatively slow clock rate it is able to perform all of its function fault operations in only a few microseconds. Unfortunately there are significant overheads for the host to resolve a fault. The two dominant operations in this procedure are for the host to reconfigure and then

Table 7.1: The amount of time required to manage an FPGA function fault.

Resource	Action	FPGA Clocks	Time (μs)
FPGA	Store queue pointers	6	0.30
	Store missing function	2	0.10
	Trigger function fault	1	0.05
	Release SRAM bank 0	1	0.05
Host	Acquire SRAM bank 0	-	13
	Process fault		8
	Load configuration from file (optional)		10,205
	Reconfigure FPGA		95,114
	Set FPGA clock		2,405
	Set function IDs		2
	Reset FPGA		56,813
	Release SRAM bank 0		7
FPGA	Acquire SRAM bank 0	8	0.40
	Reload queue pointers	4	0.20
	Reload functions IDs	9	0.45

reset the FPGA. The reconfiguration process is time consuming because approximately 700 KB of information must be serially loaded into the FPGA using PIO operations. Resetting the FPGA is time consuming because the operation requires a 50 ms delay for proper execution. Newer FPGA cards will reduce this overhead by a factor of 5-10, with custom architectures doing even better. However, the current model is on par with connection oriented programming models where pipelines are constructed and changed infrequently.

7.3 Pipeline Forwarding

The second operation that a pipeline stage must perform is forwarding computational results to the next resource in the pipeline. This task is an integral part of a streaming computation because it allows a collection of distributed resources to be utilized in a connection. At a fundamental level, forwarding mechanisms should allow pipelines to be constructed in a flexible manner. In addition to routing messages between resources, it should also be possible for users to route data through the same resource multiple times. The benefit of using the same resource to implement multiple pipeline stages is that dynamic application data can be more readily shared among the pipeline stages. One means of constructing a flexible system for managing the transfer of data between pipeline stages is to employ a forwarding directory at each endpoint in the pipeline. A forwarding directory is a user-programmable table that contains information that specifies how a pipeline stage should transmit the results of a streaming computation to the next stage in the pipeline. These tables are easily updated and serve as a simple means by which users can configure both the routing and computational operators used in a streaming computation.

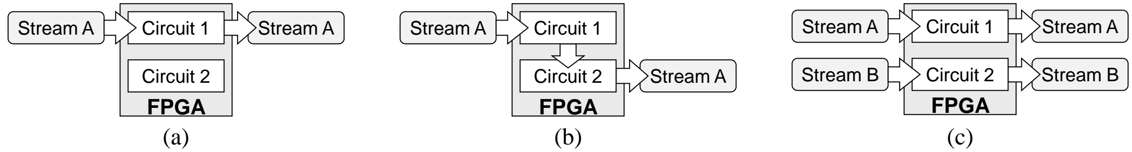

Figure 7.4: Forwarding examples for (a) a single computation on a single stream, (b) multiple computations on a single stream, and (c) multiple streams.

7.3.1 Forwarding

After a pipeline stage generates computational results for an incoming message it is necessary to forward the results to the next stage in the pipeline. Forwarding mechanisms must be flexible enough to be utilized in a number of manners. Figure 7.4(a-c) illustrates three fundamental examples of how data may be forwarded between resources in a pipeline. In the first example (a), a resource is configured to function as a single stage in a pipeline. Results from this operation are forwarded to another resource in the cluster. It is expected that most applications will utilize resources in this manner because the approach is the most straightforward to manage and implement. The second example (b) illustrates a more elaborate case where a resource is utilized to perform two sequential operations in a computational pipeline. This approach requires a means of buffering results at a resource and is beneficial for applications where data locality can be exploited. In the final example (c) a resource is utilized to process data for multiple independent streaming computations. This approach allows a resource to be utilized by multiple applications and requires mechanisms for isolating data streams.

7.3.2 Forwarding Directory

One method by which diverse forwarding operations can be implemented in a streaming environment is to store forwarding information at the resources utilized in a connection. In this approach each endpoint is equipped with a *forwarding directory* that contains information specifying where and how the endpoint should transmit the results of a streaming computation operation. A message arriving at an endpoint contains information that identifies the message as belonging to a particular computational stream. This stream identifier is used to extract information from the forwarding directory that specifies how the computational results of the operation should be formatted for transmission in the network. Therefore users can construct new connections or modify the flow of existing pipelines simply by updating the appropriate forwarding directory entries of the resources that are involved. Updates can be performed using a built-in set of active message handlers that modify forwarding directory entries.

