FORTH-ICS / TR-392 July 2007

Network Interface Architecture and Prototyping for Chip and Cluster Multiprocessors

Michael Papamichael

Abstract

As computing and embedded systems evolve towards highly parallel multiprocessors, major research and development efforts are being focused on network interface (NI) architectures that enable efficient interprocessor communication (IPC). This thesis is focused on NI architecture, prototyping and design issues for cluster and chip multiprocessors. This work includes the development of a NI queue manager, key NI design issues with respect to IPC and a proposed NI design well suited to chip multiprocessors.

The first part of this thesis presents the architecture design and implementation of a NI queue manager that supports Virtual Output Queuing, Variable-Size Multi-Packet Segmentation and QFC flow control. To increase the available network buffer space VOQs can migrate to external memory in the form of memory blocks connected in linked-lists. Free-List Bypass and Free Block Preallocation optimization techniques are employed to minimize the required number of accesses to external memory and achieve higher bandwidth. In addition, a novel packet processing mechanism that converts arbitrary traffic segments into autonomous network packets was implemented. Detailed FPGA hardware cost results are presented for each individual module, as well as for three different implementations of the queue manager. Network performance experiments were carried out using the developed queue manager on a FPGA-based prototyping platform and confirmed previous theoretical and simulation results about the behavior and performance of the buffered crossbar switch.

The second part starts with a discussion of fundamental NI design issues that affect IPC. Various approaches and solutions are evaluated based on performance, scalability, reliability and protection. The issues addressed include NI placement, virtualization and protection, address translation, data transfer mechanisms and the software interface. Relevant academic and commercial approaches and solutions are referenced throughout the discussion.

The second part also contains a proposal for a design of a NI that is lightweight and tightly coupled to the processor, making it well suited to future chip multiprocessors. Two powerful communication primitives are offered: Message Queues and Remote DMA. Message Queues are intended for low latency communication, mainly synchronization and control messages or small low-overhead data transfers. Remote DMA minimizes processor involvement in communication, is well suited for bulky data transfers and facilitates zero-copy protocols. Furthermore, the proposed NI supports a versatile protection and security solution, based on the existence of protection zones that can easily be adapted to the specific security requirements of a system.

Network Interface Architecture and Prototyping for Chip and Cluster Multiprocessors

Michael Papamichael

Computer Architecture & VLSI (CARV) Laboratory – member of HiPEAC
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
Science and Technology Park of Crete
P.O. Box 1385, Heraklion, Crete, GR-711-10 Greece
Tel.: +30-2810-391660 Fax: +30-2810-391661
e-mail: papamix@ics.forth.gr, papamix@gmail.com

Technical Report FORTH-ICS/TR-392 – July 2007

Work performed as a M.Sc. Thesis at the Department of Computer Science,
University of Crete, under the supervision of professor Manolis Katevenis

Abstract

As computing and embedded systems evolve towards highly parallel multiprocessors, major research and development efforts are being focused on network interface (NI) architectures that enable efficient interprocessor communication (IPC). This technical report is focused on NI architecture, prototyping and design issues for cluster and chip multiprocessors. This work includes the development of a NI queue manager, key NI design issues with respect to IPC and a proposed NI design well suited to chip multiprocessors.

The first part of this technical report presents the architecture design and implementation of a NI queue manager that supports Virtual Output Queuing, Variable-Size Multi-Packet Segmentation and QFC flow control. To increase the available network buffer space VOQs can migrate to external memory in the form of memory blocks connected in linked-lists. Free-List Bypass and Free Block Preallocation optimization techniques are employed to minimize the required number of accesses to external memory and achieve higher bandwidth. In addition, a novel packet processing mechanism that converts arbitrary traffic segments into autonomous network packets was implemented. Detailed FPGA hardware cost results are presented for each individual module, as well as for three different implementations of the queue manager. Network performance experiments were carried out using the developed queue manager on a FPGA-based prototyping platform and confirmed previous theoretical and simulation results about the behavior and performance of the buffered crossbar switch.

The second part starts with a discussion of fundamental NI design issues that affect IPC. Various approaches and solutions are evaluated based on performance, scalability, reliability and protection. The issues addressed include NI placement, virtualization and protection, address translation, data transfer mechanisms and the software interface. Relevant academic and commercial approaches and solutions are referenced throughout the discussion.

The second part also contains a proposal for a design of a NI that is lightweight and tightly coupled to the processor, making it well suited to future chip multiprocessors. Two powerful communication primitives are offered: Message Queues and Remote DMA. Message Queues are intended for low latency communication, mainly synchronization and control messages or small low-overhead data transfers. Remote DMA minimizes processor involvement in communication, is well suited for bulky data transfers and facilitates zero-copy protocols. Furthermore, the proposed NI supports a versatile protection and security solution, based on the existence of protection zones that can easily be adapted to the specific security requirements of a system.

Keywords: network interface, chip multiprocessors, virtual output queues, hardware linked-lists, variable-size multi-packet segmentation

Acknowledgments

I thank my advisor Professor Manolis Katevenis, who provided guidance through my graduate education, as well as Professors Dionysios Pnevmatikatos and Angelos Bilas for their co-advising and their participation in my Master Thesis committee. I also thank my instructors George Kalokairinos, Manolis Marazakis, Christos Sotiriou and the members of the CARV laboratory for their contribution through discussions and technical support, especially Vassilis Papaefstathiou and Stamatis Kavvadias. Many thanks go to Michael Lygerakis, George Passas, Nikos Chrysos, Angelos Ioannou, George Michelogiannakis, Evangelos Vlachos, Spyros Lyberis and Poulicos Prastacos. Finally, I thank my friends and family, my father Konstantinos, my mother Irene, as well as Marina for their love, support and encouragement.

This work was carried out with the financial and technical support of FORTH-ICS in the framework of the European FP6-IST program through the SIVSS (STREP002075), UNIsIX (MCEXT509595), SARC (FET027648) projects, and the HiPEAC Network of Excellence (NoE004408).

Table of Contents

List of Figures

List of Tables

1 Introduction

As computing and embedded systems evolve towards highly parallel multiprocessors, major research and development efforts are being focused on network interface (NI) architectures that enable efficient interprocessor communication (IPC). This thesis is focused on NI architecture, prototyping and design issues for chip and cluster multiprocessors. The work presented was performed at the Institute of Computer Science of the Foundation for Research & Technology - Hellas (ICS-FORTH) as part of the cross-European projects "Scalable Intelligent Video Server System" (SIVSS) [1] and "Scalable Computer Architecture" (SARC) [2] and consists of two major parts.

The first part is focused on the development of a NI queue manager, which implements Virtual Output Queues (VOQs) and supports Variable-Size Multi-Packet Segments (VSMPS) [3]. The use of VOQs eliminates the performance-degrading head-of-line blocking effect, thus greatly improving the NI's performance and localizing the effects of congestion. Variable-size multi-packet segmentation is well suited to buffered crossbar switches and eliminates padding overheads, reduces crosspoint buffer size and is suited for use with external DRAM memories. The architecture is described in section 2 and the implementation and testing in section 3.

The second part of this thesis discusses various NI design issues and is focused on a NI design well suited to chip multiprocessors, considering various design issues, approaches and alternatives. The design is targeted at a scalable lightweight NI that offers support for many connections and is tightly-coupled with the processor, sharing common local memory. Message Queues and Remote DMA (Direct Memory Access) operations are the two basic communication primitives offered. NI design issues are discussed in section 4 and the proposed NI design is presented in section 5.

1.1 Contributions of this work

The contributions of this thesis will be presented separately for each of the two parts. The first part (sections 2 and 3) presents the architecture and design of a NI queue manager and offers a "proof of concept" implementation for the variable-size multi-packet segmentation scheme [3]. This is important for two reasons. Firstly it proves the feasibility of the design, while also providing a very accurate estimate of the implementation cost in hardware. Secondly, we conducted various network performance tests, confirming some of our group's previous theoretical and simulation results. Moreover the system developed runs on real hardware and is used on a daily basis for various experiments.

The second part of this thesis (sections 4 and 5) makes two contributions. Section 4 presents a comprehensive review of various NI design issues with respect to interprocessor communication; relevant academic and commercial approaches and solutions are referenced throughout the discussion. Section 5 provides the detailed design of a NI well suited to future chip multiprocessors. It also deals with security, protection and virtualization issues and includes an analysis of the supported communication primitives, the software interface and a full description of the state that needs to be kept by the NI.

1.2 Outline

The remainder of this thesis is organized as follows. Section 2 provides the detailed architecture of a NI queue manager with VOQ and VSMPS support. Based on this architecture, section 3 presents the implementation of a queue manager prototype along with hardware cost and performance results. Section 4 discusses various NI design issues with respect to interprocessor communication, providing citations to previous work and approaches. Section 5 presents a NI design for chip multiprocessors, while considering various design issues, approaches and alternatives. Finally section 6 contains the conclusions of this thesis and future work directions.

2 NI Support for Variable-Size Multi-Packet Segments

This section presents the design and detailed architecture of a NI queue manager, which implements Virtual Output Queues (VOQs) and supports Variable-Size Multi-Packet Segments (VSMPS). Some basic introductory terms and key concepts are explained in section 2.1. Section 2.2 elaborates on network traffic segmentation schemes and section 2.3 presents the actual architecture of the design, as well as design considerations.

2.1 Key Concepts

This subsection explains some introductory terms, key concepts and approaches, which are essential in understanding sections 2 and 3. This includes an introduction to flow control, QFC credits, Head-of-Line Blocking, Virtual Output Queues and Hardware Linked lists.

2.1.1 Flow Control

In buffered networks flow control refers to the process of managing network buffer occupancy in order to avoid overflows and guarantee that no packets are dropped[1]. When two nodes communicate, buffer overflow can occur when the sending node is producing traffic at a higher rate than the rate at which the receiving node is consuming it. Flow control is usually achieved by providing feedback to the sending node in order to adjust the transmission rate accordingly.

2.1.2 QFC-Style Credit-Based Flow Control

There are various ways to implement flow control in a network depending on the kind of feedback that is sent. In credit-based flow control the receiving node sends credits back to the sending node to inform about the available buffer space. One specific form of credit-based flow control is Quantum Flow Control (QFC) [4]. In QFC, credits are accumulated at the receiving node and are only sent, in the form of a cumulative credit count modulo some power of 2, when they exceed a given threshold. In addition to providing all of the benefits of credit-based flow control, such as lossless and efficient communication, QFC is also robust, since it can tolerate lost or corrupt feedback messages.

[1] This is true for lossless flow control

2.1.3 Head-of-Line Blocking

A buffered network switch usually consists of an ingress line card with input queues, a switching fabric and an egress line card with output queues. Because of the FIFO nature of the input queues, the switching fabric can only switch packets at the head of an input queue at any given time. When a packet of a certain queue at the input cannot be switched to an output port because of contention, the rest of the packets in that queue are blocked by that Head-of-Line packet, even if there is no contention at the destination output ports for those packets. This phenomenon is called Head-Of-Line (HOL) blocking and may lead to significant network performance loss, particularly throughput limitation [5].

Figure 2.1 illustrates the head-of-line blocking effect. The head packets of inputs 1 and 2 are both destined to output 1. However they cannot be both transmitted at the same time. Assuming that input 1 is chosen to transmit its head packet, the second packet of input 2, which is destined to output 2 will be blocked, although there is no contention for output 2.

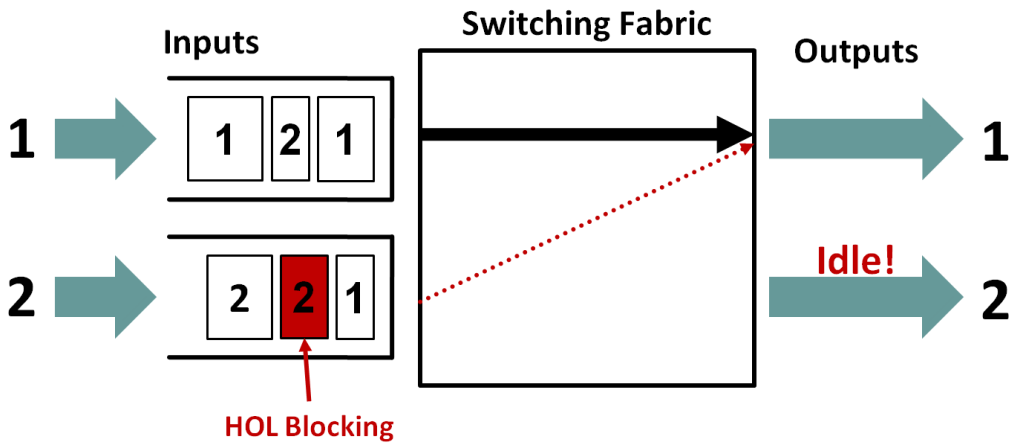

Figure 2.1: Head-of-Line Blocking

2.1.4 Virtual Output Queues

As explained previously the use of a single queue for all outgoing traffic regardless of destination leads to head-of-line blocking resulting in significant performance-degrading effects. In order to overcome the head-of-line blocking drawback multiple virtual output queues (VOQs) – usually at least one per potential destination - are introduced. When using VOQs instead of having one queue at each switch input, multiple queues are used and incoming traffic, consisting of network packets is sorted and stored in the different queues according to its destination, as seen in Figure 2.2. Potentially, the number of queues per switch input can grow to be as large as the number of available destinations in the network.

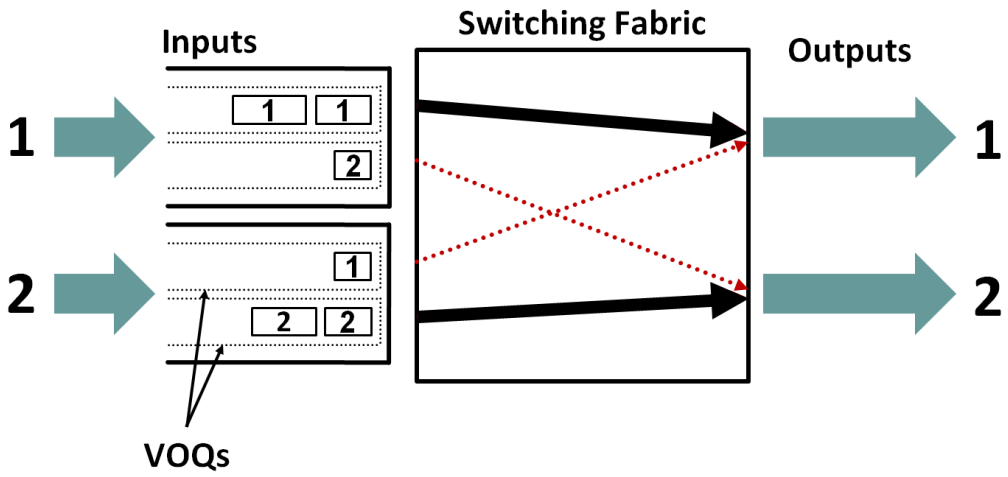

Figure 2.2: Virtual Output Queues (VOQs)

The simple way of implementing a queue in hardware is using a piece of statically allocated memory as a circular buffer, as shown in Figure 2.3. In a system that supports VOQs, if each queue is implemented in this manner, the total memory capacity requirements, even for a modest number of network destinations, can be very large and prohibitively expensive. Moreover, there is a significant probability that the memory used to implement the VOQs will be underutilized, since usually only a few destinations are very popular and their respective queues will use most of their buffer space, while the rest will be almost empty.

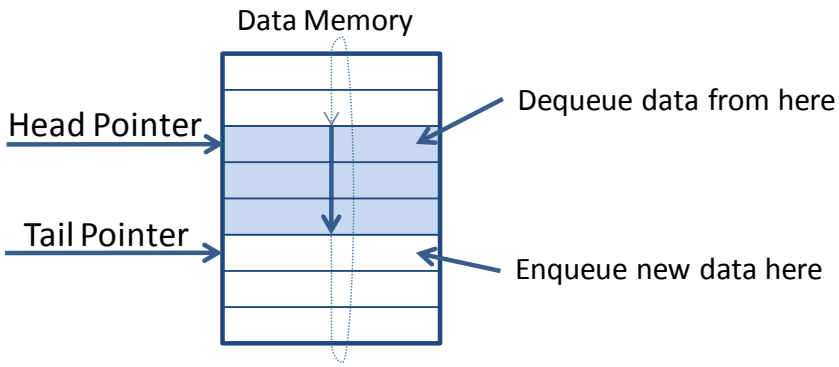

Figure 2.3: Queue implemented as a circular buffer in statically allocated memory

In order to efficiently utilize the available memory resources, VOQs should allocate memory in a dynamic manner according to their specific capacity requirements. However this is not possible if queues are implemented as simple circular buffers. Rather, dynamic resizing of queues is required, to support efficient sharing of a common memory hosting multiple queues. Ideally, each queue should allocate the exact amount of memory it requires for its packets at any given time.

2.1.5 Hardware Linked Lists

A common way of dynamically sharing a memory among multiple queues is dividing the memory in equal, fixed-sized blocks and implementing each queue inside that memory as a hardware-managed linked list of such blocks. Each block is associated with a Next Block Pointer that indicates which block is next in the queue. The Next Block Pointer may be stored either inside the block itself, along the data, or in a separate dedicated memory. In addition to storing a pointer for each block, each queue also requires memory space to store its Head and Tail pointers, which point to the first and the last block of the queue respectively. Figure 2.4 is a diagram of a linked-list in hardware.

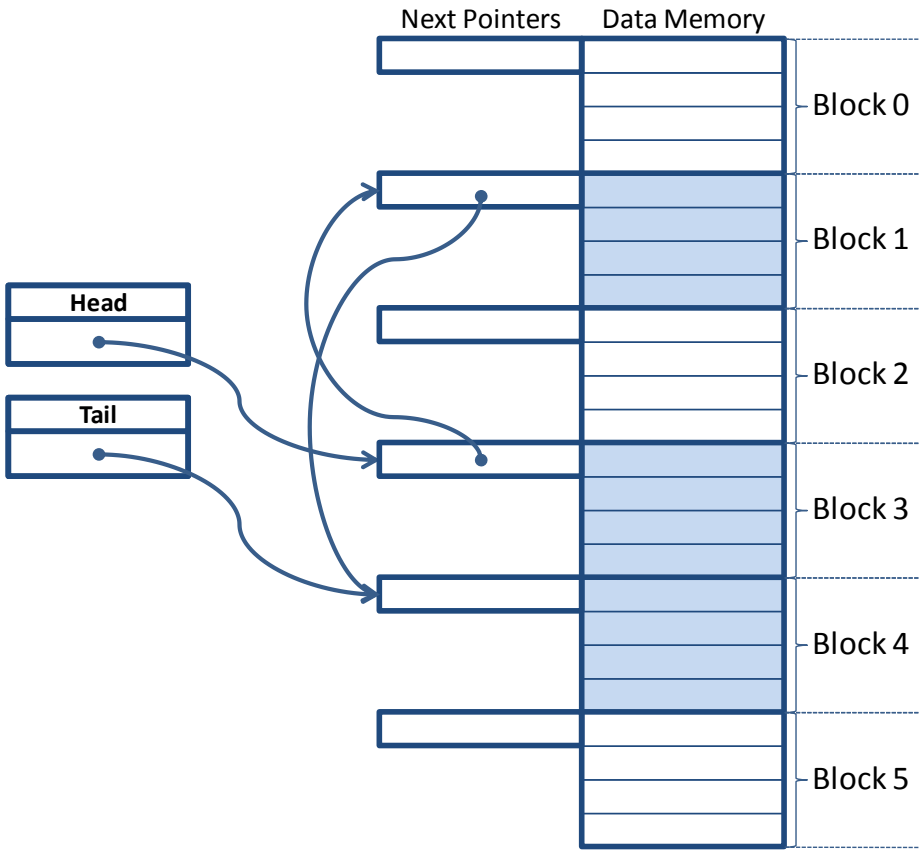

Figure 2.4: Hardware Linked-List

There are many parameters and trade-offs to consider when designing hardware linked-lists. Various alternatives exist for placing Next Block, Head and Tail pointers, ranging from having everything reside in the same physical memory to using a separate physical memory for each item. Many implementation-specific optimizations also need to be considered. For instance, when using DRAM memory it makes great sense to store the Next Block Pointers right next to the data blocks to achieve high DRAM memory throughput. Implementation optimizations are further discussed in subsection 2.3.5.

2.2 Network Traffic Segmentation

Network traffic waiting in VOQs at the input queues of an ingress line card usually consists of variable-size packets. In many cases this traffic needs to be segmented in order to go through the switching fabric. In most cases, this requires reassembly of segments when these exit the switching fabric at the output ports of the egress card. The segmentation scheme used greatly depends on the switching fabric characteristics. In this thesis the network switch is considered to be implemented as a buffered crossbar [6].

Figure 2.5 gives an overview of the segmentation schemes, which are presented in subsections 2.2.1 through 2.2.3 and relate to the work presented in subsection 2.3.

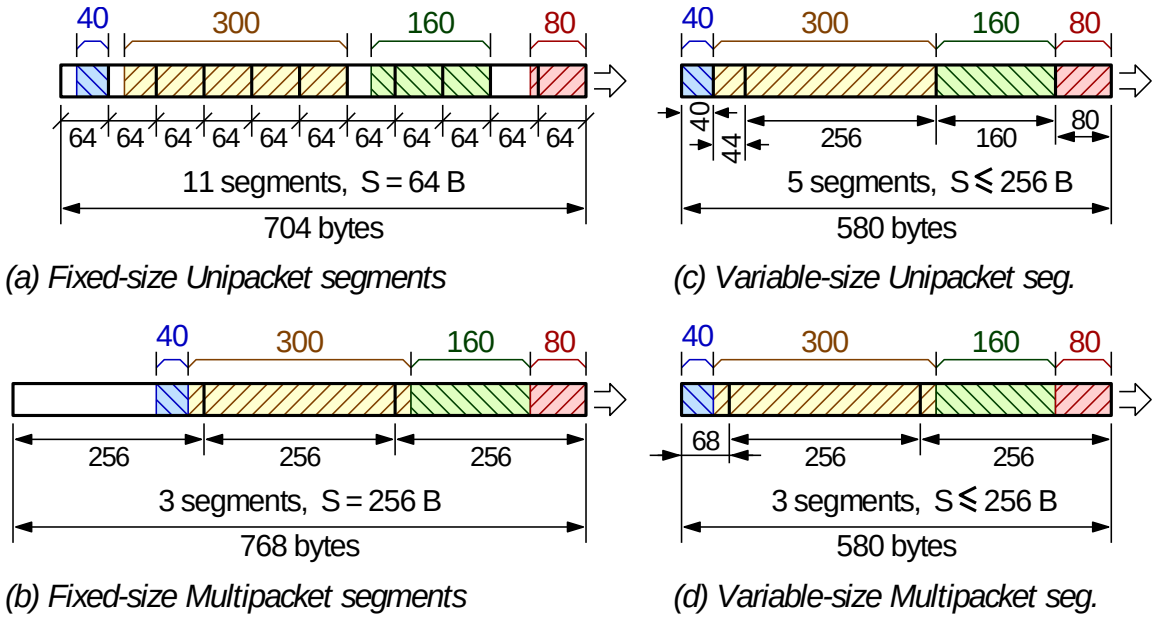

Figure 2.5: Overview of Segmentation Schemes

2.2.1 Fixed-Size Segments

Traditional crossbars, with no buffers at their crosspoints, require segmenting network traffic into fixed-sized segments i.e., cells, since transmissions need to be synchronized [6]. This ensures adequate time for the scheduler to make a scheduling decision and thus makes things simpler.

The traditional approach is to have each segment contain only a single packet or fragment thereof, but this leads to a waste of bandwidth and memory space due to header and padding overheads. Moreover, to accommodate small network packets, segments would need to be equally small, which requires a high-scheduling rate. Fixed-size unipacket segmentation is shown in (a) of Figure 2.5.

An improved approach to fixed-size segmentation is to allow a segment to contain more than one packet or fragments thereof [7]. This allows larger segments, since a minimum-size packet does not need a dedicated segment, but still suffers from padding overheads and could possibly lead to increased delays, because a segment might need to wait for accumulation of more traffic before departing. Fixed-size multi-packet segmentation is shown in (b) of Figure 2.5.

2.2.2 Variable-size Segments

Buffered crossbars, which have small buffers at their crosspoints, are capable of switching variable-size segments. Although the segmentation and reassembly approach is not absolutely necessary for buffered crossbars, since they are capable of directly switching variable-size packets [8], it greatly reduces the required buffer size per crosspoint, increasing implementation feasibility.

One approach to segmentation suitable for buffered crossbars is to have each variable-size segment contain only one packet or part thereof. This completely eliminates padding overhead, but still imposes a high header overhead and is not well suited to maintaining linked lists for storing network traffic, as described in subsection 2.3.5. The variable-size unipacket segments approach is shown in (c) of Figure 2.5.

2.2.3 Variable-Size Multi-Packet Segments

The Variable-Size Multi-Packet Segmentation (VSMPS) [3] is a better segmentation scheme that is well suited to buffered crossbars and storing traffic in linked lists. In the VSMPS approach each variable-size segment contains one or more packets or fragments thereof. This approach combines all the advantages of the segmentations schemes previously described. Variable-size segments completely eliminate padding overhead and packets need not be delayed, waiting for segments to fill up. Multi-packet segments allow larger segment sizes reducing header overhead and providing more slack for scheduling decisions. The VSMPS approach is shown in (d) of Figure 2.5.

2.3 NI Queue Manager Design & Architecture

The queue manager is a hardware module that manages and processes network traffic right before it enters the switching fabric. Packet processing may include traffic segmentation or header processing. The queue manager is also partly responsible for scheduling decisions and network flow control, which is usually done cooperatively with the switching fabric and the

egress card. This subsection describes the architecture of a NI queue manager that supports VOQs, VSMP segmentation and implements QFC credit-based flow control.

2.3.1 Design Constraints/Requirements

The queue manager described in this thesis was implemented in real hardware using FPGA-based boards, as part of the development of a broader research prototyping platform, which is described in more detail in section 3. During the design phase, the specific target hardware and platform characteristics had to be considered and vastly influenced many design decisions.

For instance, the packet processing mechanism was specifically tailored to accommodate the custom packet format used in the rest of the system. Other examples include the scheduler and credit handling mechanisms, which also had to be compatible with the flow control that was already employed in the rest of the prototyping platform. The design was also influenced by specific characteristics of the implementation target board and had to be a accordingly adapted.

2.3.2 Design Overview

Figure 2.6 is a block diagram depicting the modules of the queue manager and the communication and data transfer paths among them. The thick yellow arrows show the network packet flow through the various modules. The solid black thick and thin arrows are used to depict the flow of network data and control information through the modules respectively. The host module on the left represents a node generating traffic, while the Network module on the right accepts traffic and forwards it through the switching fabric. The Network module periodically sends flow control information to the Scheduler.

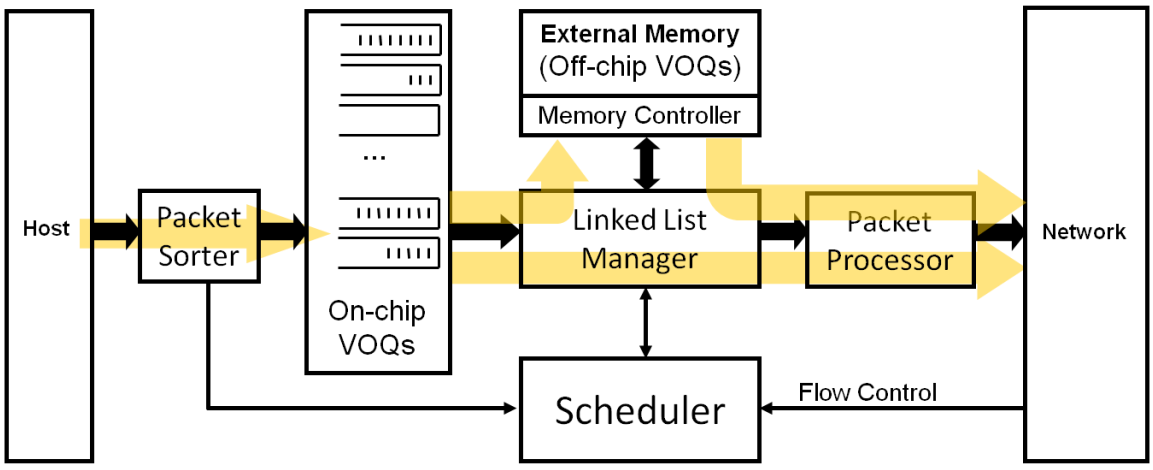

Figure 2.6: Block Diagram of NI Queue manager

The design of the queue manager consists of 5 modules, which are presented in more detail in the following order:

- Packet Sorter
- On-Chip VOQs
- Scheduler
- Linked List Manager
- Packet Processor

2.3.3 Packet Sorter

The Packet Sorter receives network traffic in the form of packets through an elastic buffer and enqueues them, according to various types of header information into the appropriate on-chip VOQ. When necessary, the Scheduler is notified about the incoming traffic.

There are quite a few alternatives as to how the packets are sorted. The most straight-forward scheme, which is also used to eliminate Head-of-Line Blocking, is to sort the packets according to their final destination. Other schemes could take into account other information found in the packet header, such as priority. If the available VOQs are fewer than all of the potential packet destinations, predefined sets of destinations can be grouped in order to share a common VOQ.

Aside from sorting the packets, the packet sorter module is also well suited to simple, light-weight packet processing tasks, such as enforcing a maximum packet size. All packets that exceed the maximum size are then split into a number of smaller packets. If the system also demands the existence of a minimum packet size, this has to be taken into consideration as well, as seen in Figure 2.7.

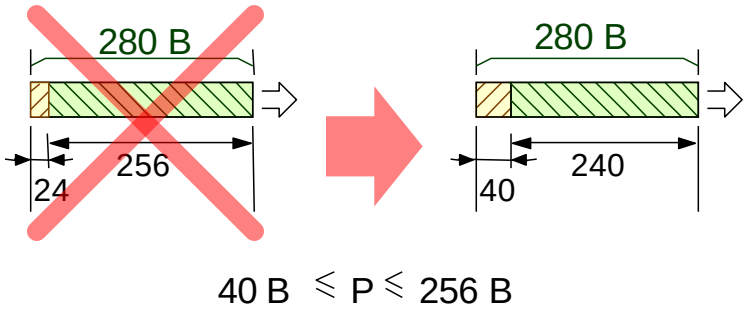

Figure 2.7: The existence of minimum packet size influences packet segmentation. For instance, if the maximum packet size is 256 bytes and the minimum packet size is 40 bytes, then a packet of size 280 bytes would have to be split into two packets, one of size 240 bytes and another one of size 40 bytes.

2.3.4 On-Chip VOQs

The On-Chip VOQ module stores network packets until they are ready either to travel towards the switching fabric or to be stored in DRAM due to congestion. The On-Chip VOQs are hosted in on-chip memory – usually SRAM - as their name implies.

One approach to implementing On-Chip VOQs is through use of statically allocated circular buffers. This can be done using either a single memory block or many smaller memory blocks, one per VOQ. A different approach is to implement hardware linked lists in a single on-chip memory block that is dynamically shared among all VOQs. The first approach requires more memory, but results in simpler hardware, while the second approach requires less memory but more complex hardware.

Choosing the appropriate implementation for the On-Chip VOQs is a trade-off between memory capacity requirements and hardware complexity. The deciding factor is external memory block size, since it dictates the maximum memory capacity requirements per VOQ. In the presented design, the memory required by each VOQ is at least two times the size of an external memory block.

2.3.5 Linked List Manager

As its name suggests, the Linked List Manager's main task is to manage the linked lists of external memory blocks that hold VOQ traffic, which has migrated to external memory due to congestion. This module is also responsible for carrying out segment transfers, as instructed by the Scheduler.

Segment Transfers

The segment transfers performed by the Linked List Manager fall into 3 categories as indicated by respective numbers in Figure 2.8.

1. VOQ segment transfers from on-chip memory to the Packet Processor

Packets received by an empty VOQ that has sufficient credits should be directly forwarded in the form of variable-size segments towards the switching fabric. Under light traffic, and thus low VOQ occupancy, this is expected to be the majority of segment transfers and is the most straight-forward.

2. VOQ segment transfers from on-chip memory to off-chip memory

Packets residing in a VOQ that is out of credits due to congestion may not proceed towards the switching fabric until flow control allows them to do so. If the accumulating pending packets exceed a given threshold, they are transferred to external memory in fixed-sized blocks that are organized in linked lists. Only the tail packets of a VOQ consume on-chip memory, while the rest of the VOQ, head and body, resides entirely in external off-chip memory. On-Chip VOQ packets are expected to migrate to off-chip memory when congestion starts to build up in the network.

3. VOQ segment transfers from off-chip memory to the Packet Processor

These transfers happen when a congested VOQ that has migrated to external memory receives credits. As explained previously, when a VOQ grows excessively large it migrates to external off-chip memory, except for its tail, which remains in on-chip memory. Since the head of the tail resides in external memory, when this VOQ becomes eligible for service, a block needs to be transferred from the off-chip VOQs towards the Packet Processor and thereafter to the switching fabric.

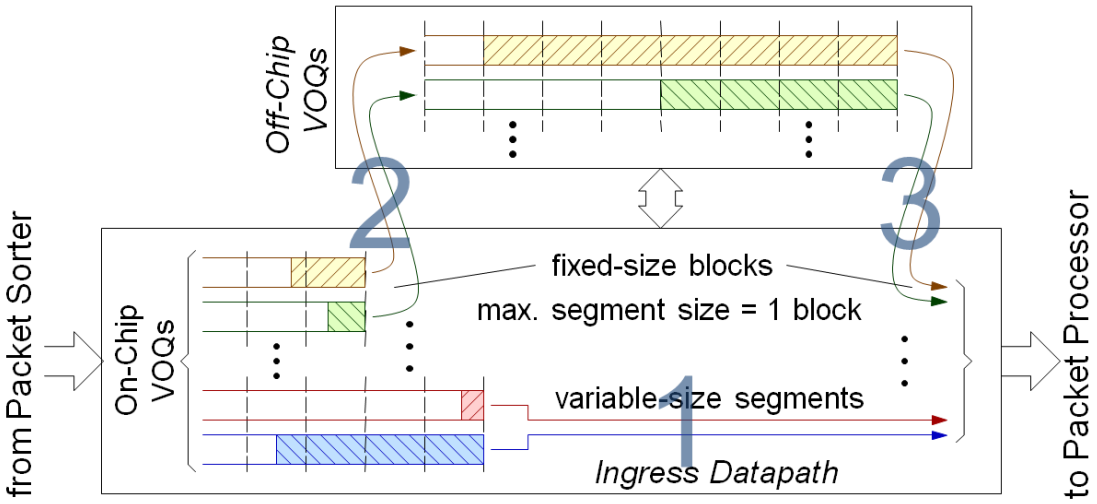

Figure 2.8: Segment Transfer Categories

In the two latter cases, when off-chip memory is involved, the segments transferred are always of fixed-size and are also called External Memory Blocks. External Memory Block Size should be chosen carefully to ensure peak external memory throughput. Block sizes for modern DRAM memories, which are quite commonly used as off-chip memories, range between 256 and 512 bytes. The exact block size is greatly influenced by external memory-specific characteristics, such as burst length and bank interleaving.

Linked List Management

As explained briefly in subsection 2.1.5, linked list implementation in hardware requires keeping track of Next Block, Head and Tail Pointers. The Linked List Manager is responsible for storing and updating this information for each VOQ, as well as for the free block list. The free block list is an additional linked-list used for storing all of the free external memory blocks. A linked list needs to support two basic operations: enqueueing and dequeueing.

The enqueue operation consists of the following actions:

1. Get a free block from the free block list.[1]
2. Write the data in the new block.
3. Read the tail pointer of this VOQ to locate its last block.
4. Update the next block pointer of the last block to point to the new block.
5. Update the tail pointer of this VOQ and have it point to the new block.

The dequeue operation consists of 5 equivalent actions:

1. Read the head pointer of this VOQ to locate its first block.
2. Read the data from the first block.
3. Read the next block pointer of the first block to find the second block.
4. Update the head pointer of this VOQ and have it point to the second block.
5. Put the original first block in the free block list. *

Depending on the type of external memory used, there are a few possible optimizations to consider that can deliver improved performance. Two such optimizations, both employed to reduce external memory accesses, are Free Block Preallocation [9] and Free List Bypass [10].

In Free Block Preallocation each VOQ always allocates a free block ahead of time. Thus, when an enqueue operation is is to be performed, instead of searching the free block list for a new block, the preallocated block is immediately used. This optimization is usually applied when the external memory comprises of DRAM and the next block pointer for each block is stored along with the block data. A minor disadvantage of this technique is that each VOQ uses an extra block of space, which, however, is insignificant for external DRAM.

[1] The free block list needs to be managed in a similar manner.

Free List Bypassing is used to reduce the number of total memory accesses. Assume that there is at least one pending enqueue operation for some VOQ and at least one pending dequeue operation for another or even the same VOQ. Rather that dequeueing a new block from the free block list for the enqueue opearation and enqueueing the used block in the free block list as part of the dequeue operation, these two operations are combined. The enqueue operation uses the block that was just freed for the dequeue operation, completely avoiding free block list manipulation.

In order to implement the above optimizations, the Linked List Manager needs to properly coordinate pending Enqueue and Dequeue requests. A simplified finite state machine that offers support for Free List Bypassing is shown in Figure 2.9. The path indicated by the green arrows corresponds to the state transitions for Free List Bypassing.

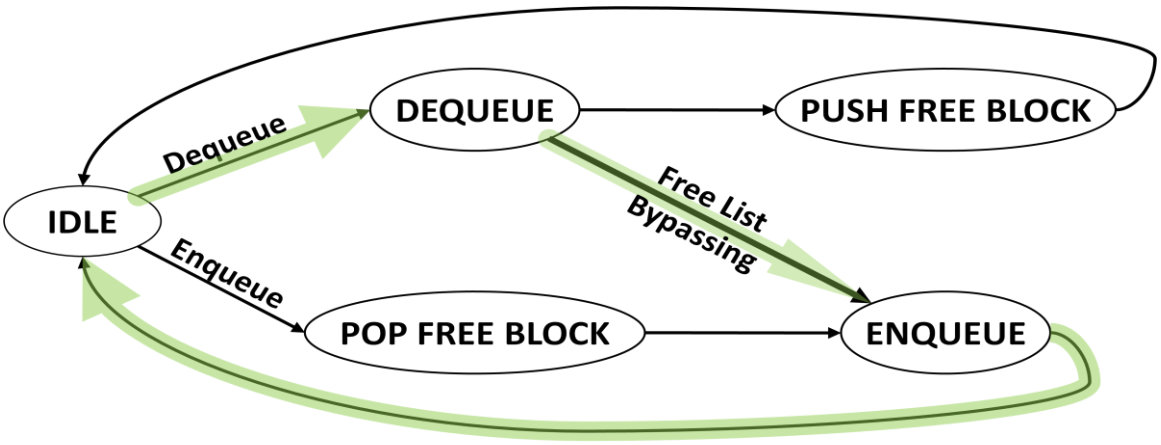

Figure 2.9: Linked List Manager FSM with support for Free List Bypassing

2.3.6 Scheduler

The scheduler keeps track of the state of the VOQ handling system, both on-chip and off-chip, in order to enforce the scheduling policy and implement network flow control. Based on the gathered information and the chosen scheduling policy, the scheduler instructs the Linked List Manager on which VOQ to service next. When a VOQ is almost out of space, the scheduler also implements local flow control by notifying the host interface to stop generating traffic for the corresponding destination.

The Scheduler stores and keeps track of the following information for each VOQ:

- on-chip occupancy (number of data words)

- off-chip occupancy (number of external memory blocks)
- number of sent data words
- available credits

This information is used to enforce a scheduling policy and to implement network and local flow control.

Scheduling - Eligibility

Scheduling refers to determining which VOQs are eligible, with respect to the actions performed by the Linked List Manager, and serving them according to a chosen scheduling policy. As described in subsection 2.3.5, the Linked List Manager is capable of performing VOQ segment transfers that fall into one of the three following categories:

1. VOQ segment transfers from on-chip memory to the Packet Processor
2. VOQ segment transfers from on-chip memory to off-chip memory
3. VOQ segment transfers from off-chip memory to the Packet Processor

VOQs are categorized according to their eligibility for the above transfers. However, a VOQ need not belong to exactly one category. For instance, an empty VOQ that does not have any segments to be transferred does not fall into any category. On the other hand, a VOQ that has a few packets stored on-chip and also has a few credits can simultaneously belong to two categories and segments can be forwarded to either the Packet Processor or the off-chip memory. In such cases it is up to the scheduler to choose the most appropriate action.

Determining eligibility for segment transfers from off-chip memory to the Packet Processor is quite straight-forward. If a VOQ has migrated to off-chip memory due to congestion and at some point its credits are replenished, then the VOQ is eligible for this kind of transfers.

On the contrary, when determining eligibility for the other two types of transfers, there are many parameters to consider, such as the size that a VOQ can grow on-chip before it is considered eligible to migrate to off-chip memory and/or the number of credits and/or the state of the rest of the VOQs.

One simple approach for dynamically allocated on-chip VOQs is to set an occupancy threshold and have each on-chip VOQ be eligible only for one kind of transfer at a time. If the occupancy of an on-chip VOQ is below the threshold it is considered eligible only for segment transfers towards the Packet Processor. Otherwise, if the occupancy becomes equal or greater than the threshold, the VOQ is only considered eligible to transfer segments to off-chip memory.

Assuming that an empty VOQ with plenty of credits suddenly starts receiving traffic, another important issue is determining the appropriate time to form and dispatch a segment from the on-chip VOQs towards the Packet Processor.

One approach is to be "eager" and create minimum-sized segments immediately after traffic starts to appear. This minimizes latency but can lead to continuous production of minimum-sized segments, which impose a greater overhead on the network, since they carry a relatively smaller payload. Another approach is to be "patient" and wait for traffic to build up, until a maximum-size segment is built and dispatched. In this case, network overhead is minimized but latency may be increased, since, when a segment is half full, it is impossible to know when and if the rest of the segment is filled. This problem can be addressed by the presence of a timer, which ensures that traffic will not be stuck in a VOQ forever. When the timer expires, a segment is created and dispatched regardless of its size. This "lazy" approach, however, increases complexity.

Scheduling - Policy

Determining eligibility and enforcing scheduling policy are closely related. The scheduling policy decides the order in which eligible VOQs are serviced. There are many scheduling algorithms to choose from. The more sophisticated the algorithm the more complex the hardware.

One of the simplest and easiest to implement scheduling algorithms that is commonly used in many scientific disciplines is the Round-Robin scheduling, in which eligible VOQs are serviced in order, one after another, transferring at most one segment each at a time. Since segments may be of variable-size, Round-Robin scheduling can lead to "unfairness". For instance, VOQs transferring large segments have an advantage over VOQs transferring smaller segments.

Deficit Round-Robin [11] is a scheduling algorithm that also takes into account the size of each segment transferred. In the long term this scheduling algorithm guarantees that each eligible VOQ transfers an equal amount of data. A positive characteristic of all the Round-Robin-based algorithms is that they are "starvation-free", which means that a VOQ will under no circumstances have to perpetually wait to be serviced.

Other alternatives to the above algorithms include weighted Round-Robin and strict-priority scheduling, as well as other custom scheduling algorithms. For instance one possibility would be to always service the most occupied VOQ. Although this would be unfair and even could lead to starvation of other VOQs, it would minimize total memory occupancy for VOQs implemented in statically partitioned memory.

Choosing the right scheduling algorithm greatly depends on the specific characteristics of the system at hand. On-chip and off-chip memory latency, size and bandwidth, available hardware, network flow-control, traffic patterns as well as quality of service guarantees, all play important roles when choosing which scheduling policy to implement.

Flow Control

There are two categories of flow control management performed by the Scheduler:

i. Network Flow Control

Network flow control is performed by keeping track of the available flow control information for each VOQ, which is periodically received by the network switches. This information, in conjunction with the occupancy of each VOQ, is used to determine the eligibility of a VOQ. Each time a segment is dispatched towards the switching fabric it is the scheduler's responsibility to ensure the existence of adequate credits.

ii. Local Flow Control

Local flow control is about regulating the generation of traffic at the host. This can be simply done by requesting that the host stop generating traffic for a specific VOQ if its occupancy grows larger than a certain threshold. For example, this threshold could be set to be 3/4 of the available on-chip memory space for each VOQ.

2.3.7 Packet Processor

The Packet Processor receives segments, which are parts of a stream of packets "chopped" at arbitrary points originating from either the on-chip or the off-chip VOQs, produces independent packets and then forwards them towards the switching fabric. To accomplish this, it needs to keep track of the most recent header for each VOQ and can insert, remove or alter packet headers to maintain packet integrity. The Packet Processor guarantees that only valid packets will be transmitted towards the network, regardless of the size, content and origin (on-chip or off-chip memory) of an incoming segment.

The employed network packet format greatly influences the capabilities of the Packet Processor module. It is not always possible to build a smaller autonomous packet using just a part of the original packet. For instance, remote DMA packets, whose header contains (among other information) the size and the destination address of the DMA transfer, are moderately easy to split into smaller packets. Conversely, it is usually not possible to split packets carrying small

synchronization or control messages. This module is also well-suited to performing simpler packet processing tasks, such as static packet header manipulation.

3 Implementation and Testing

This section describes the implementation and testing of a queue manager based on the architecture described in section 2. An overview of the system is presented in subsection 3.1, followed by a description of the hardware platform in subsection 3.2 and the methodology and implementation details in subsections 3.3 and 3.4 respectively. Subsection 3.5 describes the testing procedures and issues and subsection 3.6 presents the results in terms of hardware cost and network performance.

3.1 System Overview

The work presented in this section was conducted as part of the cross-European research project Scalable Intelligent Video Server System (SIVSS) [1] aiming at the development of a prototyping platform for conducting experiments related to high-speed networking and interprocessor communication. The prototyping platform, which is shown in Figure 3.1, was built at the Computer Architecture and VLSI (CARV) Laboratory of the Institute of Computer Science (ICS), FORTH and comprises of commercial personal computers (PCs) linked through a custom interconnect.

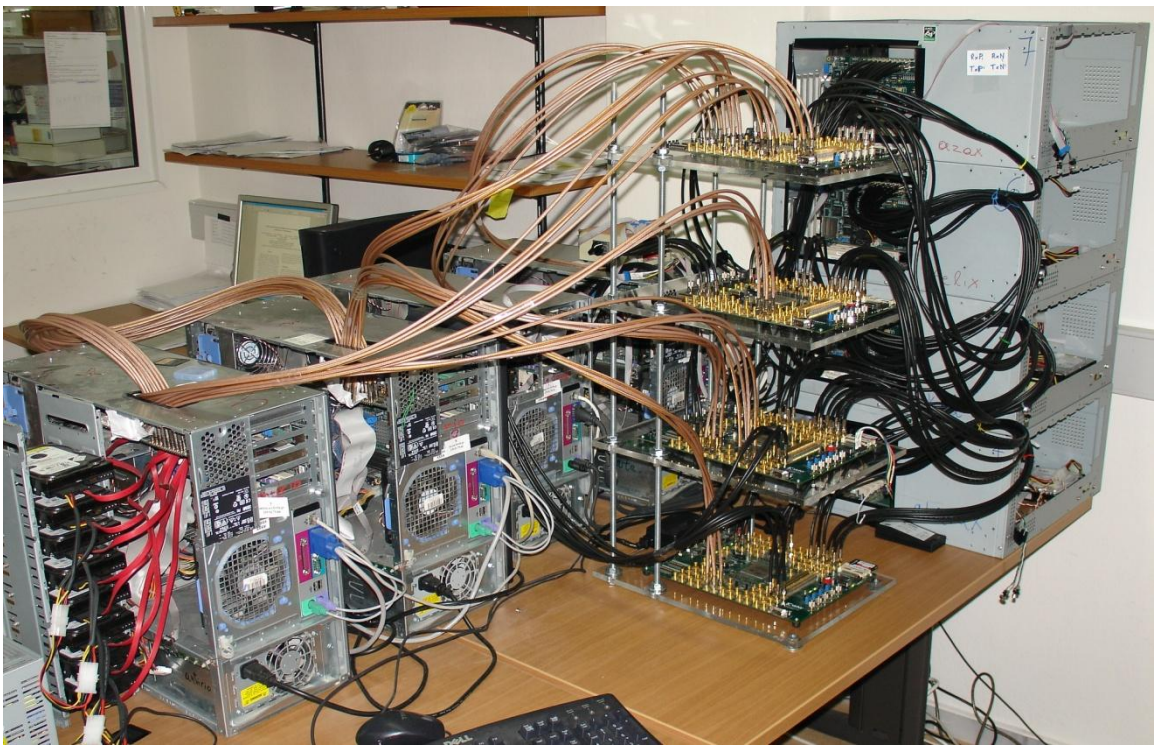

Figure 3.1: Prototyping Platform at the CARV Laboratory of ICS, FORTH

A FPGA development board plugs into the PCI-X bus of each PC, and is configured as its network interface (NI). An additional FPGA board is configured as a buffered crossbar network switch. The key features of this platform are:

- High throughput network
- PCI-X interface
- Remote Direct Memory Access (RDMA) support
- Buffered Crossbar Switch Implementation

3.1.1 NI Overview

The implemented NI card supports the PCI-X interface [12] to communicate with the host PCs and the RocketIO interface [13] to connect to the network. Software running on the host PCs views the (PCI-X) NI card as a memory-mapped peripheral.

This project focused on the queue manager, which is a module that resides between and connects the PCI-X and RocketIO interfaces. This module is responsible for network traffic storage and handling, flow isolation (VOQs), flow control (QFC credit-handling) and packet processing and segmentation (VSMPS).

3.1.2 Remote Direct Memory Access (RDMA)

An important feature of the NI developed is support for Remote Direct Memory Access (RDMA). RDMA allows a host to directly access the memory of a remote node. In this implementation RDMA operations are allowed for continuous physical address regions which have been first properly setup (i.e. pinned) by the operating system. Initiating an RDMA operation is accomplished by having software issue a RDMA request through the PCI-X interface. This triggers the construction of a RDMA packet that contains the data and among other information the remote physical target address where the data should be written.

In more detail, each RDMA packet header travelling through the network contains the following information:

- a 32-bit remote host physical destination address
- the size of the transfer in 64-bit words (maximum size is 512 words)
- the flow ID of the destination host (currently up to 128 hosts)
- an “opcode” that controls transfer details (such as completion notification)

It is important to note that the RDMA packet format used in this system allows for hassle-free segmentation of large RDMA packets into smaller autonomous packets. Essentially this means that the segmentation of a RDMA packet at the source node does not require reassembly at the destination node. This characteristic of RDMA packets is of great importance to the operation of the queue manager and specifically the packet processor module that was developed, which are described in more detail in subsection 3.4.5.

3.2 Hardware Used

The hardware used for implementing the queue manager includes FPGA (Field Programmable Gate Array)-based boards (DN6000k10SC) provided by the Dini Group (**www.dinigroup.com**). The Dini boards use FPGAs (VirtexII-Pro 2vp40) provided by Xilinx (**www.xilinx.com**). Two PowerPC processor cores designed by IBM are embedded in each FPGA. The FPGA boards were installed in the PCI-X slot of personal computers provided by Dell (**www.dell.com**) running the Linux operating system. Additional larger FPGA-based boards were configured as buffered crossbar switches to connect the Dini boards together through RocketIO links.

3.2.1 Dini Board

The board provided by Dini (DN6000k10SC) is a complete logic emulation system appropriate for logic, memory and embedded systems prototyping [14]. The system is configured around the VirtexII Pro series FPGA manufactured by Xilinx and can be used stand-alone or hosted in a 32/64-bit PCI/PCIX slot. A block diagram of the Dini board can be seen in Figure 3.2.

The Dini board includes eight serial RocketIO ports, which can support a variety of serial communication protocols at speeds up to 3.125 GB/s. The board also features four external memory chips all clocked at 133 MHz, two SRAM chips with 2-Mbits capacity each and two 512-Mbit DRAM chips. Specialized clock-generator chips (“Roboclocks”) made by Cypress provide clock input to the various components of the board.

The FPGA also offers four serial ports, various JTAG connectors, as well as a smart media connector that allows programming of the FPGA using smartmedia cards. A total of 162 signals are provided via a 200-pin connector for logic analyzer-based debugging or pattern generator stimulus.