Figure 7.5 illustrates how a forwarding directory at an FPGA endpoint can be utilized as a means of forwarding data from one pipeline stage to another. In this example an active message arriving at the FPGA contains information specifying that the message belongs to computational stream X and requires processing by the FFT active message function handler. After decoding the message's header, the FPGA utilizes an FFT computational circuit to process the payload section of the incoming active message. The results of this computation are stored in the payload section of an outgoing active message. The header for this message is supplied from entry X of the forwarding directory. This header specifies

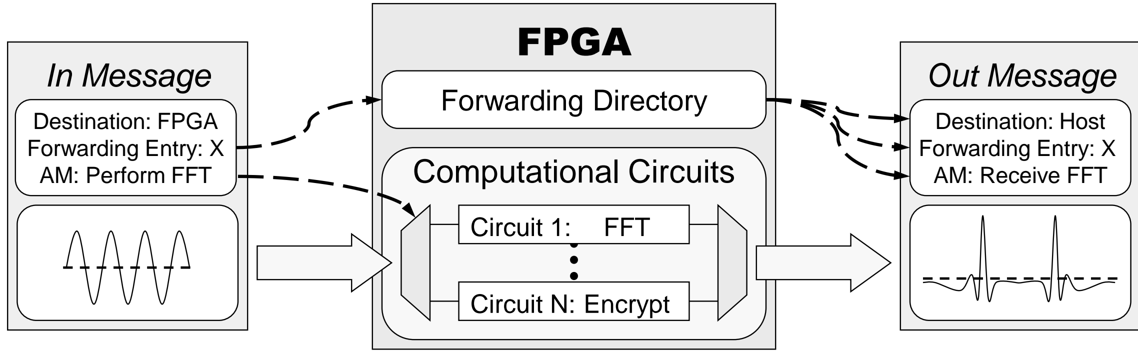

Figure 7.5: The forwarding directory provides information for transmitting a pipeline stage's results to another endpoint.

where the communication library should transmit the message as well as the operation that should be performed at the next pipeline stage.

7.3.3 FPGA Implementation

The RC-1000 endpoint's frame was modified to support a forwarding directory. The directory consists of 256 entries that are stored in the first SRAM memory bank of the card. An entry in the forwarding table is comprised of eight 32-bit words that house all of the values necessary for generating the header of an outgoing message. The frame provides a special active message function handler that allows users to program individual entries of the forwarding table. When the FPGA frame detects the arrival of a new message, it examines the message's header to establish the necessary data paths between resources in the FPGA. Users can store the results of an active message operation in on-card scratchpad memory, a recycling buffer, or in the outgoing message queue. The recycling buffer allows the message generated by one FPGA computation to be routed back to the input of the FPGA endpoint. In this manner a single FPGA can be configured to implement multiple pipeline stages for a computational stream. Forwarding directory performance is included in Section 7.5 as part of the overall performance of the RC-1000 FPGA endpoint when it is used for streaming operations.

7.4 Managing Pipeline State Information

In many streaming applications it is beneficial if application data can be stored at the individual resources utilized in the computational pipeline. This data can include dynamic state information or static arguments such as filter parameters that are used to process incoming messages. Unfortunately peripheral device endpoints have a limited amount of on-card memory for housing application data. As the number of streaming computations using a resource increases, the amount of on-card memory available to each application decreases. Therefore it is beneficial to consider mechanisms that allow peripheral device memory to be shared in a more flexible manner. For the RC-1000 endpoint a basic virtual memory system has been constructed that allows the card's scratchpad memory to be treated as a paged resource. Scratchpad pages are dynamically swapped with host memory as needed by applications. These mechanisms provide a basic form of protection for applications sharing

the FPGA card and allow the endpoint to be transparently utilized by different applications. Similar mechanisms can be implemented for other peripheral device endpoints.

7.4.1 Managing On-Card Memory for an Endpoint

Peripheral device endpoints have a limited amount of on-card memory that can be utilized for housing application data. This memory is valuable to application designers because it allows application data to be stored at the endpoint. In the case where multiple applications utilize the same endpoint, it is necessary to provide some form of management for on-card memory to prevent conflicts between applications.