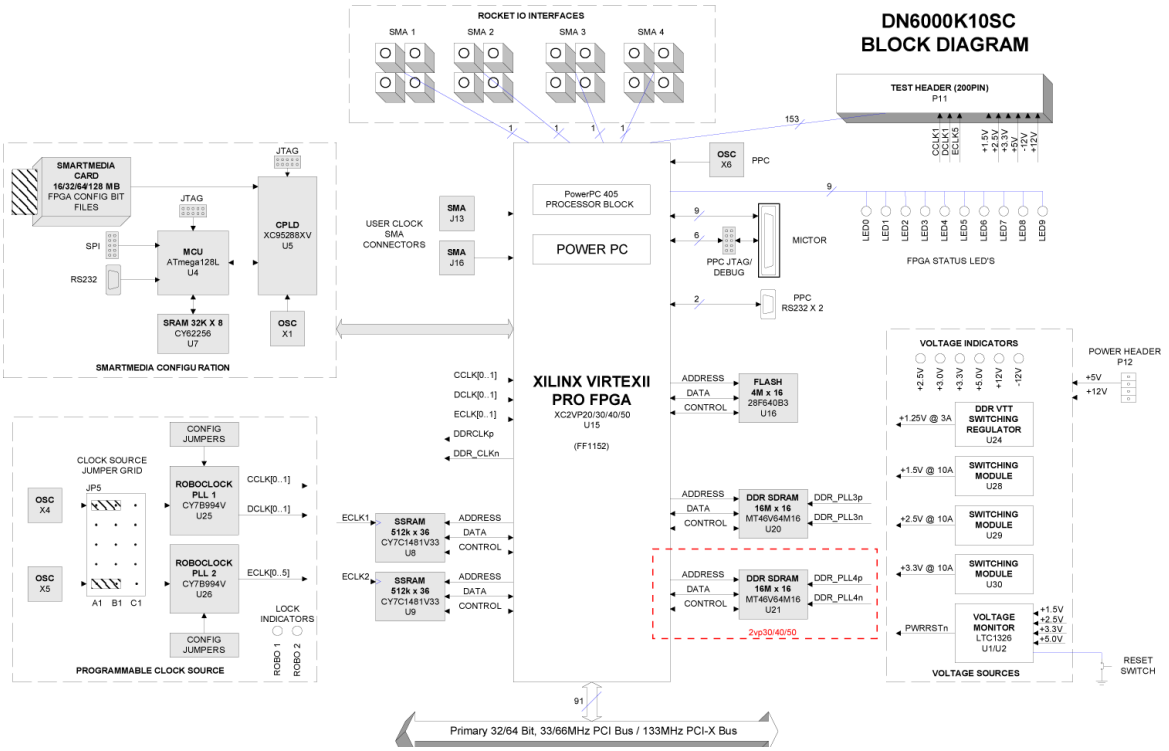

Figure 3.2: Block Diagram of the Dini board

Finally there are 10 LEDs that can be used as debugging tools or simply for visual indication of the system's status. Two push-buttons are also present on the board: one for resetting and configuring the FPGA and one that is user configurable.

3.2.2 VirtexII Pro

The FPGA of the Dini board belongs to Xilinx's VirtexII Pro family of FPGAs [15] and its specific device type is xc2vp40. It is built on a 130 nm 9-layer copper process technology and is capable of over 400 MHz operation frequencies, offering 46632 logic cells, 3456 Kbits of embedded memory (Xilinx BlockRAM or BRAM) and 192 18x18 Multipliers. It also includes 8 Digital Clock Management Blocks (DCMs) and 12 high-speed RocketIO transceivers. The heart of the VirtexII Pro FPGA accommodates two IBM PowerPC processor cores capable of operating at over 300 Mhz.

3.3 Methodology

Verilog HDL was the primary hardware description language used, although the development tools also demanded the sporadic use of the VHDL hardware description language. A Concurrent Versioning System (CVS) repository was used to keep track of all changes, versions and branches of the source code.

The developed modules comprising the queue manager are fully parametrizable and can easily be adapted to specific implementation requirements. The number and width of VOQs, maximum segment size, size of on-chip memory, as well as other parameters can be manually set.

The simulations that were carried out through the various phases of the project were done using Modelsim by Modeltech (www.model.com) and NCLaunch by Cadence (www.cadence.com).

The main software tools used for development were provided by Xilinx. The Xilinx *Project Navigator* was used to synthesize the verilog code and produce the appropriate files for programming the FPGAs, while the Xilinx *Chipscope Pro Analyzer* software was used for clock-cycle granularity verification.

3.4 Implementation

This subsection provides implementation details for each of the queue manager's modules, which were presented at the architectural level in subsection 2.3. A block diagram of the queue manager can be seen in Figure 3.3. The yellow arrows correspond to packet copying, while the black arrows correspond to exchange of control information.

All modules enclosed within the dashed rectangle belong to the queue manager. The PCI-X module on the left implements the PCI-X interface and provides packets to the queue manager. The RocketIO module, on the right, receives packets from the queue manager and transmits them towards the switching fabric, which in this platform is implemented as a buffered crossbar.

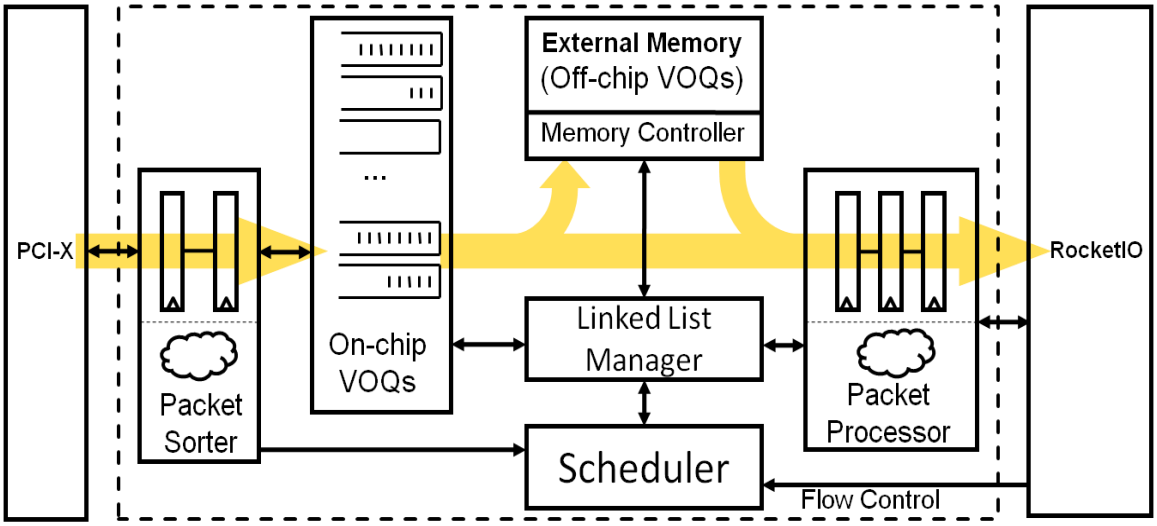

Figure 3.3: Block Diagram of the Queue manager

The queue manager, which is the top module, consists of three major versions, which were concurrently and continuously developed following varying project goals. The queue manager versions are differentiated based on the supported features and are presented below:

i. **Full version**: The full version of the queue manager module includes all available features. This includes both variable-size multi-packet segmentation and migration of VOQs to off-chip memory in hardware-manages linked lists.

ii. **No external memory**: This version does not support VOQ migration to external memory. Unfortunately the external memory found on the implementation development board offered less than half of the bandwidth of the on-chip memory (Xilinx BRAM). Enabling external memory support, and thus VOQ migration to external memory would force the queue manager to operate at less than half of its potential rate/bandwidth. This version vastly simplifies the Linked List Manager and Scheduler modules.

iii. **No VSMP segmentation**: This version does not segment traffic according to the VSMP segmentation scheme presented in subsection 2.2.3. The motivation behind the creation of this version was to lower the hardware cost and complexity of the queue manager module allowing for greater flexibility and creation of other modules on the FPGA. This version imposes many changes to the Scheduler and eliminates the Packet Processor module.

Apart from the three major versions of the queue manager as a whole, in certain cases more than one variations of individual modules were developed to satisfy specific project goals. In such cases further information is provided about each different implementation and module variations directly related to different queue manager major versions are explicitly mentioned.

The queue manager implemented at the CARV laboratory operates at the same clock rate as the PCI-X interface, which is 100 MHz and all modules employ a 64-bit-wide datapath. The card supports 8 VOQs, since the prototyping platform consists of 8 PCs, but the number of supported VOQs is configurable and can be easily changed.

The maximum DMA request size is set to 4Kbytes, while the maximum network packet size is 512 bytes. The minimum network packet size allowed is 8 bytes, i.e. its payload consists of a single 64-bit word. All packet headers occupy 16 bytes, i.e. 128 bits.

The external memory used comprises of two DDR DRAM memories operating at 133 MHz and a 16-bit-wide datapath each. External memory block size is set to 512 bytes, which is sufficiently large for achieving maximum DRAM throughput.

The modules comprising the queue manager are presented in the following order:

- Packet Sorter
- On-Chip VOQs
- Scheduler
- Linked List Manager
- Packet Processor

3.4.1 Packet Sorter

Two variations of this module were implemented. The basic version classifies packets according to their destination, enqueues them in the on-chip VOQs and notifies the scheduler. The alternate version of the packet sorter also enforces a maximum packet size by segmenting larger packets into smaller ones. Only the later version is presented in this thesis because it supersedes the former. The interface of the Packet Sorter module comprises of the signals presented in Table 3.1.

Table 3.1: Packet Sorter Interface

Signal Name	In/Out	Width	Short Description
clk	In	1	Queue Manager Clock
reset	In	1	Queue Manager Reset
fifo_din	In	68	Data bus for incoming network traffic originating from the PCI-X interface
fifo_re	Out	1	Read Enable/Dequeue signal for PCI-X fifo
fifo_empty	In	1	Indicates empty PCI-X fifo
VOQ_out	Out	3	Indicates VOQ for currently outgoing traffic
Enq	Out	1	Enqueues data into the On-Chip VOQs
fifo_dout	Out	68	Data bus for outgoing traffic headed towards the On-Chip VOQs
sram_inc	Out	1	Notifies the Scheduler when a packet is stored in the On-Chip VOQs
sram_in_size	Out	10	Indicates the size of the current packet in 64-bit words

The packet sorter receives packets from the PCI-X module through a simple FIFO-based interface. When this FIFO is not empty, new packets are constantly received. Each time a packet header is encountered it is saved until the next header replaces it. If a large packet has to be split in smaller parts, then new headers are created for each part based on the original header that was stored.

The Packet Sorter module connects to the On-Chip VOQs module through another simple FIFO-based interface. Enqueueing a packet into the On-Chip VOQs is as simple as indicating the appropriate VOQ through the VOQ_out signal and asserting the Enq signal.

While a packet passes through the packet sorter its size is monitored in order to be reported to the scheduler. Each time a whole packet is forwarded to the On-Chip VOQs the scheduler is notified about its size through the sram_inc and sram_in_size signals.

3.4.2 On-Chip VOQs

The On-Chip VOQs module employs flow isolation by storing packets into different queues (VOQs) according to their destination. It receives packets from the Packet Sorter and supplies packets to the Linked List Manager.

Incoming packets are stored in fixed-sized queues of 8Kbytes each. Once again two variations of this module were built, which differ in their queue implementation. The first variation implements all of the queues inside a common shared statically partitioned on-chip Xilinx BRAM memory. The second variation instantiates separate Xilinx FIFO modules for each queue. The interface of the On-Chip VOQs module comprises of the signals presented in Table 3.2.

Table 3.2: On-Chip VOQs Interface

Signal Name	In/Out	Width	Short Description
clk	In	1	Queue Manager Clock
reset	In	1	Queue Manager Reset
enq	In	1	Enqueues data into the On-Chip VOQs
voq_in	In	3	Specifies a VOQ for incoming packets
data_in	In	68	Data bus for incoming packets from the Packet Sorter
deq	In	1	Dequeues data from the On-Chip VOQs
voq_out	Out	3	Specifies VOQ for outgoing traffic
data_out	Out	68	Data bus for outgoing packets headed towards the Packet Processor or the Off-Chip VOQs
VOQs_dout	Out	8*68	Carries the first word of data at the head of each VOQ

The first variation of this module implements the VOQs as independent circular buffers inside a shared 64 Kbyte memory and requires head and tail pointers to be maintained for each VOQ (Figure 3.4). The head pointer of each VOQ points at the first data word that will be read for a dequeue operation, while the tail pointer points at the first free word, where data will be written for an enqueue operation. When a VOQ is empty the head and tail pointer both point at the same place (as is the case for Queue 3 in Figure 3.4).

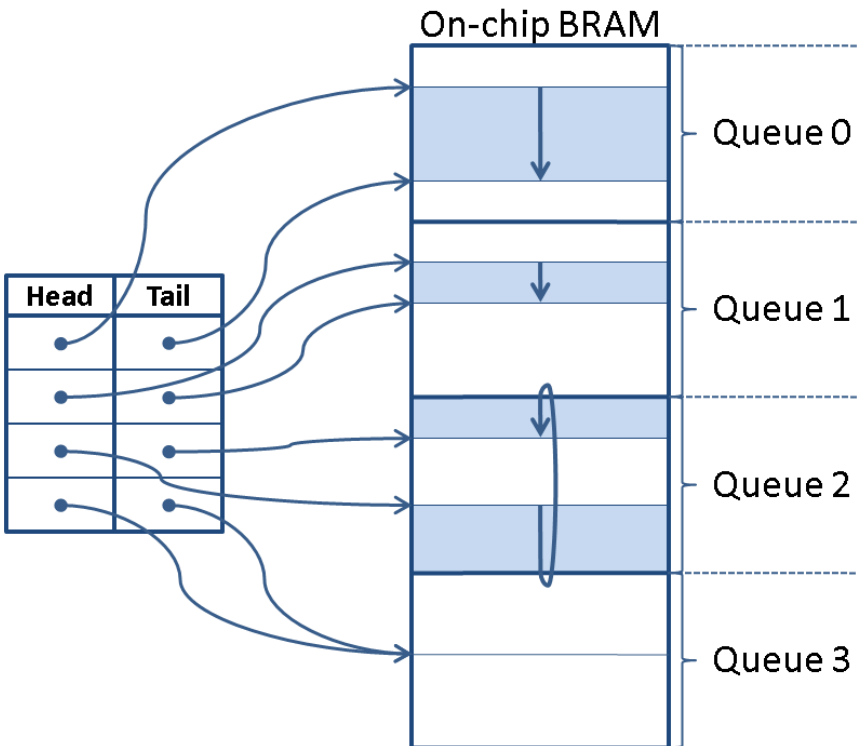

Figure 3.4: VOQs implemented as circular buffers

The second variation of the On-Chip VOQs module used Xilinx FIFO modules to implement each VOQ. Each Xilinx FIFO module is built using one or more BRAMs. This approach results in somewhat more complicated hardware, but provides more advanced features thus allowing more complex operations. For instance by enabling the First-Word Fall-Through feature it is possible to look-ahead to the next data word available from a FIFO without issuing a dequeue operation. This can be used to peek at the header of a packet and inform the Scheduler about packets waiting at the head of a VOQ. The VOQs_dout signal serves exactly this purpose and it is only present in the "No VSMP segmentation" version of the ingress card.

3.4.3 Scheduler

The Scheduler is the most important and complicated module of the queue manager. Based on information it receives from the Packet Sorter, the Linked List Manager, the Packet Processor, the switching fabric and occasionally from the On-Chip VOQs it instructs the Linked List Manager on what operation to perform next. The Scheduler also implements flow control between the queue manager and the traffic-generating host as well as between the queue manager and the switching fabric. The Scheduler was kept as independent as possible to allow easy change of policies and scheduling algorithms without affecting the rest of the modules. The interface of the Scheduler module comprises of the signals presented in Table 3.3. A block diagram of the Scheduler can be seen in Figure 3.5.

Table 3.3: Scheduler Interface

Signal Name	In/Out	Width	Short Description
clk	In	1	Queue Manager Clock
reset	In	1	Queue Manager Reset
sram_voq_in	In	3	Specifies the VOQ for currently incoming traffic from the On-chip VOQs module
sram_in_size	In	10	Specifies the size of the current incoming packet from the On-chip VOQs in 64-bit words
sram_inc	In	1	When asserted the on-chip occupancy for the specified VOQ is increased by "sram_in_size" 64-bit words
sram_voq_out	In	3	Specifies the VOQ for currently outgoing traffic originating from the On-chip VOQs module
sram_out_size	In	10	Specifies the size of the current outgoing packet from the On-chip VOQs in 64-bit words
sram_dec	In	1	When asserted the on-chip occupancy for the specified VOQ is decreased by "sram_out_size" 64-bit words
dram_voq	In	3	Specifies the currently selected off-chip VOQ
dram_inc	In	1	Increments the occupancy of the selected off-chip VOQ by one. (occupancy is measured in external memory blocks)
dram_dec	In	1	Decrements the occupancy of the selected off-chip VOQ by one. (occupancy is measured in external memory blocks)
RIO_clk	In	1	Provides the clock signal of the RocketIO interface
credit_from_RIO	In	32	Carries flow control information for up to two VOQs
sent_voq	In	3	Specifies the VOQ for traffic departing towards the switching fabric
sent_size	In	10	Specifies the size of the packet currently departing towards the switching fabric
sent_inc	In	1	When asserted the number of sent packets for the specified VOQ is increased by "sent_size" 64-bit words
xbar_voq	Out	3	Informs the Packet Processor which VOQ the current segment belongs to
seg_size	Out	10	Specifies the size of the segment currently being transferred towards the Packet Processor
sram2dram_begin	Out	1	Requests a segment transfer from on-chip to off-chip VOQs
sram2dram_VOQ	Out	3	Specifies a VOQ for on-chip to off-chip segment transfers
sram2dram_done	In	1	Indicates that a on-chip to off-chip VOQ transfer is done
dram2xbar_begin	Out	1	Requests a segment transfer from off-chip VOQs to the Packet Processor
dram2xbar_VOQ	Out	3	Specifies a VOQ for off-chip VOQs to Packet Processor segment transfers
dram2xbar_done	In	1	Indicates that an off-chip VOQs to Packet Processor transfer is done
sram2xbar_begin	Out	1	Requests a segment transfer from on-chip VOQs to the Packet Processor
sram2xbar_VOQ	Out	3	Specifies a VOQ for on-chip VOQs to Packet Processor segment transfers
sram2xbar_size	Out	10	Specifies the size of on-chip VOQs to Packet Processor segment transfers
sram2xbar_done	In	1	Indicates an on-chip VOQs to Packet Processor transfer is done
PCI_2_VOQ	Out	8	Informs the PCI-X interface about blocked VOQs
VOQs_dout	In	8x68	Carries the first word of data at the head of each VOQ

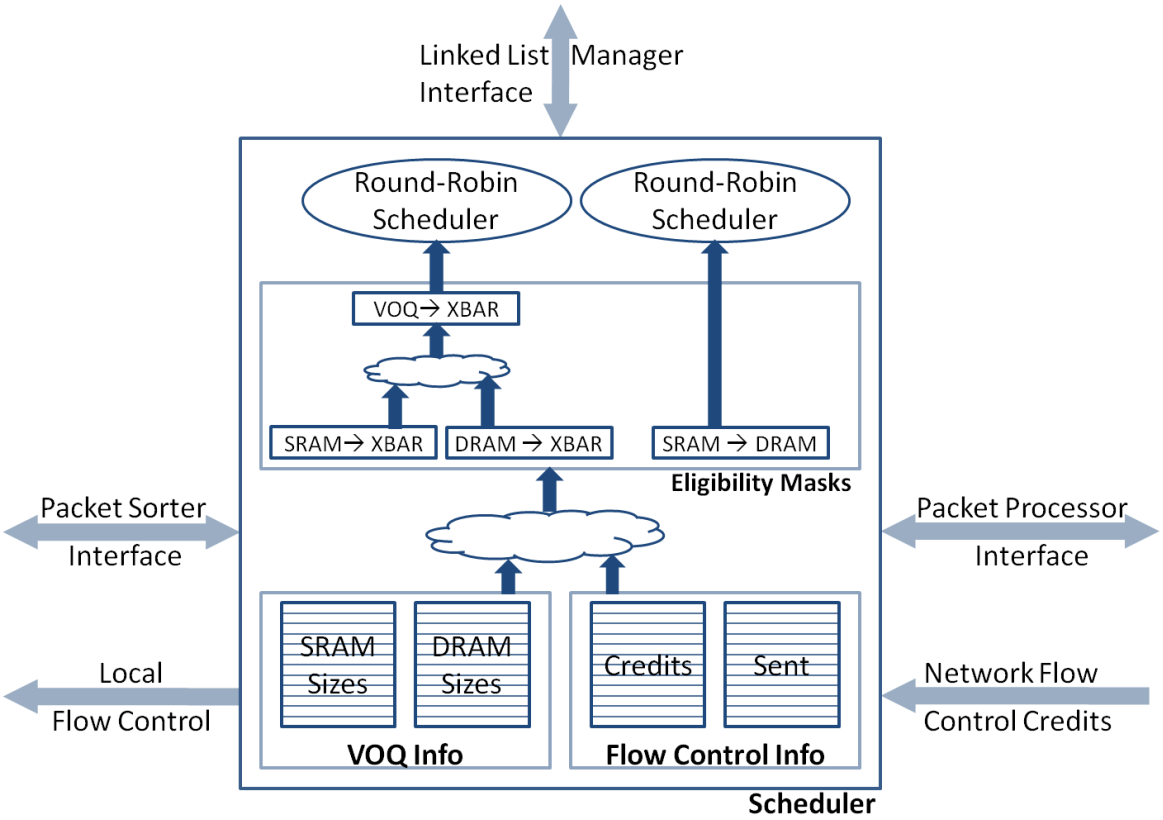

Figure 3.5: Scheduler Block Diagram

The Scheduler keeps track of the status of the on- and off-chip VOQs. This corresponds to on-chip (BRAM) and off-chip (DRAM) occupancies for each VOQ, which are measured in 64-bit words and DRAM blocks respectively. In the FPGA implementation two small register files are used to store this information.

For flow control purposes, the Scheduler keeps track of two quantities for each VOQ:

i) from the Packet Processor it receives the number of data words that have been transmitted towards the switching fabric
ii) from the downstream (next-hop) network switch (in the switching fabric) it receives the number of data words that have been forwarded and thus have left the crossbar's buffers.

Using the number of sent words in conjunction with the reported number of forwarded words, it is possible to calculate the available buffer space at the next-hop switch for each VOQ. As with occupancy information, two small register files are used to store this information.

Information regarding forwarded packets is periodically received from the switching fabric and enqueued as a 32-bit word in a special asynchronous FIFO residing in the Scheduler. Each

32-bit word may carry flow control information for up to two VOQs and contains the number of forwarded 64-bit words, specifies the VOQ and utilizes parity bits for error detection.

The scheduler decides which VOQ to service next based on the occupancy and flow control information described above. This is done using 3 eligibility masks for segment transfers:

i. from on-chip to off-chip VOQs (SRAM2DRAM)
ii. from on-chip VOQs to the network (SRAM2XBAR)
iii. from off-chip VOQs to the network (DRAM2XBAR)

Each eligibility mask dedicates one bit for each VOQ. If this bit is asserted the VOQ is eligible for this kind of segment transfer; otherwise it is not considered to be eligible. The SRAM2XBAR and the DRAM2XBAR eligibility masks are combined – using a binary OR operation - into a single eligibility mask, namely VOQ2XBAR, which represents the VOQs, on-chip or off-chip, that are allowed to sent packets towards the network. Each time a decision needs to be made, the two remaining eligibility masks, VOQ2XBAR and SRAM2DRAM, are fed into two round-robin schedulers, which output the next VOQ to be serviced.

Periodic updating of the eligibility mask is a fundamental responsibility of the Scheduler. For each and every possible event that affects a VOQ, whether this is an occupancy change, off-chip memory migration or arrival of flow control information, the corresponding eligibility mask needs to be updated. Specifically, the eligibility masks are updated for each one of the following VOQ events:

- An empty VOQ that has credits receives traffic and becomes eligible
- A VOQ that has credits runs out of packets and becomes ineligible
- A VOQ that has traffic runs out of credits and becomes ineligible
- A VOQ that has traffic receives credits and becomes eligible[1]

In the full version of the queue manager, where VSMP segmentation support is enabled, the Scheduler is allowed to transfer an arbitrary number of words, always within the segment size limits, from any VOQ ignoring packet boundaries. The only requirement is the existence of credits for sending a minimum network packet. In this case the queue manager relies on the

[1] This can happen for two VOQs that simultaneously receive credits. As mentioned previously each 32-bit flow control information packet may contain credits for up to two.

Packet Processor to transform the transmitted arbitrary piece of traffic into autonomous network packets. This is explained in more detail in subsection 3.4.5.

In contrast, in the "No VSMP segmentation" version of the queue manager, the Scheduler needs to constantly be aware of the size of the first packet sitting at the head of each VOQ and is only allowed to initiate a transfer when sufficient credits exist for this packet. This is the reason for the VOQs_dout signal, which is only present in this "no VSMP segmentation" version of the queue manager.

In the "no external memory" version of the queue manager the Scheduler is much simpler. The register file that stores off-chip VOQ occupancies is completely omitted, while all VOQs always appear as ineligible in the SRAM2DRAM and DRAM2XBAR eligibility masks, thus simplifying scheduling decisions. Likewise, in the "no VSMP segmentation" version of the queue manager the Scheduler is accordingly altered.

In order to notify the Linked List Manager about pending transfers the Scheduler uses 3 signals, namely sram2dram_begin, sram2xbar_begin and dram2xbar_begin. More than one of these signals can be asserted simultaneously; it is up to the Linked List Manager to choose in which order to service the requests. When the transfers involve external memory (DRAM) transfer size is implicitly assumed to be equal to the External Memory Block Size, while for transfers involving on-chip memory transfer size needs to be specified using the sram2xbar_size signal. When the Linked List Manager is finished with a segment transfer it asserts the sram2dram_done, sram2xbar_done or dram2xbar_done accordingly. This triggers the round-robin scheduler in the Scheduler to specify the next segment transfer to be carried out.

3.4.4 Linked List Manager

As its name suggests, the Linked List Manager's main task is to manages the linked lists of external memory blocks that hold VOQ traffic, which has migrated to external memory due to congestion. This module also connects to the off-chip memory controller through a simple FIFO-based interface and is also responsible for performing segment transfers, as instructed by the Scheduler. The interface of the Linked List Manager module comprises of the signals presented in Table 3.4.