The simplest approach is to utilize an allocation scheme where each application obtains a block of on-card memory that is exclusively owned by the application. While this method may be suitable for some endpoints and applications, there are three major drawbacks. First, as the number of applications utilizing an endpoint increases, the amount of available on-card memory for each application decreases. Second, applications must be designed to work in a cooperative manner with the memory system. Depending on how memory is allocated, this approach may make it more challenging for applications designers to work with peripheral devices. Finally, this system provides no protection between applications. Therefore an application can erroneously overwrite another application's data.

Another approach to managing on-card memory is to implement a virtual memory system for the endpoint. In this approach card memory is divided into page frames and applications reference on-card memory with virtual addresses. Before the endpoint begins processing a message it determines if the message's memory references can be satisfied with the pages that are currently loaded in the card's page frames. If a page is not loaded the endpoint must replace the current page with the requested data. Unloaded pages can be stored anywhere in the system, although the most practical location is host memory. While page faults for on-card memory can incur substantial overheads, implementing a virtual memory system for a peripheral device provides basic protection for applications that share the device.

7.4.2 Virtual Memory for the RC-1000 FPGA Endpoint

A basic virtual memory system has been constructed for the RC-1000 FPGA card's scratchpad memory. This system operates on a coarse granularity with a virtual memory page being defined as a 2 MB block of SRAM. SRAM memory banks 1 and 2 of the RC-1000 are therefore used as page frames for housing virtual memory pages that can be accessed by user-defined circuits. Incoming messages that utilize scratchpad memory reference data with a virtual memory address. This address is comprised of a page identifier and an offset into the page. Before the frame begins processing a message it examines the page identifiers of the virtual memory addresses supplied in the message to determine if the page is currently loaded in one of the two page frames. If a message's pages are loaded the frame establishes the necessary data paths for the computational circuits to access the memory. The offset value of the virtual address is used as the starting address within the page for accessing data.

If a requested page is not loaded in one of the page frames, the FPGA frame must invoke mechanisms for loading the proper data into card memory. Figure 7.6 illustrates the organization of the memory system used in this procedure. First, the FPGA frame stores the missing page identifier in SRAM. It then suspends the FPGA's execution and sends the

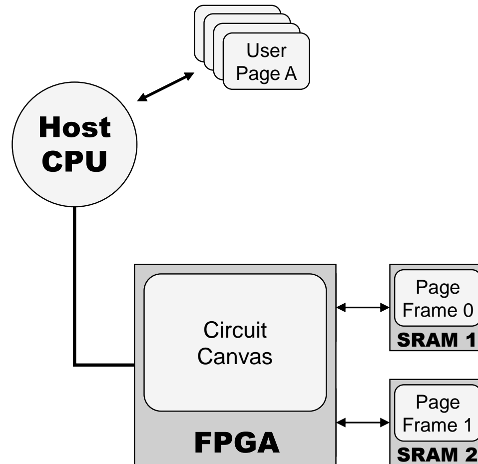

Figure 7.6: A virtual memory system is implemented for on-card SRAM. SRAM banks 1 and 2 serve as page frames for an application's scratchpad data. Unloaded pages are swapped into host memory.

host an *FPGA page fault* signal. The host receives this signal, determines which page frame needs to be updated, and then performs the necessary page swap. A page swap involves transferring the page frame's current data to a buffer in host memory and then transferring the desired page from host memory to the card. After a swap the host updates the FPGA's list of loaded page identifiers and restarts the FPGA. The restarted FPGA loads the new page identifiers and continues processing the message that originally caused the fault. From the user's perspective these operations take place automatically in a transparent manner.

7.4.3 Page Fault Performance

The main drawback to implementing a virtual memory system for a peripheral device endpoint is that there can be significant overheads in resolving page faults. In addition to using the host CPU to resolve a fault, large blocks of data must be transferred to and from host memory. Performance measurements were made of the RC-1000 endpoint software to determine how much overhead is involved in a page fault.

The results of the page fault measurements are listed in Table 7.2. As these tests reveal the most time consuming portion of this procedure is the transfer of scratchpad memory pages between the card and host memory. The differences between loading and unloading a page are due to the fact that in the current implementation the load operation is performed by a zero-copy DMA while the unload operation is performed by a one-copy DMA. Based on these measurements, page faults are expensive operations in this implementation. In order to reduce the number of page faults that take place at run time, users should implement exclusive ownership mechanisms for the card that guarantee that only one application will utilize the RC-1000 endpoint for a period of time.

There are several means by which the virtual memory system could be improved for this card. First, the page size could be reduced in order to allow multiple pages to be stored in the scratchpad memory banks. This approach allows the data sets of multiple applications to be concurrently loaded in card memory, thereby reducing the frequency of page faults. Another interesting approach is to physically attach and utilize a storage device

Table 7.2: Overhead for managing an FPGA page fault.

Resource	Action	FPGA Clocks	Time (μs)
FPGA	Detect fault	1	0.05
	Store page ids	3	0.15
	Issue fault signal	1	0.05
	Release SRAM banks 0-2	1	0.05
Host	Acquire SRAM banks 0-2	-	13
	Process fault		8
	Unload 2 MB page		43,494
	Load 2 MB page		17,927
	Notify FPGA		1
	Release SRAM banks 0-2		7
FPGA	Acquire SRAM banks 0-2	8	0.40
	Reload page ids	2	0.10

to the FPGA card for housing unloaded pages. The RC-1000 card features a large number of I/O pins that can be utilized to attach a hard drive or other storage devices such as flash memory. A disk controller can be constructed in the FPGA for managing disk interactions. Therefore page faults could be managed entirely by the card, swapping card memory to disk without the intervention of the host. The downside of this system is that it is challenging to implement and adds to the overall complexity of the FPGA frame.

7.5 Performance of FPGA as a Pipeline Stage

Measurements were performed to determine how much overhead is involved when the RC-1000 FPGA is utilized as a pipeline stage in a streaming computation. In these experiments message data arrives at the RC-1000 endpoint from either the host endpoint or the NI card. Messages contain 4 KB of payload data (i.e., 1024 words of 32b data) and specify a pass operation for the active message handler. This operation simply transfers the incoming payload data to the outgoing message's payload. FPGA clock times are extracted directly from the state machines and related to wall clock time by dividing clock periods by the FPGA clock speed (20 MHz).

The results of the measurements are listed in Table 7.3. Starting with the resources that inject the message into the RC-1000 endpoint, it is clear that the NI can insert messages into the RC-1000 more efficiently than the host endpoint. This is because the NI controls the RC-1000's memory arbitration mechanisms and the NI has better control over its injection mechanisms because it directly manipulates a PCI DMA engine.

For the FPGA endpoint, the majority of the overhead in processing the message comes from streaming the individual data values through a computational unit. In this system the fetch, compute, and store operations take place in a pipelined fashion, allowing the operations to overlap. This feature demonstrates how an FPGA can be beneficial for processing data because it illustrates how custom pipelines can be constructed in the hardware to achieve high throughputs. It is important to note that the store operations require 3 clock cycles in the current implementation, as opposed to reads which can fetch a new data

Table 7.3: RC-1000 overhead involved in processing a 4 KB message.

Resource	Action	FPGA Clocks	Time (μs)
(Host/NI)	Acquire SRAM bank 0	-	(13 / 5.5)
	Inject 4 KB message		(107 / 32)
	Release SRAM 0		(7 / 3)
FPGA	Acquire SRAM banks 0,3	8	0.40
	Fetch incoming message header	7	0.35
	Fetch forwarding information	5	0.25
	Fetch payload data	1024	51.2
	Computation latency	1	0.05
	Store results	3072	153.6
	Store outgoing header	48	2.4
	Update message queue pointers	3	0.15
	Release SRAM banks 0, 3	1	0.05
FPGA	Acquire SRAM bank 3	-	13
	Perform DMA		69
	Release SRAM bank 3		7

value every clock period. After processing a message the FPGA must format the outgoing message with a header obtained from the forwarding directory. Control is then passed to the host system, which detects the message and initiates the DMA that transfers the message to either the NI or another endpoint in the local host on behalf of the RC-1000 endpoint.

7.6 Summary

Streaming computations are a means of utilizing a collection of distributed resources to improve the throughput of a complex operation. In this effort, a series of cluster resources are utilized to implement a computational pipeline. While the cluster resources function as the computational stages in the pipeline, the message layer provides the framework for delivering data between the pipeline stages. Each resource in a pipeline is equipped with a forwarding directory that allows users to specify how data flows through the pipeline and the operations that are performed on the data streams. As a means of investigating implementation details, the RC-1000 FPGA endpoint has been extended to support streaming computations. Additional enhancements were made to the endpoint to allow multiple applications to utilize the resource at the same time.