Table 3.4: Linked List Manager Interface

Signal Name	In/Out	Width	Short Description
clk	In	1	Queue Manager Clock
reset	In	1	Queue Manager Reset
sramDeq	Out	1	Dequeues data from the On-Chip VOQs

dram2xbarData	In	64	Data bus that connects to the DRAM memory controller
dram2xbarEnq	In	1	When asserted valid data is available from the DRAM memory
dramAddr	Out	27	Provides the address to the DRAM memory controller
dramXSize	Out	15	Specifies the DRAM transfer size in bytes
dramXOp	Out	2	Specifies the DRAM access type (read or write)
dramXDone	In	1	Signals the completion of a DRAM transfer
sram2dram_begin	Out	1	Requests a segment transfer from on-chip to off-chip VOQs
sram2dram_VOQ	Out	3	Specifies a VOQ for on-chip to off-chip segment transfers
sram2dram_done	In	1	Indicates that a on-chip to off-chip VOQ transfer is done
dram2xbar_begin	Out	1	Requests a segment transfer from off-chip VOQs to the Packet Processor
dram2xbar_VOQ	Out	3	Specifies a VOQ for off-chip VOQs to Packet Processor segment transfers
dram2xbar_done	In	1	Indicates that an off-chip VOQs to Packet Processor transfer is done
sram2xbar_begin	Out	1	Requests a segment transfer from on-chip VOQs to the Packet Processor
sram2xbar_VOQ	Out	3	Specifies a VOQ for on-chip VOQs to Packet Processor segment transfers
sram2xbar_size	Out	10	Specifies the size for on-chip VOQs to Packet Processor segment transfers
sram2xbar_done	In	1	Indicates that an on-chip VOQs to Packet Processor transfer is done
sram_voq_out	Out	3	Indicates VOQ for transfers concerning On-Chip VOQs
sram_out_size	Out	10	Indicates about the size for transfers concering On-chip VOQs
sram_dec	Out	1	Instructs the Scheduler to update On-chip VOQs occupancy
NxtPtr_mux	Out	1	Used to capture the Next Block Pointers stored in the first 64-bit word of each DRAM block.
wr_count	In	10	Monitors the RocketIO FIFO occupancy
xbarEnq	In	1	Enqueues data words to the Packet Processor

The heart of this module is a finite state machine (FSM) that coordinates segment transfers and can be seen in Figure 3.6. Linked list managements is optimized using the Free Block Preallocation [9] and Free List Bypass [10] techniques, introduced in subsection 2.3.5. This module also monitors the occupancy of the RocketIO FIFO, that accepts departing packets, and can delay a transfer until adequate space is available.

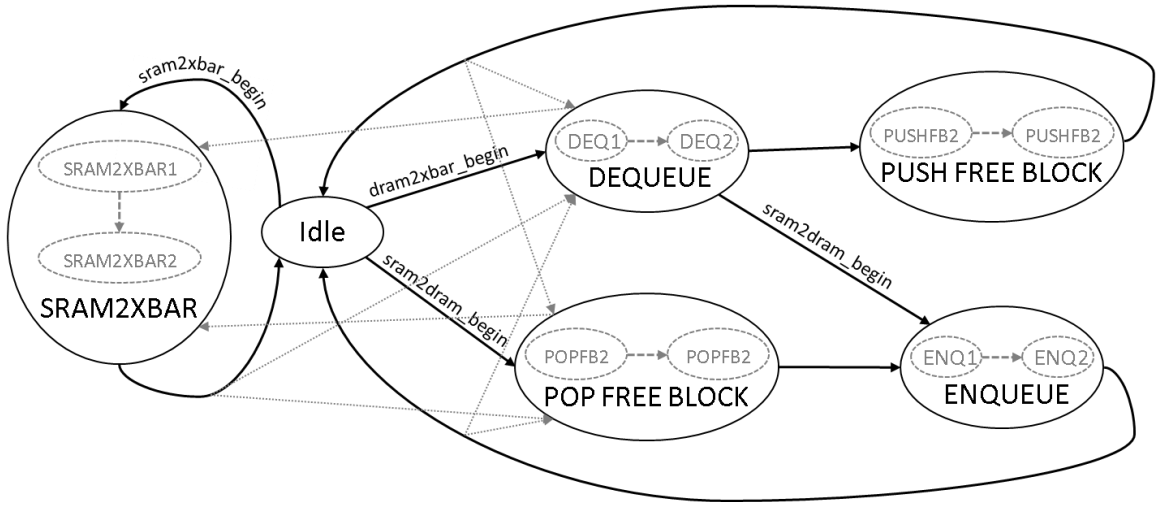

Figure 3.6: Linked List Manager FSM

The FSM consists of 11 states implemented using one-hot encoding. To simplify things, in the depicted FSM, states that belong to the same operation were grouped into six bigger states, namely IDLE, SRAM2XBAR, DEQUEUE, PUSH FREE BLOCK, POP FREE BLOCK and ENQUEUE. The state transitions for each kind of segment transfer are listed below:

- **SRAM2XBAR**: IDLE→SRAM2XBAR → IDLE
- **DRAM2XBAR**: IDLE →DEQUEUE → PUSH FREE BLOCK → IDLE
- **SRAM2DRAM**: IDLE → POP FREE BLOCK → ENQUEUE → IDLE

In order to benefit from the Free List Bypass optimization both a SRAM2DRAM transfer and a DRAM2XBAR transfer need to be pending. In this case an additional state transition path becomes valid: IDLE → DEQUEUE →ENQUEUE → IDLE. The DRAM block freed by the DEQUEUE operation can be directly used for the ENQUEUE operation, completely avoiding costly Free List operations.

In DRAM, traffic is stored in 512-byte fixed-sized blocks. Block sizes were chosen to maximize DRAM throughput. The first 64-bit word of each block contains the Next Block pointer and the remaining (31or 63) 64-bit words store actual segment data.

Since Free Block Preallocation is employed an extra block is allocated for each VOQ. This means that even an empty VOQ has a spare free block. For each enqueue operation, instead of first finding a new block and then writing the data, the already allocated free block is directly used, while a new free block is preallocated for the next enqueue operation.

3.4.5 Packet Processor

The Packet Processor module is essential for supporting VSMP segmentation. This module receives variable-size multipacket segments from the on-chip and off-chip VOQs and transforms them into fully autonomous packets that do not require segmentation at the receiving end. To achieve this, the Packet Processor needs to remember the last packet header for each VOQ. These headers are stored inside a small register file. Additionally the Packet Sorter also informs the Scheduler about the actual number of words sent towards the switching fabric for each VOQ. The interface of the Packet Processor module comprises of the signals presented in Table 3.5.

Table 3.5: Packet Processor Interface

Signal Name	In/Out	Width	Short Description
clk	In	1	Queue Manager Clock
reset	In	1	Queue Manager Reset
data_in	In	64	Data bus for incoming segments
enq	In	1	Indicates the presence of valid data at the input (data_in)

VOQ	In	3	Indicates the VOQ for the current incoming segment
seg_size	In	1	Indicates the specified segment size
fifo_dout	Out	68	Data bus for outgoing packets towards the RocketIO interface
fifo_we	Out	1	Enqueue data into the RocketIO FIFO
sent_size	Out	10	Indicates the outgoing packet size (after header modifications)
sent_voq	In	1	Indicates the VOQ for the current outgoing packet
sent_inc	In	1	Instructs the Scheduler to update the "Sent" information

Each segment received by the packet processor contains an arbitrary number of continuous 64-bit words that belong to one or more packets. In most cases packet boundaries are note aligned with segment boundaries. Each segment can be of any size within segment size limits and may contain an arbitrary number of whole packets and at most 2 fragmented packets. The Packet Processor is capable of performing three operations on incoming segments as seen in the example of Figure 3.7:

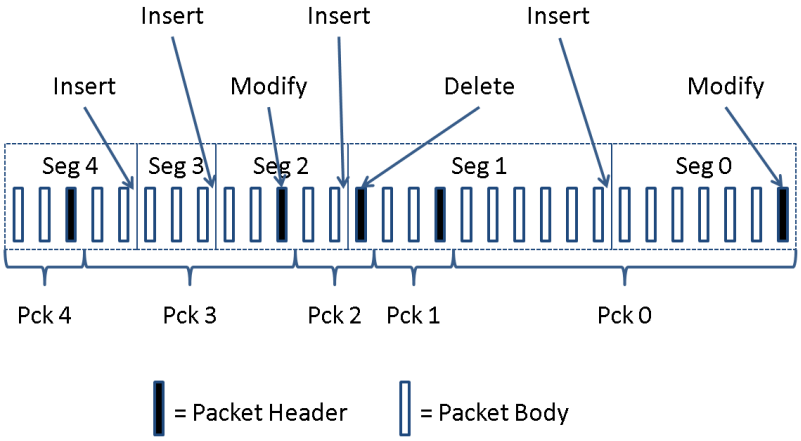

Figure 3.7: Packet Processor Operations

- **Modify**: An existing header of a packet needs to be modified if and only if the remaining packet is longer than the remaining segment. When modifying a header the number of 64-bit words to be transmitted remains the same. At most one header will be modified per segment. In Figure 3.7 this is the case for packet 0 and packet 3.

- **Insert**: If the first 64-bit word of an incoming segment does not contain a header a new header needs to be created and inserted. At most one header is inserted in any given segment. Thus when only inserting a header in a segment the number of transmitted 64-bit words is increased by exactly one word. In Figure 3.7 this happens for segments 1, 2,3 and 4.

- **Delete**: A header is only deleted if it is located at the end of the segment, i.e. in the last 64-bit word. This special case exists to avoid the injection of zero-sized packets into the network. Otherwise, under normal circumstances, and following the "modify" operation described above, the Packet Processor would modify the header instead of deleting it (creating a zero-

sized packet). When only deleting a header of a segment the transmitted number of words is decreased by one word. However, when inserting and deleting a header from a segment, the number of transmitted words remains the same. In Figure 3.7 this is the case for segment 1.

3.5 Results

This subsection includes hardware implementation cost and network performance results for the implemented queue manager. Hardware cost results include FPGA post placement & routing results for each individual module, as well as for each major version of the queue manager as a whole. Network performance results were obtained using a special modified version of the NI to conduct delay and throughput experiments[1] for the buffered crossbar.

3.5.1 Hardware Implementation Cost

Table 3.6 presents the hardware cost of each individual module of the queue manager. Separate results are presented for modules with more than one variations. The results presented concern the Xilinx VirtexII Pro xc2vp40 FPGA and were obtained using back-end tools provided by Xilinx.

The results for the On-Chip VOQs are quite interesting. They indicate that using the feature-rich FIFOs provided by Xilinx imposes a very large hardware cost. If these additional features are not required by a design it seems more attractive to implement the FIFOs from scratch, using simple circular buffers in a single statically partitioned memory.

Table 3.6: Hardware Cost Results of Individual Modules

Module	LUTs	Slices	Flip Flops	BRAMs	Gate Count
Packet Sorter $_{\text{with segmentation}}$	179 (1%)	135 (1%)	80 (1%)	0 (0%)	1909
Packet Sorter $_{\text{no segmentation}}$	42 (1%)	25 (1%)	12 (1%)	0 (0%)	392
On-Chip VOQs $_{\text{in BRAM}}$	320 (1%)	236 (1%)	170 (1%)	31 (16%)	2035342
On-Chip VOQs $_{\text{in Xilinx FIFOs}}$	904 (2%)	1408 (7%)	1648 (4%)	32 (16%)	2119015
Scheduler	2240 (5%)	1256 (6%)	428 (1%)	1 (1%)	86226
Linked List Mgr $_{\text{with ext mem}}$	680 (1%)	365 (1%)	425 (1%)	0 (0%)	7069
Linked List Mgr $_{\text{no ext mem}}$	33 (1%)	23 (1%)	18 (1%)	0 (0%)	426
Packet Processor	521 (1%)	511 (2%)	617 (1%)	0 (0%)	9983

[1] The experiments were conducted by Vassilis Papaefstathiou, member of the CARV team.

Table 3.7 presents the hardware cost results for the three major versions of the queue manager module. It is interesting to note that the "No VSMPS" version of the ingress card is more costly than the both the "Full" and the "No External Memory" version that both include VSMPS. The explanation has to do with the On-Chip VOQs module. As mentioned in subsection 3.4.2 the "No VSMPS" version of the ingress card requires the existence of the Xilinx FIFOs, because it uses some of the advanced features they provide. Not only does Hence, for this FPGA implementation , not only does VSMPS make the buffered crossbar work more efficiently, but it also lowers the hardware cost.

Table 3.7: Hardware Cost of each Queue Manager Version

Ingress Card Version	LUTs	Slices	Flip Flops	BRAMs	Gate Count
Full	2962 (7%)	2106 (10%)	1571 (3%)	34 (17%)	2263151
No External Memory	2713 (6%)	1900 (9%)	1467 (3%)	34 (17%)	2260321
No VSMPS	3430 (8%)	2639 (13%)	2286 (5%)	37 (19%)	2471047

3.5.2 Network Performance Results

Obtaining network performance results required modifications to various modules in the prototyping platform, which mostly affected the PCI-X module and the buffered crossbar switch. To obtain accurate results cycle-precise timestamps were recorded and appended in the payload of the packets at various points in the system. Using special software to generate traffic patterns, delay and throughput experiments were conducted in order to validate the simulation results for the buffered crossbar [16].

For each packet traversing the network, timestamps were recorded at the following points:

(i) upon packet creation, when the host processor writes a transfer descriptor
(ii) upon packet departure from the NI to the network
(iii) upon packet arrival at the switch port
(iv) upon departure of the packet from the switch

Timestamps (i) and (ii) measure the queuing delay and the pipeline latency in the NI, which includes delays imposed by the queue manager. Timestamps (iii) and (iv) measure the delay and latency in the switch.

Average Delay vs. Input Load under uniform traffic network performance results are presented in Figure 3.8, while throughput measurements using unbalanced traffic patterns can be seen in Figure 3.9. The observed curve closely follows the simulation results of [16].

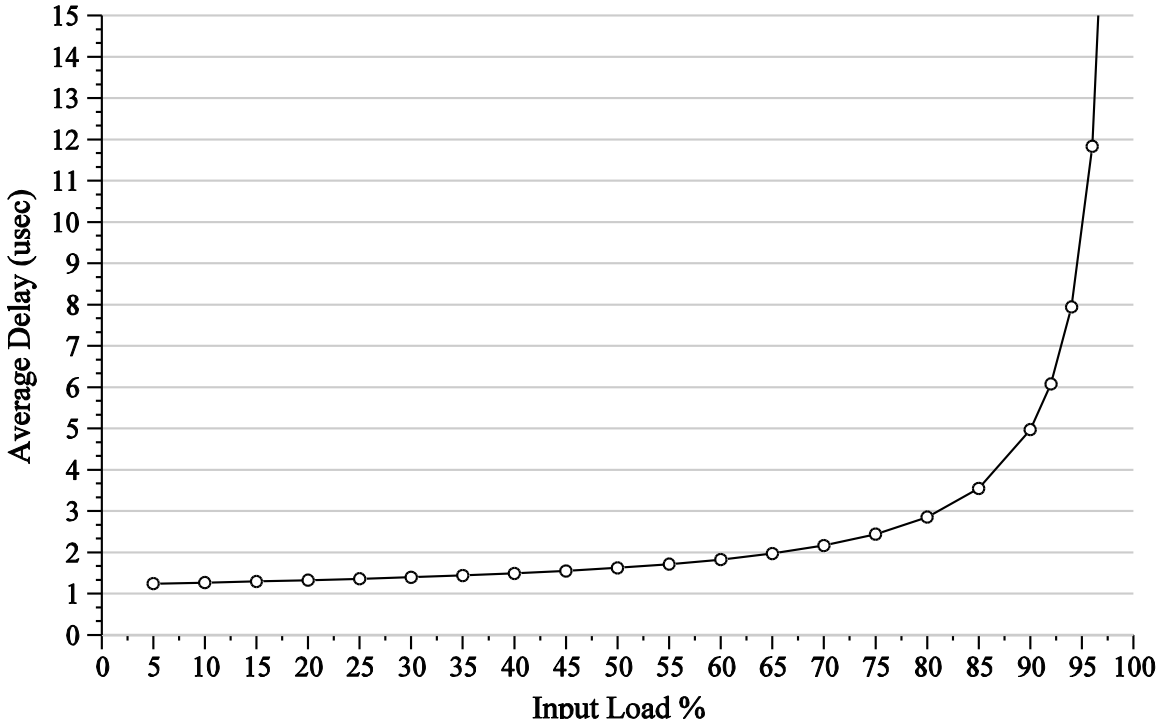

Figure 3.8: Average Delay vs. Input Load under uniform traffic. Max load is 96%.

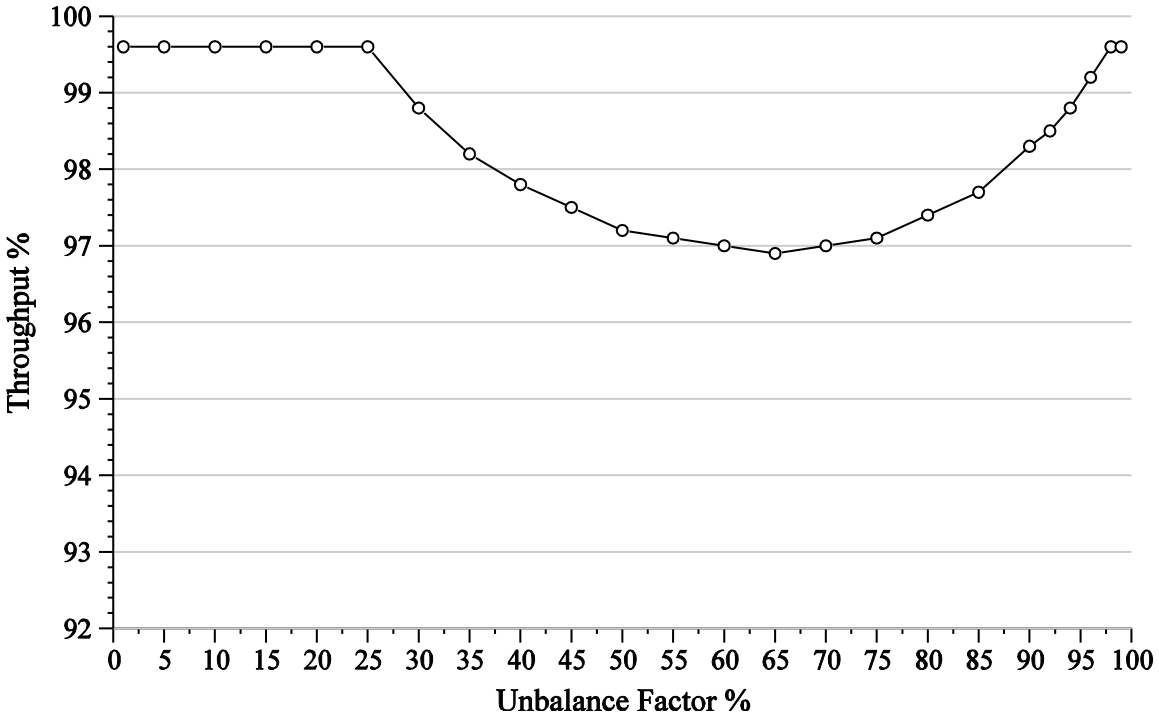

Figure 3.9: Throughput measurements using unbalanced traffic patterns.

4 Interprocessor Communication: NI Design Issues

Interprocessor Communication (IPC) refers to the passing of data among the processors of a parallel computer during the execution of a parallel program. This section focuses on several fundamental NI design issues that affect IPC and presents resolution approaches used in commercial and academic systems. The information provided in this section also serves as background for understanding and evaluating the proposed NI design presented in section 1.

4.1 Fundamentals of IPC

IPC can be implemented through Shared Memory or Message Passing. In the first case, a shared memory is assumed and communication happens implicitly by processors writing into and reading from this shared address space. To the contrary, in Message Passing each processing node has its own private local memory and communication takes place in an explicit manner using send and receive operations. In addition to the extreme cases of Shared Memory and Message Passing there are possibilities for hybrid designs that combine features of both.

The traditional implementation for Shared Memory systems is using a shared bus to connect all of the processors to the memory. To improve performance and reduce bus contention each node usually has a local cache. In order to preserve a coherent view of memory by all nodes some cache coherency scheme needs to be employed, such as MESI or some other snooping cache coherence protocol [17].

In an effort to improve the scalability of shared memory systems, distributed shared memory systems were introduced where each processing node owns and is responsible for a small piece of the entire system memory. In this case local caching of data is usually carried out through some kind of a directory-based coherence protocol [18]. Shared Memory offers great performance for small-scale dense systems, but it suffers from scalability issues, which is the main reason that large-scale parallel computers do not support it.

Message Passing systems comprise of processing nodes, each with its own private memory, that are connected through an interconnection network. Communication among computing nodes is explicit, through initiation of send and receive operations. Although highly scalable, and therefore well-suited to large-scale parallel machines, Message Passing systems are harder to efficiently program and debug, especially when fine-grain communication is required/involved.

Shared Memory and Message Passing systems come with their respective programming models, namely the Shared Memory and the Message Passing programming models. Shared

memory parallel machines usually offer support for both programming models, since it is relatively easy to efficiently implement the message passing programming model on a shared memory machine. To the contrary, it is very hard and inefficient to implement shared memory on message passing machines.

As parallel computers shifted away from bus-based systems due to scalability issues, engineers realized that NI design plays a vital role in achieving high performance, affecting both distributed shared memory and message passing machines. Technology and application trends indicate that communication latency will become a critical bottleneck. In order to accommodate processor performance increases and parallel software complexity it is imperative that modern NIs offer efficient, reliable, high performance communication.

4.2 NI Design Goals

The design of NIs for parallel computers requires consideration of numerous design parameters. However there seems to be a set of fundamental NI design goals, which are valid regardless of the specific parallel machine requirements. To build a powerful parallel machine it is imperative that the NI offers high performance, scalability, reliability, protection and minimizes overheads.

4.2.1 High Performance

To deliver high performance an NI needs to offer low latency and high bandwidth. Latency reduction mostly affects short frequent transfers and is crucial for implementing request-grant protocols, offering efficient fine-grain communication and communicating control traffic.

Latency has three components [19]:

1. the software protocol latency

2. the latency through the operating system, and

3. the hardware latency to access the interconnection network.

Careful NI design can greatly improve on all three components of latency.

The second aspect of performance is high bandwidth, which mostly affects bulky transfers, when data or entire memory regions migrate from one node to another, or when pieces of data need to be broadcasted to many nodes in the system. In most cases, it is the interconnection

network that sets the hardware limits for bandwidth. However the amount of the available bandwidth that is utilized is greatly influenced by NI design.

4.2.2 Scalability

The number of nodes in a parallel computer is rapidly growing, making scalability an increasingly important consideration in NI design. Even today's parallel machines employ tens of thousands of processing nodes, e.g. [20], and advances in silicon technology indicate that this trend will continue at least for the next few years. In simple terms scalability means maintaining high performance and good speedup values when building a larger parallel machine.

4.2.3 Reliability

In addition to performance and scalability, reliability is another very important NI design goal for two reasons. The first reason is quite obvious. A parallel machine with a reliable NI can spend less time dealing with communication errors and more time doing useful work, which directly translates to greater utilization of the available bandwidth. The second reason is more subtle; the presence of an unreliable NI requires complex protocol software to accommodate for the frequent errors. This ultimately leads to higher network message latencies, a problem that is greatly manifested in the TCP/IP protocol stack [21].

4.2.4 Protection

Offering elementary protection can be a fairly easy task. The challenge is to combine protection with high performance. The most straight-forward form of protection is across different processes running on the same node and sharing a common NI. In NI design protection is usually tightly related to address translation issues, i.e. a process is only aware of virtual addresses, while the NI can only access the memory through physical addresses. On another note, it is also important for an NI to offer mechanisms for efficient protection between the kernel and the user-level processes, as well as among different nodes.

4.2.5 Overhead minimization

Overhead minimization suggests that the NI should reduce the cost of communication as much as possible. Each time a network message has to be sent or received the processor should be "distracted" as little as possible. There are many techniques to minimize communication overhead, such as the use of zero-copy or minimal-copy protocols and allowing the NI to perform DMA operations to and from main memory. Providing protection and minimizing communication

overhead are in some cases two conflicting goals and it is quite important not to sacrifice the one for the other; they both are desired goals.

4.3 NI Placement

NI placement greatly affects interprocessor communication performance, i.e. latency and bandwidth. The main way of classifying different NI placements is by the proximity of the NI to the processor. Placing the NI closer to the processor grants lower latency and higher bandwidth. On the other hand, as one moves away from the processor, more buffer space is available and interfaces become more standardized and thus easier to develop for. NIs mapped to processor registers and located on the I/O bus provide the two extremes of NI placement. Figure 4.1 shows 4 candidate locations for placing the NI. The numbers indicate proximity to the CPU, e.g. location 1 is the most distant from the processor, while location 4 is closest to the processor.

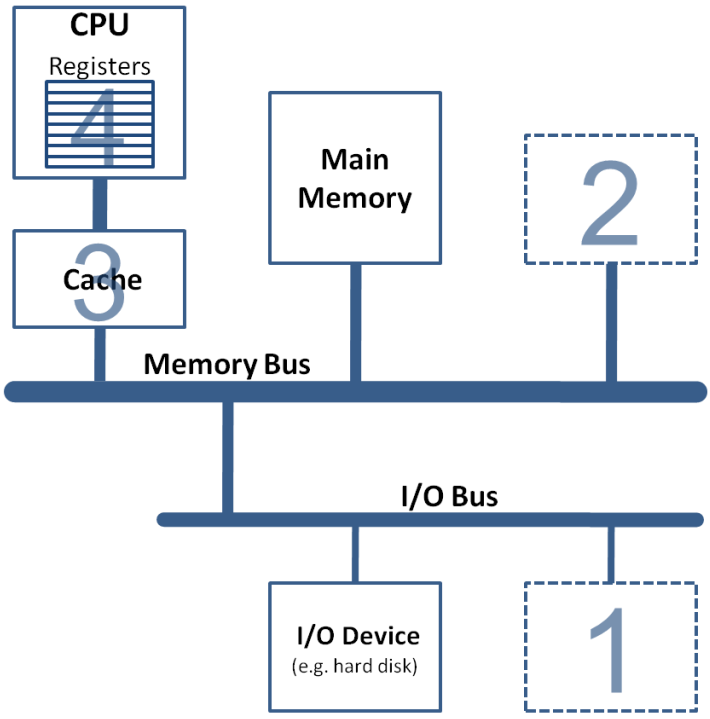

Figure 4.1: Alternative NI Placements

Traditionally NIs have been placed on the I/O bus along with other peripheral devices, such as hard disks. Even today, most commercial NIs made by third-party vendors sit on the I/O bus, because it offers a standard interface. However performance-wise this is the worst possible NI placement choice, for both bandwidth and latency. Today's 64-bit PCI-X buses running at 100 MHz can offer a maximum theoretical bandwidth of 800 Mbytes/sec, which is at least an order of magnitude less than what current memory buses can offer. In order to truly offer high-

performance the NI needs to move away from the I/O bus, closer to the processor. NI Placement on the I/O bus corresponds to location 1 in Figure 4.1.

Memory buses offer both very high bandwidth and relatively low access latencies, which makes them great candidates for NI placement. Unfortunately memory buses are, with very few exceptions [22], proprietary, so only companies that manufacture processors are able to design such NIs, as, for example, Intel's Teraflop supercomputer, where the NI sat directly on the PentiumPro memory bus [23]. To allow the production of high-performance third-party NIs a few processor vendors have created standard interfaces that connect to the memory bus and offer greater performance than the I/O bus. Examples include the SGI Power Challenge [24] and the Intel 82547EI Gigabit Ethernet Controller [25]. NI Placement on the memory bus corresponds to location 2 in Figure 4.1.

As NIs move closer to the processor, they become even more proprietary. Placing the NI in the processor cache is only a choice for processor manufacturers, but can yield excellent performance results. Latency is very low, in the order of tens of cycles, while enormous bandwidth is available. An additional advantage of integrating the NI in the cache is the opportunity for resource sharing, which reduces overheads. In its simplest form this means that the processor and the NI can share some parts of memory. The idea of resource sharing can be taken even further, as, for example, in making the memory hierarchy TLB accessible to the NI for address translation. Moreover, cache eviction mechanisms can be used to implement copy-on-write protocols. Examples of such NIs include the MIT Alewife machine [26] and the MIT *T-NG [27]. NI placement near the cache corresponds to location 3 in Figure 4.1 and seems to be a strong candidate for future Chip Multiprocessors with hundreds or thousands of nodes on the same die.

A few research projects, such as the MIT J-machine [28] and the MIT M-machine [29] have attempted to map the NI to processor registers. Such designs are very intrusive and may thus require complete redesign of a significant part of the processor itself. On top of that, NIs usually require large amounts of buffer space which cannot easily be offered with this placement. Nevertheless NIs mapped to the processor registers offer the lowest possible access latency which is in the range of one to two processor clock cycles, i.e. a few nanoseconds. NI placement near the processor registers corresponds to location 4 in Figure 4.1.

4.4 NI Virtualization & Protection: User-Level NIs

To share an NI among many processes traditional implementations require that user-level software executes a system call each time the NI is accessed to send or receive a message. This virtualization approach, which is also the way Unix sockets work, imposes tremendous overheads to the entire parallel machine, with the most apparent disadvantage being enormous latencies to initiate a transfer. Transitions from user mode to kernel mode and back cost in the order of hundreds or thousands of instructions [30] and this number is expected to rise as processor performance increases.

To alleviate this problem, modern NIs allow direct user-level access to the NI. This is achieved by mapping the NI resources to the user-level virtual address space, in a way similar to virtual memory being allocated to processes. The operating system can still be used to manage NI resources, but sending and receiving network messages, which are the usual tasks, do not have to suffer from OS intervention overheads. Such network interfaces are called User-Level Network Interfaces (ULNIs), which are quite common in high-performance parallel computers. Examples include Myricom Myrinet [31] and U-Net [32]. Figure 4.2 shows the two ways of accessing the NI.

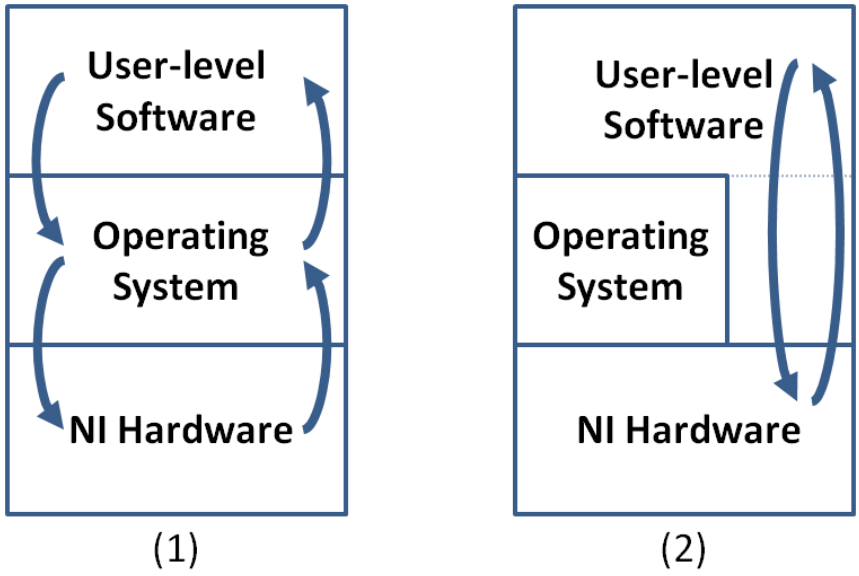

Figure 4.2: Accesing the NI. (1) Traditional NIs access the NI through the Operating System.(2) ULNIs bypass the Operating System when accessing the NI.

When using ULNIs, special care needs to be taken to guarantee protection among different processes that are accessing the NI. A trivial approach to implementing protection is to only allow the existence of a single process per node, i.e. completely disallow multiprogramming. A similar approach is to allow the existence of many processes per node – i.e. allow multiprogramming -

but allocate the NI only to a single process. Again, if only one process accesses the NI there are obviously no protection issues, as was done in TMC CM-5 [33]. These approaches, however, do not really solve the problem, but rather turn a blind eye to it.

To allow both multiprogramming and simultaneous user-level access to the NI by multiple processes on the same node, a different protection scheme needs to be used. NI resources (e.g. buffer space) need to be divided and separately mapped through different virtual address ranges to the various processes that desire to access the NI. Protection is implicitly present (in a "natural" way) through the OS memory virtualization mechanisms, i.e. the paging system. An advantage of this approach is that it is possible to dynamically allocate the exact amount of NI resources for each process. For instance a process that constantly sends messages can be allocated a lot of outgoing buffer space. A slight disadvantage of this approach is that resource division and protection are done at memory page granularity. If NI resources are scarce, it is likely that some form of finer grain protection is desirable. The Mondrian Memory Protection scheme [34] is a solution for fine-grained protection

4.5 NI Data Transfer Mechanisms

In addition to bypassing the operating system through the use of ULNIs the overheads associated to sending and receiving a message are also greatly affected by the NI Data Transfer Mechanisms offered by a NI. NI Data Transfer mechanisms describe the available ways to transfer data to/from the NI and belong to two major categories:

4.5.1 Using Store/Load instructions

In the first category, the processor sends/receives messages by conducting consecutive stores/loads to/from the memory region that is mapped to the NI. In traditional systems, where the NI is "just another" I/O device, the loads and stores performed need to be uncached, to ensure that they bypass the processor cache and reach the NI. Unfortunately, this approach is not a viable solution for building high-performance parallel machines, because uncached loads and stores offer very low bandwidth.

An alternative to transfer data to and from the NI through the use of cached stores and loads. This approach, however, requires the employment of some kind of coherence protocol that the NI participates in. If no cache coherency is present there is no guarantee that the NI will be notified about the processors actions. To receive arriving messages the processor issues loads, which lead to cache misses and are served by the NI. Sending messages is very similar and happens through consecutive stores, while cache coherency mechanisms make sure that the data will get delivered

to the NI. The main advantage of this approach is that data are transferred in cache blocks, thus offering more bandwidth.

An even more aggressive approach is to allow the NI to place data directly into the processor cache. In fact, Intel has proposed such a mechanism, called DCA [35]. Although it completely eliminates the cache miss penalty for receiving new messages, this approach may lead to the processor cache being polluted. Firstly, the processor might not be ready to process the just received data that have possibly evicted useful data from the cache. Secondly, and even worse, the received data might not be designated to the processor, but to some other device (i.e. the hard disk).

4.5.2 Using Direct Memory Access (DMA)

The second major category consists of DMA-based data transfer mechanisms. When using DMA to send a message the processor informs the NI about the memory location of the data to be sent. From there on, it is the NI's responsibility to fetch the data and construct the message. The advantage of this approach is that the processor does not spend useful cycles to copy data; instead the processor is decoupled and can perform useful computations, while the DMA engine performs the copy.

User-level software only has knowledge of virtual addresses, while DMA engines can only operate on physical addresses. In traditional DMA-based implementations when sending messages, a costly system call is required to get the virtual to physical address translation for the data to be sent. This adds a considerable overhead to sending messages and makes DMA only attractive for bulky data transfers (to compensate for the associated overheads).

To completely avoid the operating system when sending a message, special address translation techniques need to be employed. Either the NI needs to have access to the system's translation table or the user-level software needs to somehow supply the physical address location of the data it wishes to send, as was done in the SHRIMP NI [36].

4.6 Address Translation

As explained earlier, if a ULNI is to support DMA as one of its data transfer mechanisms, an address translation mechanism is required. User-level software is only aware of virtual address, while the NI requires the physical location of data to initiate a transfer. Modern ULNIs usually support one of two address translation schemes.

In the first scheme, only a few ranges of the virtual address space, for which the virtual to physical translation is known, can be used by user-level software for communication. This constrain forces user-level processes to place their communication data structures in these regions and may also lead to frequent copying of data into and out of this restricted virtual address memory region. Moreover these physical memory pages need to be pinned, to avoid being swapped out to the hard disk. Each time a new virtual address space region is to be used for communication, a system call has to be performed to pin the physical pages and inform the NI about the corresponding physical pages. When a process requires large buffers for communication-intensive tasks it is usually best to find an address region that consists of contiguous physical pages, which can sometimes be hard to find.

In the second address translation scheme, the whole virtual address space of each process can be used to communicate data to the NI. Obviously, this requires that the NI has access to each and every entry of the operating system translation table. The advantage of this approach is that user-level software is able to initiate a transfer, regardless of the location of the data to be transferred. One awkward way of letting the NI know all of the virtual to physical memory translations is to place the whole translation table inside the NI, as was done in the T3E Multiprocessor [37].

In other solutions, the most used translation table entries are cached in the NI. In this case, care should be taken to update this cache when pages are remapped or swapped to disk. Again pages that are used by the NI need to be pinned. The NI translation cache can be filled either by interrupting the operating system on each miss, which can be quite costly, or by using shadow address spaces to fill the cache in user-level mode without any operating system intervention. This shadow address space technique was used in the SHRIMP NI [36] and the Stanford Flash Multiprocessor [38].

An example of the SHRIMP NI approach can be seen in

Figure 4.3. Both virtual and physical address spaces are divided into two regions—a regular space and a shadow space. For each address space there exists one-to-one mappings from the regular space to the shadow space. To initiate a DMA to a destination virtual address 0XYZ, the user process does a store to the corresponding shadow virtual address 1XYZ. The shadow mapping is obtained through a simple invertible function such as flipping a bit. The virtual memory hardware translates the shadow virtual address to the shadow physical address 1ABC, which the UDMA device observes on the bus. Finally, the NI converts the shadow physical address back to the regular physical address by flipping the first bit and interprets the store to 1ABC as the user-level process' intention to initiate DMA to the destination physical address 0ABC. Thus, a user-level process delivers authentic physical addresses (in this case the

translation from 0XYZ to 0ABC) using commodity virtual memory hardware to the UDMA device without invoking the operating system.

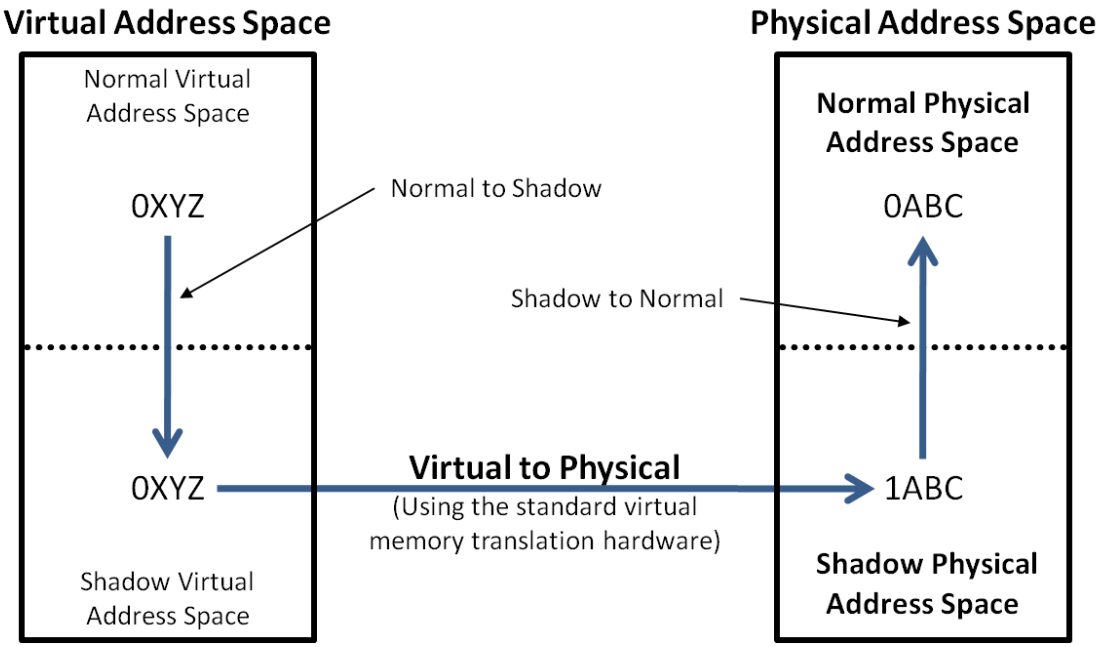

Figure 4.3: Address translation through the use of a shadow address space.

4.7 Software Interface

The NI Software Interface is essentially the programmer's view of the NI, i.e. the provided Application Programming Interface (API). In its most primitive form the NI Software Interface can simply expose the NI data transfer mechanisms described in subsection 4.5. For instance Programmed I/O (PIO) corresponds to transferring data through consecutive uncached stores and loads. Likewise a user-level DMA API directly matches the DMA-based transfer mechanism offered by a NI.

However, in certain cases, it can be advantageous to design richer software interfaces with higher-level features using the data transfer primitives offered by the NI. Such interfaces are commonly provided in the form of queues. Queue semantics have been used in quite a few systems, such as the Cornell U-Net [32], the Mitsubishi DART [39] and the Wisconsin Coherent Network Interfaces (CNIs) [40].

For outgoing queues, the sender enqueues data at the tail of the queue by issuing stores and triggers a message dispatch by updating the tail pointer. The NI needs to read the data, construct the outgoing message and advance the head pointer. Likewise for incoming queues the receiver dequeues from the head of the queue by issuing loads and informs the NI about complete message

reception by updating the head pointer. Again, the NI is responsible for writing incoming data and updating the tail pointer.

Queues may be located in dedicated NI buffer space, main memory or the cache, while hybrid solutions, where the incoming and outgoing queues reside in different locations, are also viable. An advantage of providing high-level interfaces is that the NI data transfer mechanisms are decoupled from the NI API, allowing for future NIs to preserve and implement the same software interface using alternative data transfer mechanisms.

The software interface also defines the mechanism for notifying the processes about incoming messages. The two most widely used notification schemes are through interrupts and polling. In the first case, each arriving message causes an interrupt, which needs to be handled by the operating system and is therefore a very costly method, mostly suitable to scarce traffic. In the second case, user-level software needs to poll on specific NI locations to find about the arrival of new messages. Polling is a lot cheaper and efficient when there is lots of incoming traffic. However if there is little or no incoming traffic polling can greatly reduce the processor performance. To get the best of both cases, many systems use a combination of interrupts and polling [41].

4.8 NI Complexity

As seen to this point, NIs can range from very simple and primitive peripheral devices to very complicated, feature-rich, semi-autonomous hardware systems. NI complexity seems to be tightly related to NI placement. A good metric for evaluating NI complexity is to compare the NI to the processor with respect to hardware logic and buffer space.

In one extreme, the NI is chosen to be almost as complex as the processor itself. This is a viable solution when the NI is located relatively far from the processor, usually on the I/O bus. Such an example is the Stanford Flash Multiprocessor [38], which uses the Magic processor inside each NI to implement various protocols. In the Intel Paragon massively parallel supercomputer [42] each node has two identical processors; one for computation and the other, exclusively for NI utilization.

As NIs move closer to the processor their complexity usually decreases, for two reasons. Firstly, a device placed closer to the processor needs to operate at the same speed range as the processor, which in turn means that simpler hardware has to be used to achieve such high speeds. Secondly, the closer the NI is placed to the processor the more opportunities exist to share

resources. For instance, if the NI is placed on the memory bus, it makes sense to use main memory regions instead of dedicated buffer space.

When considering future chip multiprocessor systems with hundreds of nodes, where the NI is placed even closer to the processor (e.g. in the cache), it makes even more sense to keep the NI very simple, since area is limited anyway and it also keeps the per node overhead low. Register-mapped NIs are placed so close to the processor that they cannot afford to be complex. These NIs are so intrusive, that each feature added risks increasing the processor clock cycle period, thus degrading the whole system's performance. For the same reasons register-mapped NIs also offer very limited buffer space.

5 Proposed NI Design

Based on the design issues addressed in section 0, this section describes a proposed NI design, well-suited to chip multiprocessors (CMPs) and other dense and tightly integrated parallel machines. The presented NI was designed for use in the new prototyping platform currently being built at the CARV laboratory of ICS, FORTH. This prototyping platform consists of many interconnected FPGA-based prototyping boards and its development is part of the cross-European "Scalable Computer ARChitecture" (SARC) project (www.sarc-ip.org), which investigates future NI design and interprocessor communication mechanisms.

5.1 Design Goals / Desired Features

Future chip multiprocessors will contain hundreds to thousands of interconnected processing cores and memory blocks. In tiled architectures processing cores are usually placed in a grid formation surrounded by memory blocks as seen in Figure 5.1. Either a fixed amount of memory blocks can be statically allocated to each processor, as seen in (1) of Figure 5.1. or memory blocks can be dynamically allocated on demand, according to each processor's current memory needs, e.g. as in (2) of Figure 5.1. In either case, it is assumed that these memory blocks offer very low latency and very high throughput (through the use of interleaving) equivalent to the level one (L1) or level two (L2) cache of a processor.

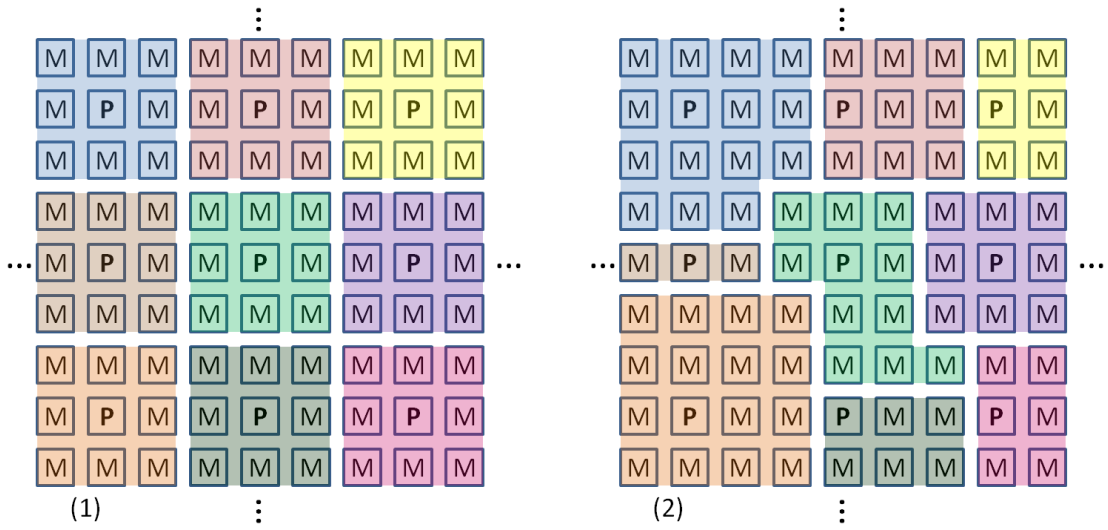

Figure 5.1: Tiled CMP Architecture.
(1) Static memory block allocation. (2) Dynamic memory block allocation.

The NI described in this section is intended to be included next to each one of the hundreds or thousands processing cores. Thus, each parallel system will contain a vast number of such NIs.

To afford placing such a large number of NIs inside a CMP, it is of great importance to keep their cost and complexity very low (compared to the processor). Such NIs are referred to "lightweight" NIs.

Traditional NIs are usually expensive because they require dedicated memory for buffering messages and implement complex protocols through the use of dedicated costly hardware, e.g., protocol processors. To avoid use of dedicated buffer space, which leads to the underutilization of the available memory, the proposed NI can dynamically share available resources, such as memory, with the node processor. For instance, nodes that perform communication-intensive tasks can allocate most of their memory to the NI for message buffer space and storage of connection metadata. On the other hand, nodes performing mainly computation can use most of their memory as a local processor cache or scratchpad memory for storing computation data. An additional advantage of the NI and the processor sharing memory is that it helps the implementation of zero-copy protocols.

As for dealing with complex protocol processing there are two ways to confront this problem. Firstly, in a fixed and constrained CMP environment just employing simpler protocols may suffice, as costly protocols are usually required in more diverse and complex environments, like the Internet. Secondly, the presence of so many processors on a single chip makes it more tolerable to have the main processor handle higher-level protocols. Ultimately, maybe a small set of processors can be solely dedicated to protocol processing.

In addition to keeping the NI cost low and sharing resources with the node processor, a third design goal is to offer a set of primitives for versatile powerful communication with minimal overheads. The first aspect of powerful communication is offering very low NI access latency - in the order of the time needed to execute a few instructions – useful for short synchronization and control traffic. This is achieved through a versatile queue mechanism for sending and receiving short messages. The proposed Message Queues support one-to-one, one-to-many and many-to-one communication. The second aspect of powerful communication is offering high-bandwidth, low-overhead communication for bulky transfers. This is achieved by adding support for Remote DMA (RDMA). To minimize overheads, RDMA operations are designed to be carried out in user-level, without any OS intervention.

With the advent of CMPs future parallel machines will consist of thousands or even million of nodes making scalability a major issue. In a CMP environment, on-chip memory is a very precious resource that should be carefully managed to preserve scalability. In a large-scale parallel system, NI data structures containing one entry per each available node in the system would be prohibitively large, flooding the entire on-chip memory. Hence, no data structure in the

NI should be required to hold as many entries as there are nodes in the system. In the proposed NI, connection-related data are efficiently stored in a data structure called the "Connection Table" that follows the above guidelines.

5.2 Target Hardware/System

The prototyping platform for which the proposed NI was designed consists of many interconnected FPGA-based boards that aim to replicate the behavior of a CMP. Each node of the system is implemented using the "Xilinx XUP" FPGA-based development board provided by Xilinx. Each "Xilinx XUP" board hosts a Xilinx VirtexII-Pro 2VP30 FPGA, which has two embedded PowerPC processor cores, designed by IBM. Additional larger FPGA-based boards [ref] were configured as switches to connect the "Xilinx XUP" boards together through RocketIO links.

The available hardware influenced and occasionally constrained NI design aspects. For instance, although the prototyping platform was intended to replicate a CMP the characteristics of the FPGA-boards that represented CMP nodes were quite different from the characteristics of an actual CMP node. In particular, communication between two FPGA boards suffers from very high latency compared to the communication latency between two nodes in a CMP. Furthermore the performance of the processor in each FPGA is quite low and disproportional to the bandwidth offered by the interconnection network.

5.2.1 "Xilinx XUP" Board

The "Xilinx XUP" board [43], which can be seen in Figure 5.2, is a complete logic emulation system that can function as a digital design trainer, a microprocessor development system or a host for embedded processor cores and complex digital systems. The system is configured around the VirtexII Pro series FPGA manufactured by Xilinx.

The Xilinx XUP board includes 4 serial RocketIO ports, which can support a variety of serial communication protocols at speeds up to 2.5 GB/s. The board also features one DDR SDRAM DIMM slot that can accept up to 2Gbytes of DRAM, as well as a compact flash memory card slot. A variety of I/O ports are supported, which include 100 Mbit Ethernet, USB 2, VGA Video, Stereo Audio, SATA, PS/2, RS232 and JTAG ports.

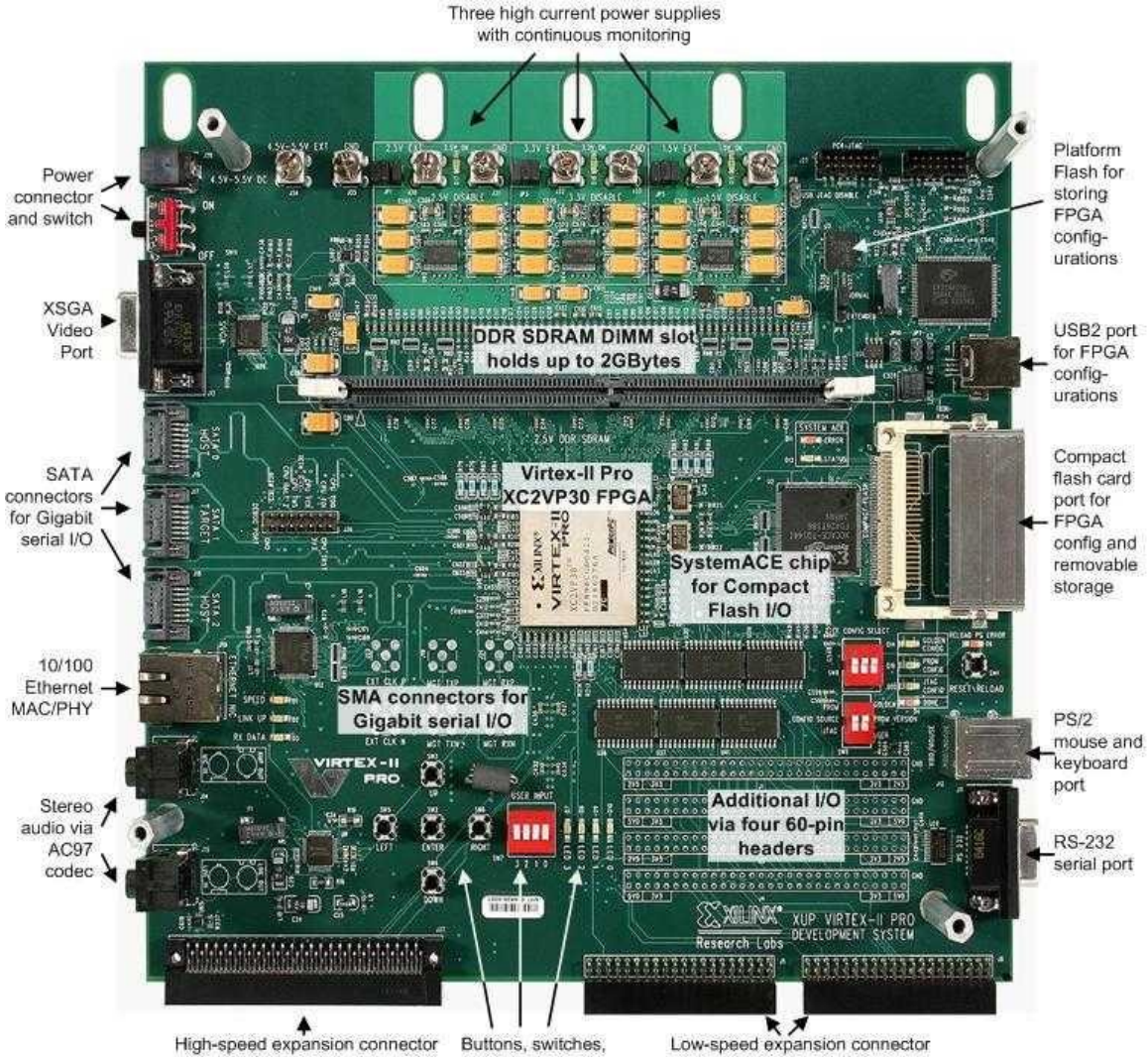

Figure 5.2: The Xilinx XUP board

Additional I/O pins are provided through four 60-pin headers for logic analyzer-based debugging or pattern generator stimulus. Finally there are 4 LEDs that the user is free to use as debugging tools or simply for visual indication of the system's status. Nine DIP switches and five push-buttons are also present on the board: one for resetting and configuring the FPGA and four that are user configurable.

5.2.2 VirtexII Pro

The FPGA on the "XilinxXUP" board belongs to Xilinx's VirtexII Pro family of FPGAs [2] and its specific device type is xc2vp30. It is built on a 130 nm 9-layer copper process technology and is capable of over 400 MHz operation frequencies, offering 30816 logic cells, 2448 Kbits of embedded memory (Xilinx BlockRAM or BRAM) and 136 18x18 Multipliers. It also includes 8 Digital Clock Management Blocks (DCMs) and 8 high-speed RocketIO transceivers. The heart of

the VirtexII Pro FPGA accommodates two IBM PowerPC processor cores capable of operating at over 300 Mhz.

5.2.3 Node Configuration/Overview

A block diagram of each node's configuration is presented in Figure 5.3. The dashed box represents the Xilinx XUP board. The inner box corresponds to the VirtexII Pro FPGA, which hosts a PowerPC processor clocked at 266 MHz along with dual-port on-chip memory and the Network Interface (NI). The on-chip memory consists of 8 memory banks allowing for high throughput through memory interleaving. Each memory bank is 32 bits wide and comprises of dual-port Xilinx BRAM blocks. The PLB bus connects the PowerPC to one port of the on-chip memory and to DRAM external memory, located on the "Xilinx XUP" board. The NI connects to the second port of the on-chip memory module. In order to maximize throughput, future plans include also utilization of a dedicated bus, called OCM , to connect the processor to the on-chip memory.

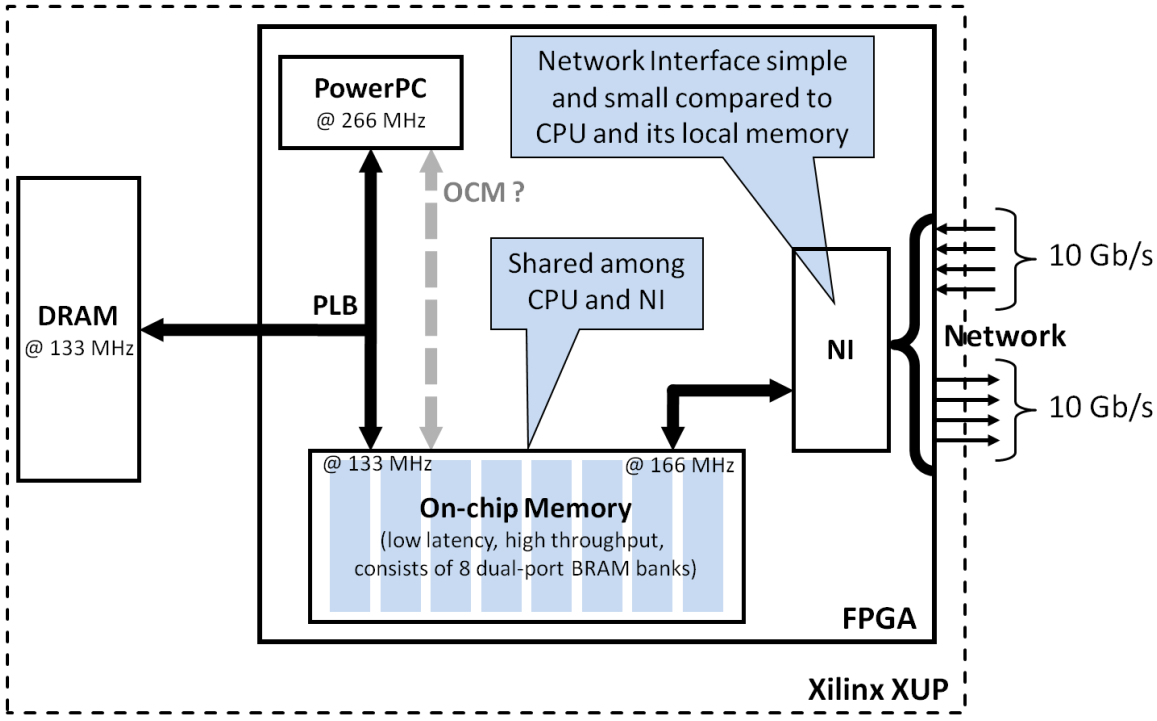

Figure 5.3: Node Configuration

Initially the on-chip memory will operate as a local scratchpad memory. However, the long-term plan is to dynamically configure some portions of the on-chip memory as scratchpad memory and configure some other portions as typical level-one cache memory. Scratchpad memories can be explicitly accessed by software, using a dedicated part of the address space. They serve as high-speed local storage and can be thought of as software controlled caches.

Scratchpad memories are widely used in embedded processing because the offer deterministic access times. On the other hand, cache memories are "transparent" to the software and do not match to a specific part of the memory address space. Although the NI presented here only requires the presence of scratchpad memory, it could also benefit from the existence of a cache memory.

5.3 Communication Primitives

The proposed NI offers two fundamental, powerful communication primitives: Queues and Remote DMA. Message Queues are intended for low latency communication, mainly synchronization and control messages or small low-overhead data transfers. Remote DMA minimizes processor involvement in communication and is well suited for bulky data transfers and facilitates zero-copy protocols.

5.3.1 Message Queues

Message Queues are powerful communication and synchronization primitives and play a very central role in the proposed NI design. They offer minimal overhead, low-latency and FIFO-based one-to-one, one-to-many and many-to-one communication. Using one-to-one queues, which is the simplest case, a source node can send messages by specifying the ID of a queue residing on a remote node. One-to-many Queues allow for a source node to transmit many messages destined to different queues, residing on remote nodes, through a single local queue. Likewise many-to-one queues allow a node to receive messages from multiple senders in a single queue. Examples of the different type of queues can be seen in Figure 5.4.

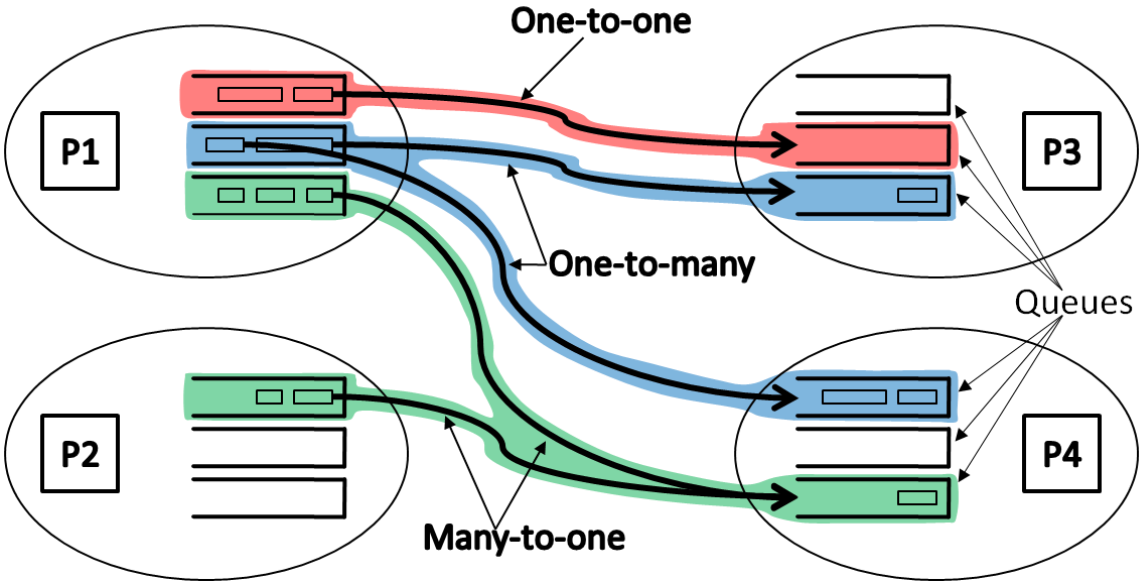

Figure 5.4: Message Queue Types.

When sending a message to a queue the sender only specifies the ID of the queue where its data will be placed. The exact memory location for each incoming datum is only determined upon arrival at the receiving node where the queue is maintained. The receiving NI maintains the write pointer, and atomically increments it upon each message arrival. This behavior makes many-to-one queues a valuable synchronization primitive. One-to-many queues can be used as a powerful job dispatching mechanism. While Queues are excellent for job dispatching and synchronization purposes, they are not well suited to bulk transfers, because data arrive at non-controlled locations, and will usually have to be copied elsewhere to be used, especially when they are to be accessed multiple times or in non-sequential order.

5.3.2 Remote DMA

Remote Direct Memory Access (RDMA) allows data residing in the local memory of one node to directly move into a remote node's memory and is well suited to pair-wise (one sender, one receiver) producer-consumer type communication. There are two types of RDMA operations, RDMA Write and RDMA Read. In RDMA Write, the transfer initiator specifies the source of the data transfer in one of its local memory regions, and the destination of the data transfer within a remote memory region on another node. In RDMA Read, the transfer initiator specifies the source of the data transfer at a location in a remote node's memory region, and the destination of the data transfer within a local memory region. RDMA Read can easily be implemented using RDMA Write. (revise) Instead of issuing a RDMA Read operation, a message can be sent to the remote node requesting a RDMA write to be performed.

RDMA is the basic data transfer operation needed to enable zero-copy protocols. Zero-copy protocols deliver data in-place, i.e. at the precise memory location where these data will eventually be needed, so as to avoid the receiver having to copy them from one memory location to another. Copying data induces major costs in latency, memory throughput and energy consumption, which makes zero-copy protocols a very important factor in overhead reduction. Every RDMA network packet specifies its destination address, where its data should be written, and can therefore be delivered directly in place. In a sense RDMA packets are somewhat "autonomous", because upon arrival they do not rely on protocol software to copy data from a temporary buffer to their eventual final location.

A central issue in implementing RDMA is to properly integrate it with virtual-to-physical address translation and protection. The proposed NI supports DMA operations for data residing in memory regions that have been previously registered through OS interventions, i.e. a system call. The memory registration process defines one or more both virtually and physically contiguous pages as a Memory Region. When a Memory Region is registered, every page within the region is

locked down in physical memory. This guarantees that the memory region is physically resident (not paged out) and that the virtual to physical translation remains fixed when the NI is processing requests that refer to that region. Memory is registered on a per process basis, which means that all threads within a process can access the same registered memory regions. Additionally if two processes share memory, which means that they both own mappings to the same piece of physical memory, they can both register the same memory region.

5.4 Connections

Connections form the mechanism for sending messages and initiating RDMA operations to enable communication among two or more nodes. Each Connection describes a Queue pair, that offers bidirectional communication, and holds the information needed for performing RDMA, as well as various Queue, DMA and Flow Control meta data. Each such Queue pair, consisting of an incoming Queue and an Outgoing Queue, allows for both one-to-one or many-to-many communication. The Outgoing Queue of a Connection is used to post commands that trigger RDMA operations and to send messages to remote nodes. Similarly, messages and notifications concerning RDMA operations that have been completed are received through messages appearing in the incoming Queue of a Connection.

Connections comprise a powerful and versatile communication mechanism and come in two "flavors", one-to-one and many-to-many. One-to-one Connections, offer bidirectional pair-wise communication between two nodes with support for both messages and RDMA operations. The Incoming and Outgoing Queues of one-to-one Connections on a node pair are only allowed to send and receive messages to and from each other and the same goes for RDMAs. Such a scenario can be seen in (1) of Figure 5.5, where Node A and Node B have established a set of bidirectional one-to-one Connections. When two nodes are to exchange lots of data, one-to-one connections is the preferred method for communication, because the proposed NI offers special support to minimize overheads and latency for this kind of connections, as will be seen in subsection 5.7. In addition one-to-one connection offer higher levels of protection, as will be explained in subsection 5.8.

While one-to-one connections are well suited to a few pairs of nodes transferring lots of data, they do not scale well, especially in the case of some large-scale parallel applications, which require sporadic low-traffic global communication for synchronization and notification purposes. For instance, in a parallel application where every node in the system needs, at some point during

the execution, to send a little piece of data to a single particular node[1]. If one-to-one connections were the only available communication mechanism, this would require that the receiving node, that collects the data, establishes an enormous amount of connections, one with each node in the system. This way of implementing many-to-one communication is terribly inefficient due to excessive resource consumptioin. The same is true for one-to-many communication patterns. For example, if during the execution of a parallel program a single node desires to send short messages to all of the nodes in the system, a massive number of one-to-one connections would need to be established.

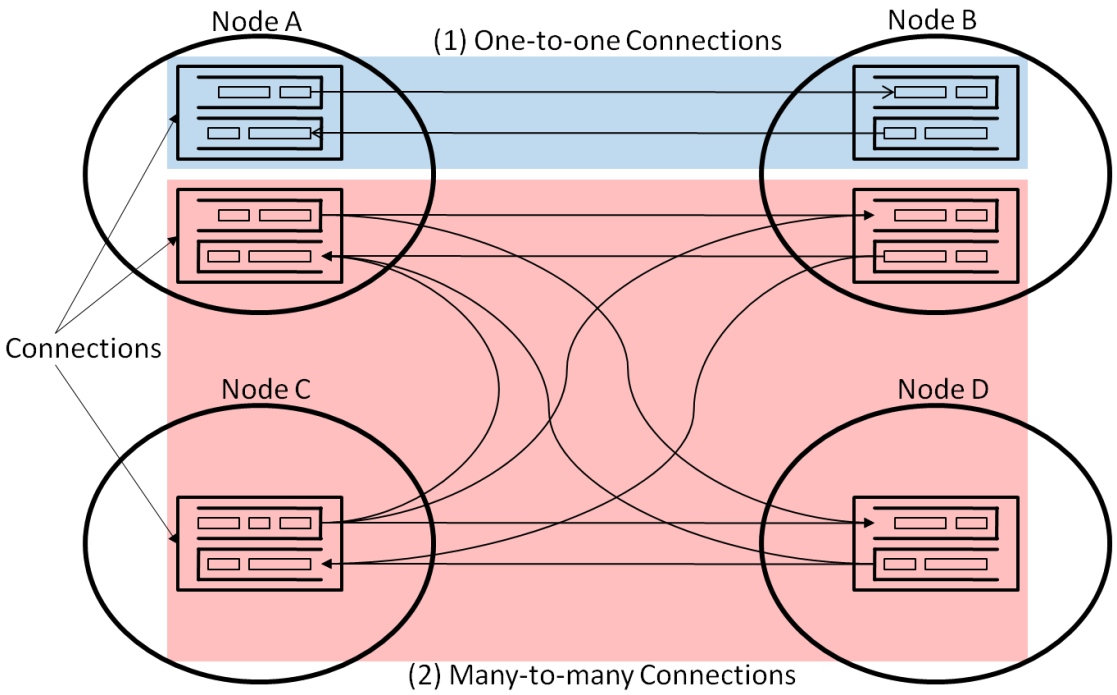

Figure 5.5: Connection Types

To alleviate the above problems and satisfy global communication needs the proposed NI offers support for many-to-many connections. When using many-to-many connections it is possible for a node to send messages to multiple destinations through a single Outgoing Queue and to receive messages from multiple sources in a single Incoming Queue. Such an example can be seen in (2) of Figure 5.5. Node C uses a single connection to communicate with multiple nodes (Nodes A, B and D). Similarly Node A uses a single connection to receive data from multiple nodes (Nodes B, C and D). Many-to-many connections minimize the required resources for

[1] This communication pattern resembles the transpose operation in the Complex 1D FFT application of the SPLASH benchmarks [45]

communication, but impose additional initiation overhead. More on this matter is presented in subsection 5.7.

It is important to realize that Connections do not constrain the software. Quite the opposite; they allow software to dynamically adjust the type and cost of communication. If a node has a lot of available memory for the NI, it can serve most of its communication needs through dedicated one-to-one connections. If memory resources are becoming sparse, sets of several one-to-one connections can be grouped into fewer many-to-many connections. Ultimately in a large-scale parallel application it is up to the software to choose the portion of the required communication that will be served through one-to-one connections and the portion that will be served through many-to-many connections. As a rule of thumb it is usually beneficial to satisfy the needs for global communication through many-to-many communication, so as to reduce overheads and resource consumption.

An example of a communication pattern representative of several scientific parallel applications[1] is shown in Figure 5.6. Nodes are placed in a grid formation and spend most of their time communicating with their four neighbors which is represented by the thick lines. Occasionally every node also needs to take part in global communication, which is represented by the thin lines. This mainly consists of exchanging short control and synchronization messages with the rest of the nodes in the system. The naïve and inefficient choice would be to create a one-to-one connection for each line. However this would lead to the establishment of countless connections leading to inefficient resource consumption and huge overheads.

A better approach is to only use one-to-one connections for the communication represented by the thick lines and use many-to-many connections for the thin lines. Four one-to-one connections for communicating with neighboring nodes and one many-to-many connections to satisfy global communication needs. This way establishing 5 connections suffices for each node; If plenty of memory resources are available, to achieve better performance, even more than one many-to-many connections per node could be used for global communication to initiate the required transfers in a pipelined fashion. It is also important to note that the number and type of connections can also potentially be dynamically handled by low-level software transparently from the application software.

[1] This communication pattern resembles the transpose operation in the Ocean Simulator application of the SPLASH benchmarks [45]

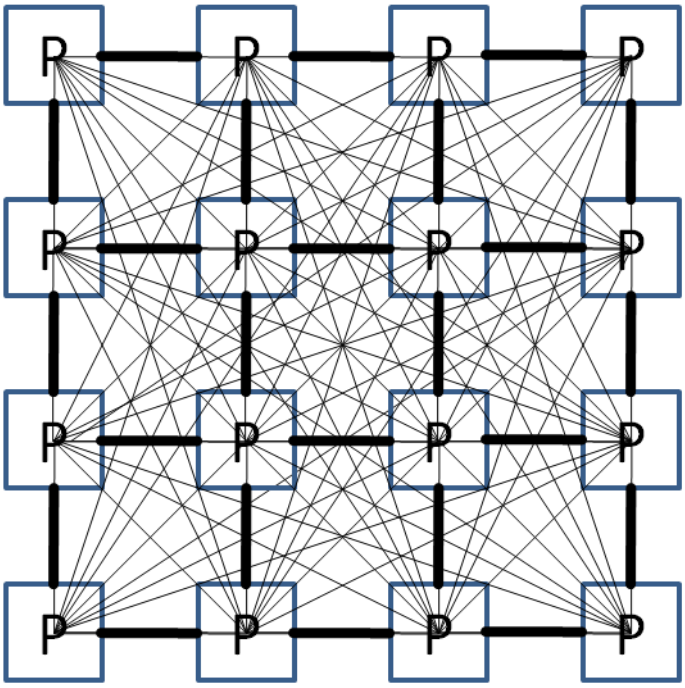

Figure 5.6: A Grid Communication Pattern.

5.5 Connection Table

From the hardware perspective, each Connection encapsulates all of the necessary information to support the communication primitives described in subsection 5.3. Information regarding Connections is stored and organized inside a data structure, called the Connection Table (CT) in the form of Connection Table Entries (CTEs). The CT, which plays a very central role in the proposed design, occupies a contiguous variable-size piece of scratchpad memory, which is accessible by both the NI hardware and the node software and can be seen in Figure 5.7. CTEs are uniquely identified by their Connection IDs (CIDs), which essentially are the scratchpad memory addresses that correspond to CTEs. As described later, in subsection 5.6, which describes the packet format, CIDs are contained in packet headers and are used to directly index the CT and find the CTE that is needed to handle an arriving packet.

Each CTE has the capacity to host one connection. This can be a one-to-one connection, a many-to-many connection or a hybrid connection, which is a combination of a one-to-one and a many-to-many connections that share some common characteristics. A CTE consists of several fields that store Connection-specific data and the information necessary for managing Queues and performing RDMA. CTEs are also well-suited to storing other types of metadata concerning Connections, such as Flow Control Information.

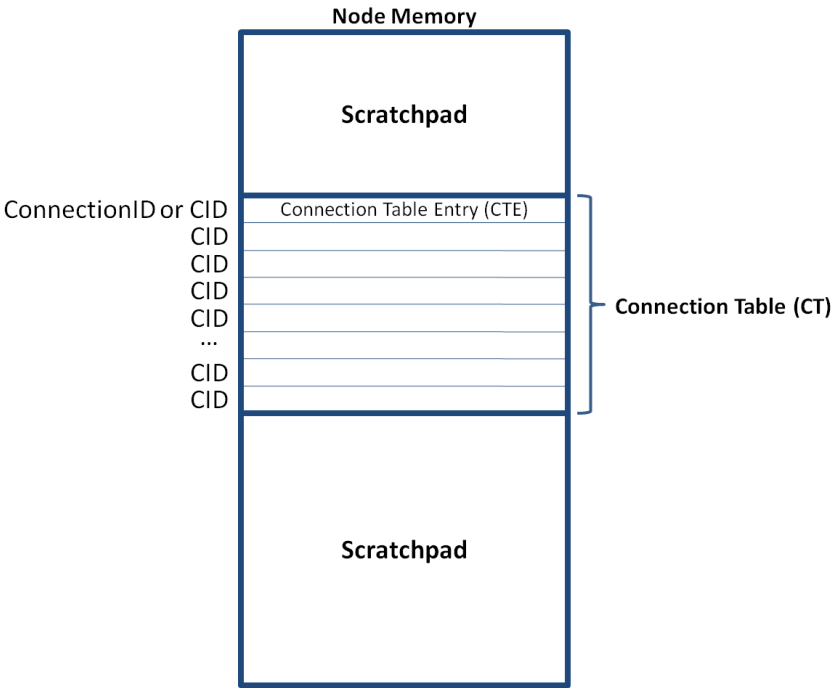

Figure 5.7: Connection Table in scratchpad memory.

The fundamental information contained in a CTE is listed below:

- **Destination Info:** Contains information about the destination, if the connection is used for one-to-one communication. This may include the id of a remote node, used for routing and the CID of the CTE that receives traffic on the remote node
- **Protection Info**: Contains Protection information. For example this could be a Process Group ID that is used to form groups of processes and offers protection for many-to-many connections.
- **Incoming/Outgoing Queue Info**: Describes Outgoing/Incoming Message Queues by specifying the location of Queues in memory, as well as head and tail pointers.
- **Incoming/Outgoing RDMA Info**: Contains info about RDMA operations, such as a list of the preregistered RDMA-capable regions.
- **Flow Control Info:** Depending on the interconnection network, each CTE may need to dedicate a few bits for flow-control information. For instance, this can hold credit information or just be a pointer to the next CTE waiting to be serviced.
- **Other Info:** In addition to the above other connection-specific information can be stored in each CTE. An example is a field that specifies the kind of notification for arriving messages.

Figure 5.8 shows how information is arranged in the CTE devised for the proposed NI. The numbers enclosed in parentheses represent number of bits, each CTE occupying 256 bits. The

length of the CTE was chosen to match the width of each node's memory in the prototyping platform, which consists of 8 interleaved 32-bit banks. An effort was made to place independent information in independent banks, to allow parallel accesses to a CTE.

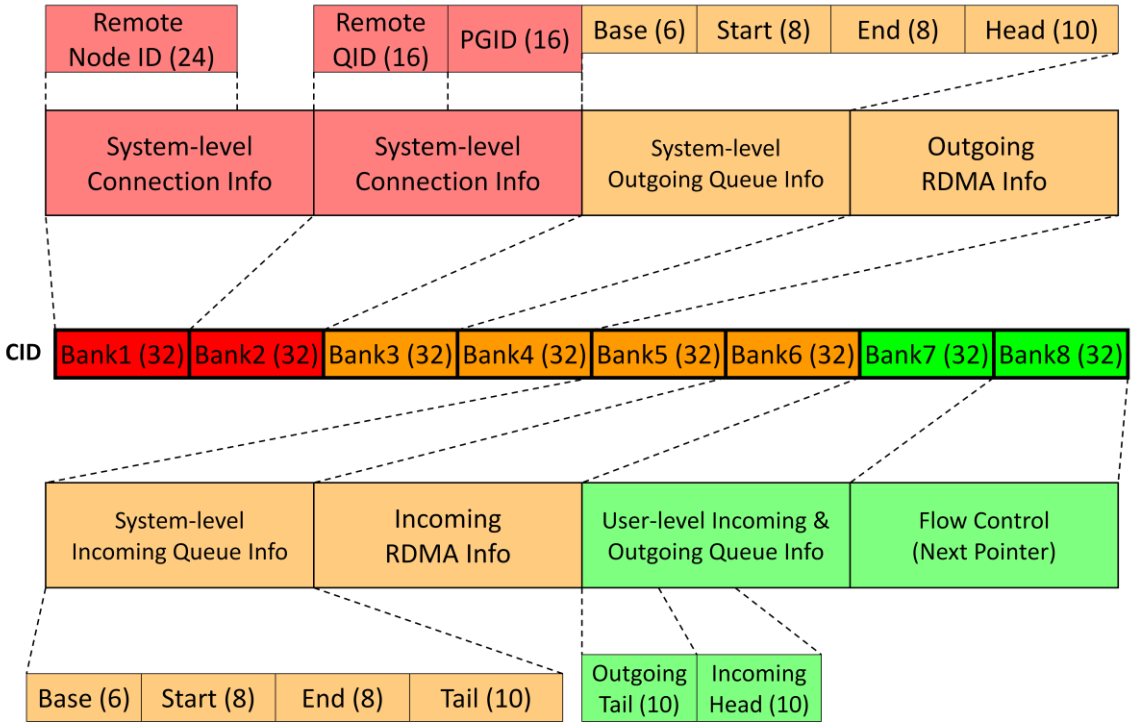

Figure 5.8: Connection Table Entry.

The first two memory banks contain information about the Connection. In a one-to-one or hybrid connections the first bank contains the Node ID of the remote node. A Node ID uniquely identifies a node in the system and is used for routing purposes. The second bank contains two fields. The first field is again used in one-to-one or hybrid connections and is the CID identifying a CTE in the CT of the remote node. The second field defines a Process Group ID (PGID) which is used to offer protection among nodes communicating with many-to-many connections. Processes running on different nodes with CTEs that are sharing the same PGID, belong to the same Process Group. A PGID value of zero means that this CTE is exclusively used for a one-to-one connection. The next four memory banks, banks 3, 4, 5 and 6, are used to store information related to outgoing and incoming Message Queues or RDMA.

The third and fifth memory banks contain compacted information about the Outgoing and Incoming Queues of a connection. The description of a Queue consists of a set of upper and lower Queue boundary addresses and a head and tail pointer, each of which is 16 bits wide which use a common 6-bit base address in the proposed CTE. These 16 bits correspond to 64K of 4-byte word addresses, which means that in the presented CTE a queue can be located anywhere within a 256 Kbyte region in the scratchpad memory. In our prototyping platform 256 Kbytes is far beyond the

available scratchpad memory of a node, which means that queues can be located anywhere within the scratchpad.

Complete definition of a Queue in the proposed CTE requires values of the Base, Start, End, Head and Tail fields. For Outgoing Queues these values can be found in the third and seventh bank. Similarly for Incoming Queues these values reside in the fifth and seventh bank. Using these values found in the CTE it is possible to "reconstruct" the Queue boundary addresses and the head and tail pointers, as is shown in Figure 5.9.

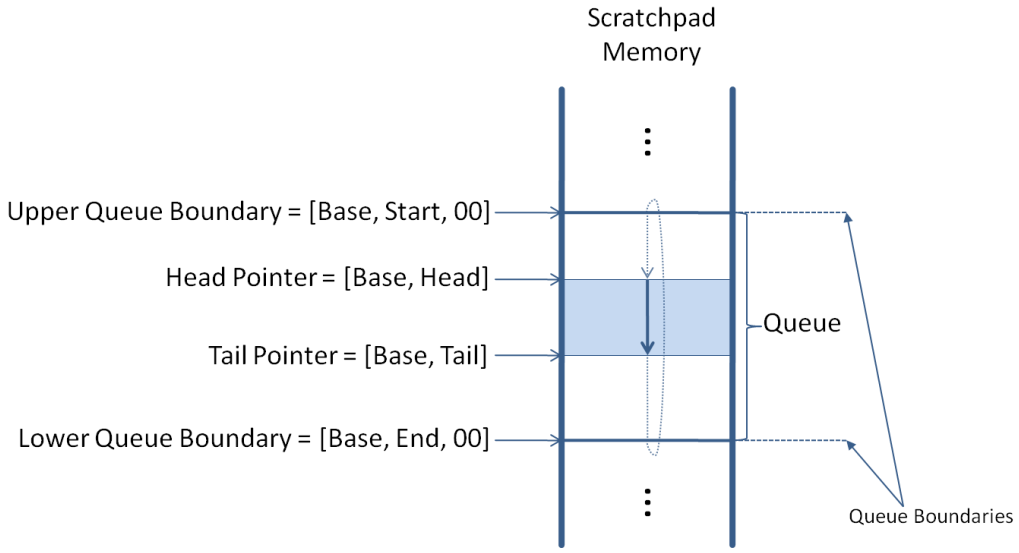

Figure 5.9: Queue Representation in Scratchpad Memory.

Getting the byte address for the upper Queue boundary, requires concatenation of the Base value with the Start value and appending two zeros. Likewise, the byte address of the lower Queue boundary requires concatenation of the Base value with the End value and appending two zeros. Appending two zeros means that queue boundaries cannot point to any valid 4-byte word address in the scratchpad memory. In fact, upper and lower Queue boundaries can be set every four memory words. Thus the size of a queue needs to be a multiple of 4 words and the smallest possible queue size is 4 words. The maximum queue size is 1024 words.

The process for reconstructing the head and tail pointers is similar. Getting the word address for the head pointer of a queue requires concatenation of the Base value with the Head value. Similarly, the word address of the tail pointer requires concatenation of the Base value with the Tail value. The resulting 16 bits correspond to the head and tail pointers. As described in more detail in subsection 5.7, Outgoing Queues have their tail pointer written by software and their head pointer advanced by hardware. Conversely, in Incoming Queues the tail pointer is advanced

by hardware while the head pointer is written by software. When a head or tail pointer reaches the lower queue boundary, its value has to be set to the upper queue boundary.

The fourth and sixth memory banks contain information regarding Incoming and Outgoing RDMA operations. A simple approach is to have each CTE define two small fixed-sized contiguous physical memory regions for use with outgoing and incoming RDMA operations. The size of each such region can be set to a few 4 Kbyte pages, e.g. 64 Kbytes. The memory pages corresponding to these physical address regions need to be pinned by the operating system so that they are always resident in memory and never swapped out. (As described in subsection 5.7 each RDMA packet travelling through the network can not specify an absolute physical memory target address, but only a RDMA offset. Upon arrival this offset is added to the Incoming RDMA base address of the CTE to calculate the final memory destination for the arriving data.) Similarly, when initiating a RDMA operation only the offset of the data to be transferred is specified, which is added to the Outgoing RDMA base address to find the location of the data in physical memory.

A more advanced and versatile approach is to associate more than one physical memory regions with each CTE. In this case the RDMA information in each CTE could point to a location in scratchpad memory that hosts a small list of valid RDMA base addresses for use with Outgoing and Incoming RDMA transfers. Such memory regions can also have different lengths. Ultimately, these lists of RDMA base addresses can be shared among several CTEs to more effectively utilize the available scratchpad memory. A drawback of this approach is that it requires a few additional memory accesses for each RDMA operation. Again, all of the pages belonging to the specified physical memory regions need to be pinned by the operating system. This solution resembles the Buffer Descriptor Tables found in the SiCortex SC5832 and SC648 parallel machines [44].

The remaining memory bank (Bank 8) of a CTE can be used for other kinds of information. A possible candidate is Flow Control information, which can have the form of a pointer to another CTE that needs to be serviced next. Offering a “Next pointer” for each CTE is quite useful and allows for maintaining linked-lists of CTEs. Alternatively, this bank can hold the number of available credits. In addition to flow control information this remaining memory bank can store configuration parameters for a CTE. For instance there could be a field indicating the type of desired notification mechanism for incoming traffic. This field could then be used to turn on/off interrupts for arriving messages.

It is important to keep in mind that the CT structure presented in this subsection was custom tailored for a specific prototyping platform. For instance, the placement of different fields in different memory banks was done in a way that allows for parallel access in frequently used fields containing independent data. As an example, Incoming and Outgoing Queue information is stored

in different memory banks, avoiding access conflicts in the CTE for simultaneous incoming and outgoing messages.

The fact that the interleaved memory in each node has a width of 256-bits allowed for spreading CTE information to occupy the whole memory width using the 8 interleaved banks. Under other circumstances, for CTEs residing in non-interleaving narrower memories, it might have been more efficient to have each CTE occupy less space by compacting, migrating or even eliminating some fields. The essence of CTEs is not the specific placement of the data inside them, but that they encapsulate all of the data required to support the available communication primitives of a system.

5.6 Packet Format

When travelling through the interconnection network, messages destined to remote queues and RDMA data need to organized in packets. This subsection presents the proposed packet format for queue messages and RDMAs. Although the proposed NI was intended for a prototyping platform, the objective was to create a packet format that could be used unaltered in a real large-scale parallel machine, i.e., to be as close as possible to the intended, "real" system. For instance, the selected packet header offers support for up to 16 million nodes, which is many orders of magnitude greater than the number of nodes to be ever present in our prototyping platform, but is quite realistic for future systems. In general an effort was made to not restrict the design by constraints that were specific to this prototyping platform.

5.6.1 Packet Format Considerations

When designing the packet format of a system, there are several trade-offs to consider. For instance, offering support for very large packets and minimizing the required buffer space in the interconnection network and at the NI are two conflicting goals. Large packets minimize header overhead, but require excessive buffer space in the interconnection network and at the NI. Furthermore, using a unified same-size packet header for all kinds of traffic may lead to overheads in the form of underutilized space, but greatly simplifies the switch and NI designs. For instance, switches do not need to implement separate logic for different kinds of packets; rather, they can switch all of the packets in a uniform manner.

Another important issue is how to protect the packet against errors, boiling down to the placement and the number of available checksums. One solution is to use a single checksum at the end of the packet, both for the packet header and the packet data. However this forces the NIs, and possibly the switches too, to store the whole packet before finding out if it is corrupt. An

approach that tries to alleviate this problem places two checksums inside each packet; one at the end of the packet header and another at the end of the packet body. This way, the network switches need not wait for the whole message before forwarding it. Upon receiving the packet header and checking it for errors against the checksum the packet can be dropped or forwarded using cut-through switching.

5.6.2 Proposed Packet Format

Figure 5.10 presents the proposed unified packet format for both queue messages and RDMA packets. The numbers enclosed in parentheses represent number of bits, which makes the entire header of any network packet 96 bits or (12 bytes) long. The packet header is pieced into 4-byte words, because this is the memory word size and datapath width in each node. Essentially, this means that as a message is being received, its header will be processed 4 bytes at a time, which influences the placement of information in the packet header.

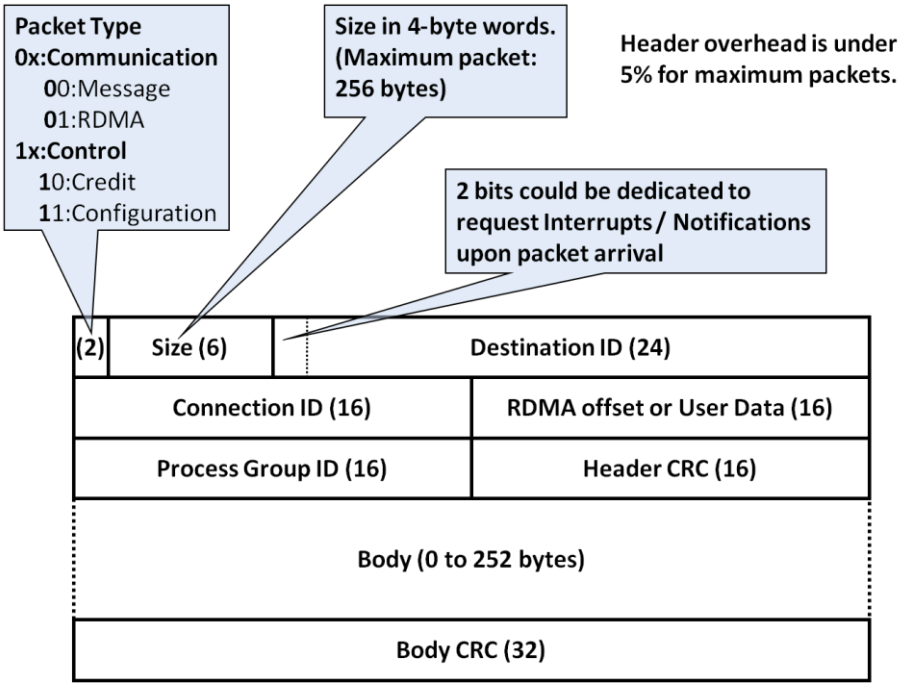

Figure 5.10: Proposed Unified Packet Format for messages and RDMAs

The first 2 bits of the header belong to the Packet Type field. Specifically the first bit of the Packet Type is used to distinguish communication packets from control packets, i.e., 0 for communication and 1 for control packets. Control packets are consumed by the network hardware, i.e. the switches and the NIs. The second bit of the Packet Type further classifies a packet. For communication packets the second bit is used to distinguish queue messages from RDMA packets, i.e., 0 for queue messages and 1 for RDMA packets. In the case of control packets the second bit separates flow control packets that carry credits and configuration

messages consumed by the NIs. The next 6 bits of the packet header define the size of the packet, which is measured in 4-byte words. Maximum packet size is 256 bytes. The remaining 24 bits uniquely specify the destination node and are used for routing purposes. A maximum of 16.777.216 nodes are supported.

The second 4-byte word of the packet header is used to specify the Connection ID and contains a 16-bit field that is differentiated for queue messages and RDMA packet to supply the RDMA offset or user data. The Connection ID field identifies a specific connection within the destination node and is used to index the destination node's Connection Table, which is presented in more detail in subsection 5.5; 16 bits allow for 65.536 connections per node. The Process Group ID field corresponds to a group of processes that trust each other and is used for protection purposes. 16 bits allow for the existence of 65.536 process groups. As presented later, in subsection 5.7, setting the connection ID or the Process Group ID to 0 has a special meaning.

The third 4-byte word of the packet header contains a 16-bit field that is differentiated for queue messages and RDMA packets. For queue messages these 2 bytes can be used for arbitrary user data. It is important to note that tiny messages carrying a payload of only 2 bytes can be sent using this field in zero-sized packets. For RDMA packets, this field specifies an offset that is added to a preconfigured base address at the receiving node to determine the final physical memory location for delivering the RDMA data. The remaining 16 bits store the header checksum, which is used to check the integrity of the header while the packet is in transit. The rest of the packet contains the body of the packet, which ranges from 0 to 252 bytes of data, and finally a 32-bit checksum, which is separately calculated for the body of the packet.

5.7 Software Interface

The interface presented to the software for communicating with other nodes is based on the notion of Connections, which were described in subsection 5.4. In order to communicate with a remote node or with a set of remote nodes a one-to-one or many-to-many connection needs to be established respectively. Software uses the Outgoing Queue of a Connection to send messages to remote nodes as well as commands that trigger RDMA operations. Likewise, messages and notifications concerning RDMA operations that have been completed are received through messages appearing in the Incoming Queue of a Connection.

5.7.1 Descriptors

Assuming that a process has already established a connection with the nodes it wishes to communicate, it can send a message or initiate a RDMA operation by posting a Descriptor in the

Outgoing Queue of a CTE. Descriptors are "stripped down" packet headers[1] containing all necessary information for sending a message or carrying out RDMA operation and act as commands towards the NI. Having the software post Descriptors instead of the actual packet headers reduces both communication initiation overhead and, more importantly, plays a basic role in protection as described in subsection 5.8. The proposed NI supports four different kinds of Descriptors which are distinguished by their first two bits and are shown in Figure 5.11 – numbers enclosed in parentheses represent bits. Two of these Descriptors are used to send messages, while the other two are used to initiate RDMA operations.

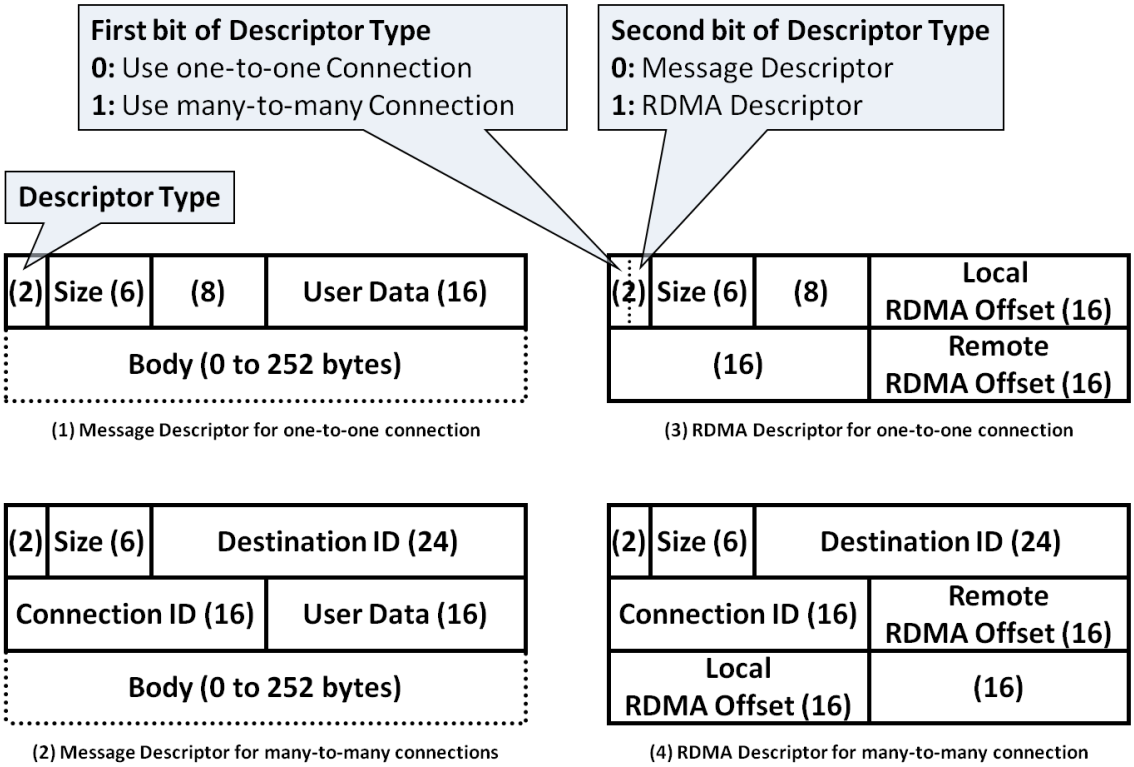

Figure 5.11: Message and RDMA Descriptors

The first type of Descriptors is used to send messages through one-to-one Connections and corresponds to (1) of Figure 5.11. This type of descriptor only consists of a single 4-byte word, offering the lowest possible communication initiation latency. It only contains the size of the message and 16 bits of user data. Zero is a valid message size and results in very small messages that contain only 16 bits of data embedded in the header of a packet. For non-zero sized messages the data to be transmitted has to be placed in the Outgoing Queue following the Descriptor. When

[1] "stripped down" means that they contain a subset of the fields found in a network packet header

using this type of Descriptors the rest of the information required to build the network message packet, namely the Destination ID, Connection ID and Process Group ID fields, are copied from the corresponding CTE. This approach ensures that packets are destined to the node corresponding to the Destination ID of the CTE.

The second type of Descriptors is also used to send messages, but through many-to-many connections and corresponds to (2) of Figure 5.11. Since these messages are not necessarily destined to the Destination ID described in the CTE, additional information is needed in the Descriptor to specify the recipient of the message. These Descriptors consist of two 32-bit words. In addition to the information needed for message Descriptors of one-to-one connections these Descriptors also contain the Destination ID and Connection IDs of the receiving node. As before, if the size is not set to zero, the message body needs to be placed directly after the Descriptor. To construct the network packet header the NI only needs to copy the Process Group ID field from the CTE.

The two remaining Descriptors are for RDMA operations. The third type of Descriptors is used to initiate RDMA operations through one-to-one connections and can be seen in (3) of Figure 5.11. Such Descriptors consist of two 4-byte words containing the size and the local and remote RDMA offsets. The local RDMA offset is added to the Outgoing RDMA base address found in the CTE to get the local location of the data to be transferred. In a similar manner at the receiving node, the NI adds the remote RDMA offset to the Incoming RDMA base address of the CTE to find the destination address for placing data. As with the first type of descriptors, the rest of the information required to build the network RDMA packet, namely the Destination ID, Connection ID and Process Group ID fields, are copied from the corresponding CTE.

Although RDMA is typically used in pair-wise one-to-one communication the proposed design also supports RDMA for many-to-many connections[1]. This feature is useful in satisfying scatter-gather communication patterns. The fourth type of Descriptors is used exactly for this purpose, i.e. for initiating RDMA operations through many-to-many connections and is shown in (4) of Figure 5.11. These Descriptors consist of three 32-bit words. In addition to the information

[1] RDMA over many-to-many communication requires careful coordination. For instance if many nodes are performing RDMA write operations within the same memory region of a receiving node, care needs to be taken to avoid overwriting someone else's data.

needed for RDMA Descriptors of one-to-one connections these Descriptors also contain the Destination ID and Connection ID of the receiving node.

5.7.2 Initiating Communication

To trigger the transmission of one or more network packets after posting one or more descriptors in the Outgoing Queue of a CTE,, the software needs to advance the tail pointer of the Outgoing Queue. When the previous value of the tail pointer is overwritten, NI hardware, that monitors "writes" to the memory banks containing the Tail field of the Outgoing Queue of CTEs, checks for new Descriptors in the Queue and services them. An advantage of this approach is that it allows sending multiple messages or initiating multiple RDMA operations at a time by posting multiple descriptors in the Outgoing Queue. Only when the Tail updated is advanced will the creation and transmission of all of the corresponding network packets be triggered. When the NI hardware services a descriptor, it advances the head pointer, which the software can check to find out if a message has been sent or if a RDMA transfer is complete.

The steps for sending a message or initiating a RDMA operation are as follows:

1. Get Head and Tail pointers of the Outgoing Queue of a CTE to see if there is enough available space to post the Descriptor.
2. Write the Descriptor at the tail of the Outgoing Queue.
3. Advance the Tail pointer to trigger the NI
4. (Poll the Head pointer of the tail to find out if the message/RDMA was sent)

Likewise, the steps for receiving a message or a RDMA notification are as follows:

1. Poll the Tail pointer of the Incoming Queue of a CTE or wait for an interrupt if interrupts are enabled, to find out about received messages or RDMA notifications.
2. Get the Head pointer of the Incoming Queue of a CTE.
3. Read message or RDMA notification from the head of the Incoming Queue.
4. Advance the Head pointer of the Incoming Queue to inform the NI hardware that the space has been freed.

5.7.3 Connection Establishment

If a process running on a node desires to communicate with another process running on a remote node, it needs to find a way to request a new connection. However, in the absence of an already established connection, it is impossible to notify another process to create a new connection. To solve this issue, each node in the system has a special Connection, called System

Connection, which accepts messages from any node in the entire system. This connection has a CID of zero and occupies the first CTE of the CT in each node. This Special Connection is handled by the operating system for protection purposes and is primarily used for sending and receiving messages concerning the establishment of new connections. Thus, user-level software needs to perform a system call to create a new connection with another node.

Establishing a Connection among two or more nodes corresponds to creating a new CTE in the CT of each node involved in the Connection. A rough outline of the process necessary to establish a one-to-one Connection is as follows. Initially, the process desiring to communicate finds a new CTE in the CT of its node. Subsequently a message, containing the CID of the newly found CTE, is sent to the System Connection of the remote node. The remote node also allocates and fills-in a new CTE in its CT and sends a response message with the corresponding CID back to the first node. Upon receiving the response, the first node fills in the remaining fields of its CTE, concluding the connection establishment process. The connection establishment process presented in the example above has been simplified to some extent and does deal with protection issues which are presented in the next subsection.

5.8 Protection & Virtualization

Protection is a very central issue in the proposed NI design and can be broken into two major domains, intranode protection and internode protection. Intranode protection deals with protection issues within a node, which also includes typical NI virtualization issues. Internode protection deals with protection issues among different nodes and is closely related to network security.

5.8.1 Intranode Protection

Intranode protection includes protection among user-level processes that are running on the same node and protection between the user-level processes and the operating system kernel. The main objective is to isolate malicious processes and prevent them from causing harm to other processes or even worse the kernel. Intranode protection greatly depends on the NI virtualization mechanisms, which allow for many processes to efficiently simultaneously communicate through a single NI and share its resources, in the same manner that memory virtualization allows for many processes to efficiently use and share the memory of a system.

Figure 5.12 shows a set of three user-level processes running on a single node. Process 13 is labelled as malicious. Malicious does not necessarily describe a process that is intentionally trying to cause damage to the system. For instance, a process can also be considered malicious due to programming errors. Intranode protection guarantees that a malicious process will not

affect other processes. In particular this means that a malicious process (e.g. process 13 in the figure) should not be able to read or write data that belong to the connections of other processes (e.g. processes 27 & 42 in the figure). Generally malicious processes should not be able to corrupt or tamper with connections of other processes.

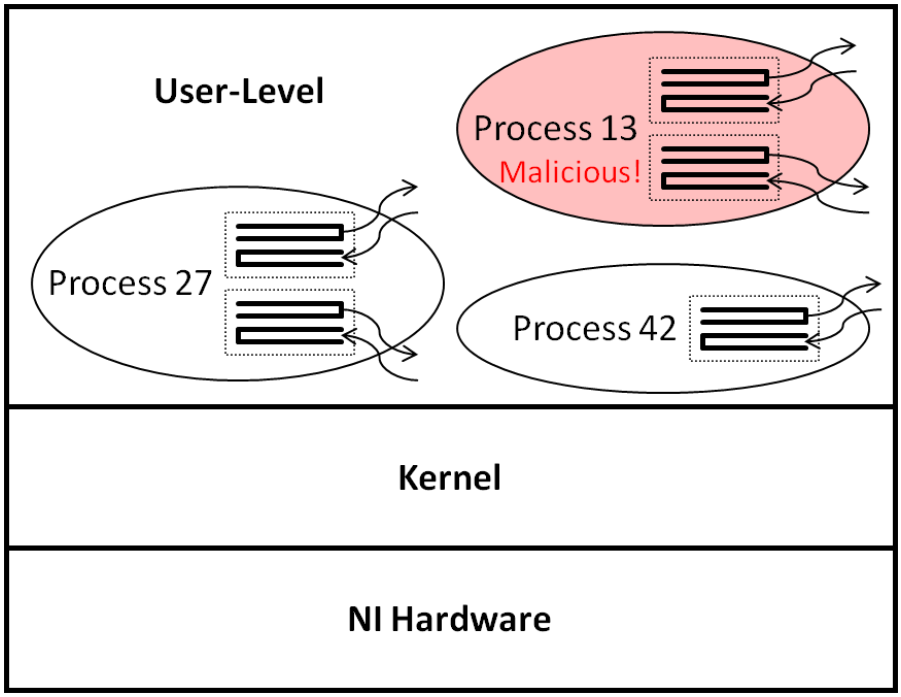

Figure 5.12: Intranode Protection Example.

5.8.2 Internode Protection

As mentioned previously internode protection has to do with protection issues among different nodes. The main issue in internode protection is to deal with compromised nodes. A node is considered as compromised if the kernel or the operating system responsible for the node has been taken over, or is even just behaving erroneously due to a programming error. Internode protection guarantees that compromised nodes are isolated and do not cause problems harm to other nodes.

Figure 5.13 shows a set of five nodes, where node C has been compromised. The first priority is to ensure that nodes with no connections to the compromised node (Nodes B and E in the figure), are not affected. For the nodes that have connections with the compromised node (Nodes A and D in the figure) it is crucial to limit the harm to those specific connections and processes that were communicating with the compromised node. For example in Figure 5.13 node A will possibly receive malicious traffic through the connection it has with the compromised node C, but this should not affect A's communication with B or D.

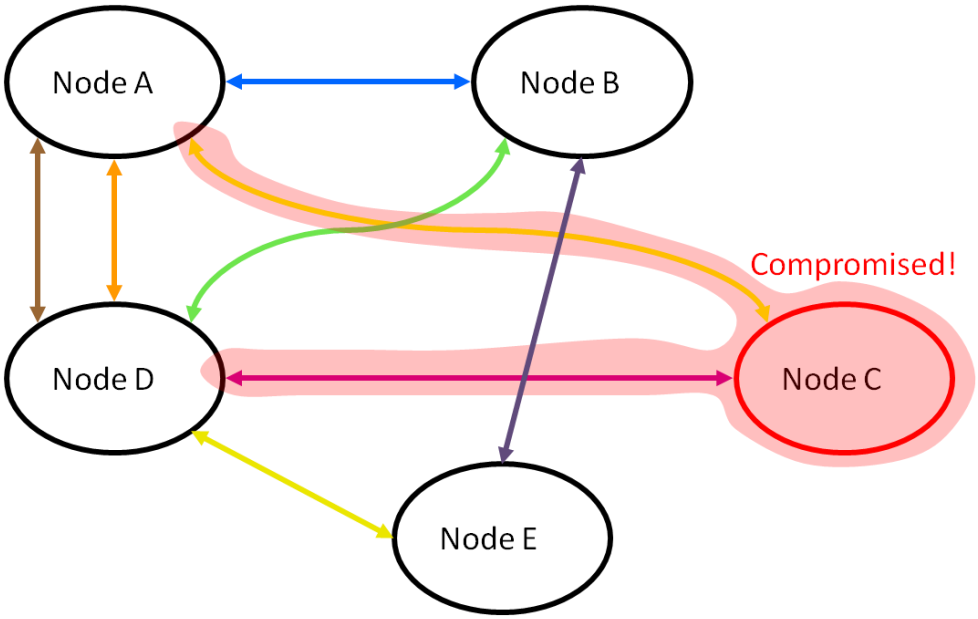

Figure 5.13: Internode Protection Example

5.8.3 Integrating Protection into the proposed NI

Implementing intranode and internode protection requires proper support from the NI hardware and is tightly related to the organization of the Connection Table (CT) and the way it is accessed by the processes and the kernel. In the presented system, each Connection Table Entry (CTE) of the CT is 256 bits wide and is spread across 8 memory banks. To implement protection, the CT is split into 3 vertical protection zones, as seen in Figure 5.14. Protection zones are essentially used to control write access to the various memory banks comprising a CTE.

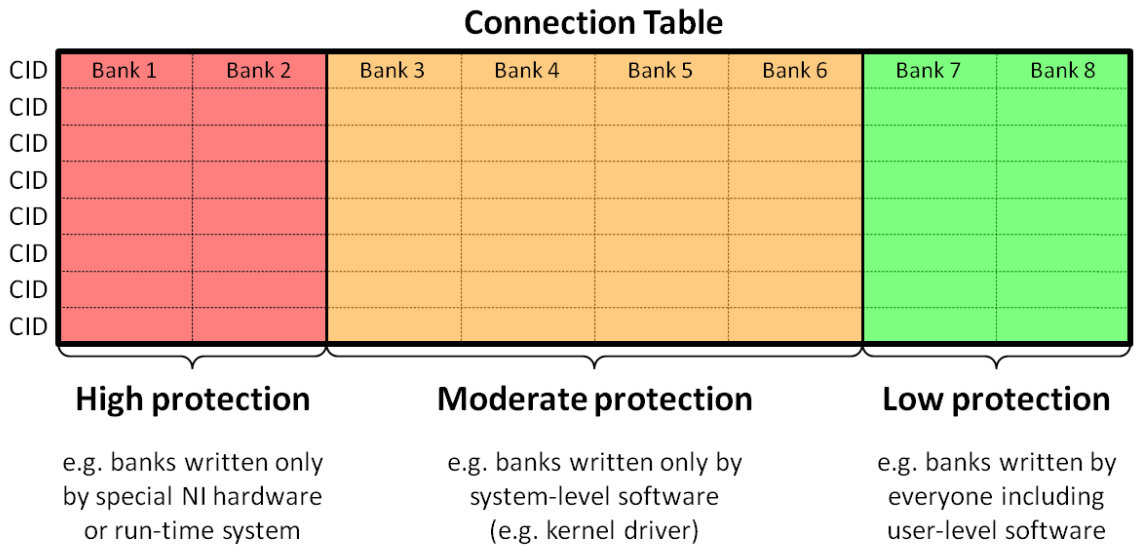

Figure 5.14 Connection Table Protection Zones

This first protection zone includes banks 1 and 2. As explained in subsection 5.5, one-to-one connections use these banks to hold the Node ID and Connection ID for communicating with remote nodes. Many-to-many connections use these banks to store information about the Process Group they belong to. In both cases, NI hardware directly copies the Node ID and Connection ID or the Process Group fields to the headers of the packets that are injected into the network. Thus software does not have control over these fields of the packet header. These banks need to be highly protected, because they control the allowed destinations or process groups for the packets injected into the network. Internode protection is primarily based on restricting access to these banks. If these banks are kept safe, even a compromised node is unable to impersonate another node or generate malicious traffic towards nodes that it doesn't have established a connection with.

The second protection zone contains banks 3, 4, 5 and 6. These banks hold critical connection-specific information that identifies the memory regions that the NI hardware is allowed to access, such as the RDMA base addresses. The main purpose of this protection zone is to prohibit access to the user-level software. For example, if a user-level process could alter the Incoming RDMA physical base address, it would indirectly gain write access to any physical address in the node; even to physical addresses belonging to the kernel. However the harm that can be caused is only local and does not affect other nodes of the system, which means protection for these banks is not as crucial as for banks 1 and 2. In the case something goes wrong, This protection zone is crucial to supporting intranode protection.

The third protection zone concerns banks 7 and 8. These two banks store connection-specific information that may be written by everyone, including user-level software. For example these banks hold the tail pointer for the Outgoing Queue of a connection. As described in subsection 5.7, the tail pointer needs to be written by user-level software to trigger a transfer. This zone has the lowest level of protection and resembles the protection of normal scratchpad memory that has been allocated to a process, which means it is only allowed to access it if it has first acquired a virtual address mapping to it.

The memory of a node can be accessed by the NI and by several layers of software, such as a run-time system, a virtual machine monitor, system-level software and user-level software. To implement protection, each protection zone described above needs to be delegated to either the NI hardware or to some layer of software. To achieve a sufficient level of internode protection, the banks that belong to the first protection zone should only be written by the NI hardware or the lowest level of software (e.g. run-time system or virtual machine monitor) during the connection establishment. This ensures that even a node with a compromised kernel will not affect other nodes of the system.

The banks belonging to the second protection zone should only be written by a "privileged" software layer, such as the NI kernel driver. This zone mainly affects intranode protection, in that it prohibits malicious user-level software from "tricking" the NI hardware and gaining access to memory it does not own. Thus, a user-level process desiring to change the contents of fields that belong to the "moderate" protection zone should perform a system call.

An alternative approach to the above, is to merge the first and second protection zones into a single larger protection zone that contains all six banks from 1 to 6. These banks may only be written by the NI hardware and by privileged software, such as the NI kernel driver. This eliminates the "high" protection zone making the system less secure, but is an attractive solution for systems where high levels of protection cannot be afforded or are not a major concern.

5.8.4 Controlling Access to the Connection Table

Enforcing different access rights to the three protection zones presented above and among different user-level processes, requires special protection mechanisms that allow for the NI to recognize who is making the access. Firstly, a mechanism is required to distinguish accesses made by user-level processes and the OS kernel. Secondly, to provide protection among different processes using the NI, a mechanism to distinguish which process makes the access is also necessary.

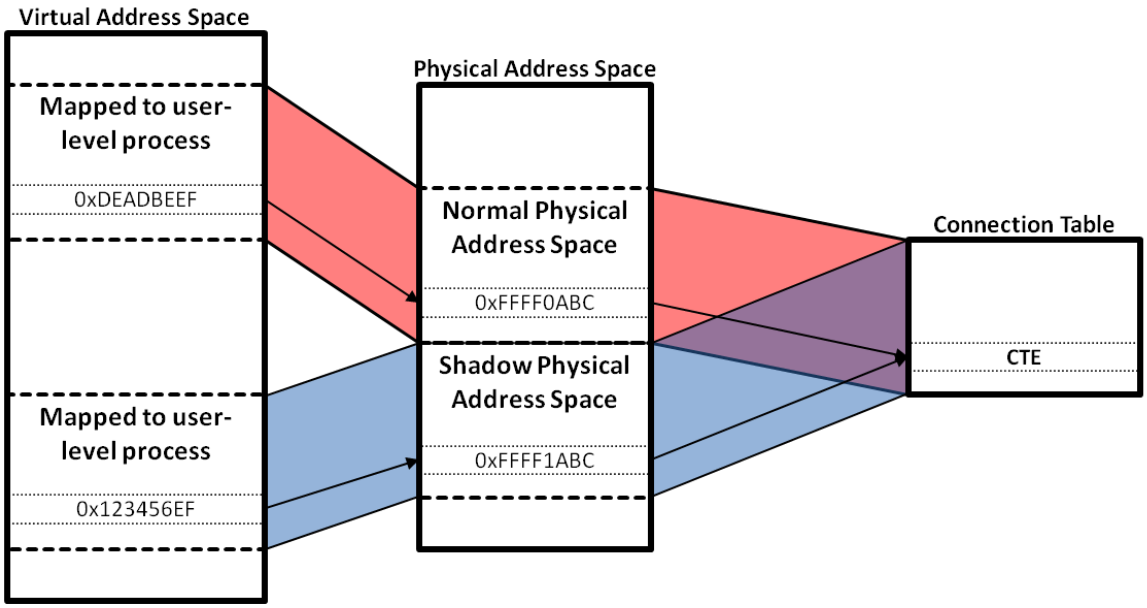

Figure 5.15: Protection between user-level processes and the kernel

Providing different access rights to a user-level process and the kernel can be done through the use of a shadow address space, as shown in Figure 5.15. The user-level process and the kernel driver are given separate virtual address mappings to access the CT. These mappings correspond

to a normal physical address space and a shadow address space, which differ in only one bit (or a small number of bits) and are both mapped to the CT. In this manner the NI can distinguish if an access to the CT is made by a user-level process or the kernel driver and enforce proper protection according to the existing protection zones. This technique doubles the size of the required address space that corresponds to the CT, but this is a minor drawback, since the CT only occupies a tiny part of modern 32-bit or 48-bit bit address spaces.

The CT hosts many connections that belong to different user-level processes, which requires the existence of a fine-grain protection mechanism among processes using the CT. Although traditional virtual memory systems offer sufficient protection at page granularity, the CT requires finer-grain protection. In the proposed system a typical 4Kbyte page can host 128 Connections. Using traditional virtual memory protection, this would force each process to own a minimum of 128 connections, which would either limit the number of processes that communicate or require a very large CT.

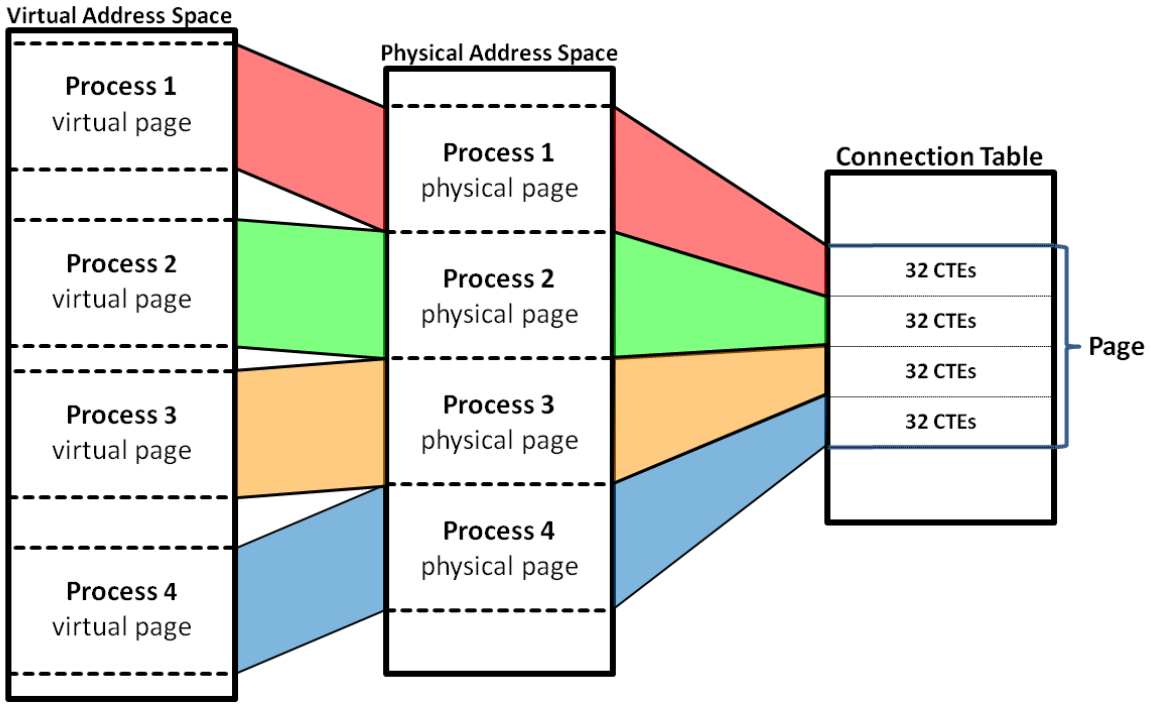

Figure 5.16: Protection among user-level processes sharing a CT page

A technique for sharing a page of the CT (containing 128 CTEs) among many processes is shown in Figure 5.16. Each page of the CT is mapped to more than one physical pages, which are in turn mapped to virtual pages that belong to different user-level processes. The number of physical and virtual pages used to access a single page of the CT is equal to the number of processes sharing that particular page of the CT. This can range from a single page, offering 128

CTEs to a single process, up to 128 pages, offering 1 CTE to 128 processes. The latter case occupies 512Kbytes of physical and virtual address space for every page of the CTE.

To guarantee that a user-level process can only access its own CTEs in the CT, special NI hardware logic is needed to manipulate the physical addresses that reach the CT, as shown in Figure 5.17. Although a whole physical page is mapped to each user-level process, access has to be limited to only a subset of the CTEs in that particular page. To achieve this, for each physical address that reaches the CT, a few of the LSBs of the Page Number are copied and overwrite a few of the MSBs of the page number. This prevents processes from accessing CTEs that belong to another process. The number of bits copied can range from 0 to 7, which corresponds to the number of processes (1-128) sharing a single CT page.

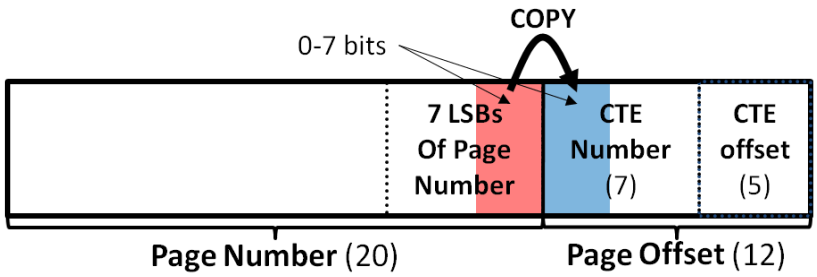

Figure 5.17: NI Physical Address Manipulation

The above physical address manipulation scheme can be combined with the shadow address space technique for protection among user-level processes and the kernel to provide a full protection solution that takes advantage and is built on top and of the standard virtual memory system. A drawback of this scheme is that many processes simultaneously communicating and sharing a single page of the CT will use many Translation Look-aside Buffer (TLB) entries. However this can be alleviated by limiting the number of processes that share each page of the CT, i.e., to 4 or 8 processes with 32 or 16 CTEs each, which also occupies less of the virtual and physical address space. An alternative fine-grain protection solution, which requires additional hardware and OS support has been proposed in [34].

6 Conclusions & Future Work

As computing and embedded systems evolve towards highly parallel multiprocessors, major research and development efforts are being focused on NI architectures that enable efficient interprocessor communication. NI design goals include high performance (i.e. low latency, high bandwidth), scalability, reliability and protection. The work presented in this master thesis includes the development of a NI queue manager for use in computer clusters and a proposed NI design well suited to chip multiprocessors.

The NI queue manager implements Virtual Output Queues (VOQs) and supports Variable-Size Multi-Packet Segments (VSMPS) and QFC flow control. To increase the available network buffer space, the queue manager supports VOQ traffic migration to external memory, stored in the form of memory blocks connected in linked-lists. Free-List Bypass and Free Block Preallocation optimization techniques are employed to minimize the required number of accesses to external DRAM, thus achieving higher bandwidth.

Three versions of the presented queue manager were implemented in real hardware on FPGA-based boards and are used on a daily-basis at the CARV laboratory of ICS, FORTH for interprocessor communication research. The FPGA-based implementation of the design proved the feasibility and effectiveness of variable-size multipacket segmentation [3] and to our knowledge the developed NI is the first one to employ such a segmentation scheme. The FPGA hardware cost results for each individual module, as well as for each version of the queue manager, show that the flow-control related register files and the on-chip buffers are the limiting factors for scalability.

Network performance experiments using the developed queue manager confirmed previous theoretical and simulation results about the behavior and performance of the buffered crossbar switch [6]. Network performance results indicate that the presented queue manager offers satisfactory performance, both in terms of bandwidth and latency. The novel packet processing mechanism employed to convert arbitrary traffic segments into autonomous network packets completely eliminated the need for reassembly, dramatically reducing buffer space and hardware complexity at the receiving end.

The proposed NI design is lightweight and tightly coupled to the processor, making it well suited to future chip multiprocessors. It is customized for implementation in the new prototyping platform of the CARV laboratory, which will be used to replicate and explore the architecture of future chip multiprocessors, offering two powerful communication primitives Message Queues

and Remote DMA. Message Queues are intended for low latency communication, mainly synchronization and control messages or small low-overhead data transfers. Remote DMA minimizes processor involvement in communication, is well suited for bulky data transfers and facilitates zero-copy protocols.

The proposed NI is based on the notion of connections, which support one-to-one and many-to-many communication patterns among processing nodes. From a hardware perspective connections contain and organize all of the required state for the ongoing communication. From a software perspective connections are a powerful and versatile mechanism and form the interface for sending messages and initiating remote DMA operations. Furthermore the proposed NI supports a versatile protection and security solution, based on the existence of protection zones, that can easily be adapted to the specific security requirements of a system.

NI design for chip multiprocessors is an emerging field and there is plenty of future work to be conducted. Open issues include NI support for migration of processes from one node to another and proper support for virtualization to enable the presence of multiple operating systems on a single node. Flow control poses another important issue that needs attention and requires proper support from the NI. Finally, in the case of shared memory systems, it is also important to investigate NI mechanisms for efficiently supporting cache coherence.

7 Bibliography

[1]. *Scalable Intelligent Video Server System.* [Online] http://www.sivss.org.

[2]. *Scalable Computer Architecture.* [Online] http://www.sarc-ip.org.

[3]. *Variable-size multipacket segments in buffered crossbar (CICQ) architectures.* **Katevenis, M and Passas, G.** 2005. Communications, 2005. ICC 2005. 2005 IEEE International Conference on. Vol. 2, pp. 999-1004.

[4]. *Quantum Flow Control (QFC).* [Online] http://archvlsi.ics.forth.gr/~kateveni/534/qfc/.

[5]. *Input Versus Output Queueing on a Space-Division Packet Switch.* **Karol, M, Hluchyj, M and Morgan, S.** 12, dec 1987, Communications, IEEE Transactions on [legacy, pre - 1988], Vol. 35, pp. 1347-1356.

[6]. *Variable packet size buffered crossbar (CICQ) switches.* **Katevenis, M, et al.** 2004. Communications, 2004 IEEE International Conference on. Vol. 2, pp. 1090-1096.

[7]. *Performance Evaluation of Packet-to-Cell Segmentation Schemes in Input Buffered Packet Switches.* **Christensen, Kenneth J, et al.** 2004, CoRR, Vol. cs.NI/0403030.

[8]. *A parallel-polled virtual output queued switch with a buffered crossbar.* **Yoshigoe, K and Christensen, K J.** Dallas, TX : s.n., 2001. High Performance Switching and Routing, 2001 IEEE Workshop on. pp. 271-275.

[9]. *Efficient per-flow queueing in DRAM at OC-192 line rate usingout-of-order execution techniques.* **Nikologiannis, A and Katevenis, M.** Helsinki, Finland : s.n., 2001. Communications, 2001. ICC 2001. IEEE International Conference on. Vol. 7, pp. 2048-2052.

[10]. *A VLSI architecture for an 80 Gb/s ATM switch core.* **Andersson, P and Svensson, C.** Austin, TX, USA : s.n., 1996. Innovative Systems in Silicon, 1996. Proceedings., Eighth Annual IEEE International Conference on. pp. 9-15.

[11]. *Efficient fair queueing using deficit round robin.* **Shreedhar, M and Varghese, George.** 4, New York, NY, USA : ACM Press, 1995, SIGCOMM Comput. Commun. Rev., Vol. 25, pp. 231-242.

[12]. **Shanley, Tom.** *PCI-X System Architecture with CD.* [ed.] Karen Gettman. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2000.

[13]. Rocket I/O User Guide. *Xilinx Inc.* [Online] http://www.xilinx.com/bvdocs/userguides/ug024.pdf.

[14]. The DiNI Group. *DN6000k10SC.* [Online] http://www.dinigroup.com/DN6000k10SC.php.

[15]. Virtex-II Pro FPGAs. *Xilinx Inc.* [Online] http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_pro_fpgas/.

[16]. **Passas, G.** *Performance Evaluation of Variable Packet Size Buffered Crossbar Switches.* s.l. : Institute of Computer Science, FORTH, Heraklion, Greece, 2003.

[17]. *Evaluating the performance of four snooping cache coherency protocols.* **Eggers, S J and Katz, R H.** 3, New York, NY, USA : ACM Press, 1989, SIGARCH Comput. Archit. News, Vol. 17, pp. 2-15.

[18]. *An evaluation of directory schemes for cache coherence.* **Agarwal, Anant, et al.** New York, NY, USA : ACM Press, 1998. ISCA '98: 25 years of the international symposia on Computer architecture (selected papers). pp. 353-362.

[19]. **Mukherjee, Shubhendu S and Hill, Mark D.** *A Survey of User-Level Network Interfaces for System Area Networks.* 1997.

[20]. *Overview of the Blue Gene/L system architecture.* **Gara, Alan, et al.** 2-3, 2005, IBM Journal of Research and Development, Vol. 49, pp. 195-212.

[21]. *An analysis of TCP processing overhead.* **Clark, D D, Romkey, J and Salwen, H.** Minneapolis, MN, USA : s.n., 1988. Local Computer Networks, 1988., Proceedings of the 13th Conference on. pp. 284-291.

[22]. **Inc., Sun Microsystems.** SPARC MBus Interface Specification. April 1991.

[23]. *Cavallino: The teraflops router and NIC.* **Joseph Carbonaro, Frank Verhoom.** 1996, Hot Interconnects IV, pp. 157-160.

[24]. *Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor.* **Galles, Mike and Williams, Eric.** 1994. HICSS (1). pp. 134-143.

[25]. Intel® 82547EI Gigabit Ethernet Controller . *Intel Inc.* [Online] http://www.intel.com/design/network/products/lan/controllers/82547ei.htm.

[26]. *The MIT Alewife machine: architecture and performance.* **Agarwal, Anant, et al.** New York, NY, USA : ACM Press, 1998. ISCA '98: 25 years of the international symposia on Computer architecture (selected papers). pp. 509-520.

[27]. *START-NG: Delivering Seamless Parallel Computing.* **Chiou, Derek, et al.** London, UK : Springer-Verlag, 1995. Euro-Par '95: Proceedings of the First International Euro-Par Conference on Parallel Processing. pp. 101-116.

[28]. *The J-Machine: A Fine-Gain Concurrent Computer.* **Dally, William J, et al.** 1989. IFIP Congress. pp. 1147-1153.

[29]. **Fillo, Marco, et al.** *The M-Machine Multicomputer.* 1995.

[30]. *The interaction of architecture and operating system design.* **Anderson, Thomas E, et al.** New York, NY, USA : ACM Press, 1991. ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems. pp. 108-120.

[31]. *Myrinet: A Gigabit-per-Second Local Area Network.* **Boden, Nanette J, et al.** 1, 1995, IEEE Micro, Vol. 15, pp. 29-36.

[32]. *U-Net: a user-level network interface for parallel and distributed computing.* **Eicken, Thorsten von, et al.** 1995. Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP). Vol. 29, pp. 303-316.

[33]. *A comparison of architectural support for messaging in the TMC CM-5 and the Cray T3D.* **Karamcheti, Vijay and Chien, Andrew A.** New York, NY, USA : ACM Press, 1995. ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture. pp. 298-307.

[34]. *Mondrian Memory Protection.* **Witchel, E, Cates, J and Asanović, Krste.** 2002. Proceedings of ASPLOS-X.

[35]. *Direct Cache Access for High Bandwidth Network I/O.* **Huggahalli, Ram, Iyer, Ravi and Tetrick, Scott.** 2, New York, NY, USA : ACM Press, 2005, SIGARCH Comput. Archit. News, Vol. 33, pp. 50-59.

[36]. *Protected, User-level DMA for the SHRIMP Network Interface.* **Blumrich, M A, et al.** 1996. Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2).

[37]. *Synchronization and communication in the T3E multiprocessor.* **Scott, Steven L.** New York, NY, USA : ACM Press, 1996. ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems. pp. 26-36.

[38]. *Integration of message passing and shared memory in the Stanford FLASH multiprocessor.* **Heinlein, John, et al.** New York, NY, USA : ACM Press, 1994. ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems. pp. 38-50.

[39]. **Osborne, R, et al.** *DART –- A Low Overhead ATM Network Interface Chip.* 1996.

[40]. *Coherent Network Interfaces for fine-Grain Communication.* **Mukherjee, S S, et al.** 1996. Proc. of the 23rd Annual Int'l Symp. on Computer Architecture (ISCA'96). pp. 247-258.

[41]. **Hennessy, John L and Patterson, David A.** *Computer Architecture : A Quantitative Approach; second edition.* s.l. : Morgan Kaufmann, 1996.

[42]. **Berrendorf, Rudolf, et al.** *Intel Paragon XP/S - Architecture, Software Environment, and Performance.* 1994.

[43]. Xilinx XUP Virtex II Pro Development System. *Xilinx Inc.* [Online] http://www.xilinx.com/univ/xupv2p.html.

[44]. **Stewart, Lawrence C. and Gingold, David.** A New Generation of Cluster Interconnect. *SiCortex.* [Online] December 2006. www.sicortex.com/whitepapers/sicortex-cluster_interconnect.pdf.

[45]. *The SPLASH-2 programs: characterization and methodological considerations.* **Woo, Steven Cameron, et al.** 2, New York, NY, USA : ACM Press, 1995, SIGARCH Comput. Archit. News, Vol. 23, pp. 24-36.