Effective Parallel Computation on Workstation Cluster with a User-level Communication Network

by

James C. Hoe

B.S. University of California at Berkeley
(1992)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of
the Requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

February, 1994

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science
January 15, 1994

Certified by [illegible]
Gregory M. Papadopoulos
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by [illegible]
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Effective Parallel Computation on Workstation Cluster with a User-level Communication Network

by

James C. Hoe

Submitted to the Department of Electrical Engineering and Computer Science
on January 15, 1994
in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Leveraging the engineering effort in the microprocessor sector, massively parallel processing (MPP) systems have emerged as the leading supercomputing architecture in terms of price and performance. By using commercial workstations as processing nodes to further reduce engineering costs, workstation-based MPP systems promise to deliver truly affordable supercomputing performance. However, existing workstation-based parallel systems are confined to coarse-grained parallelization because of the large overhead of interprocessor communication over existing local area networks. To address this problem, this thesis proposes to augment a conventional LAN-connected workstation cluster with a Fast User-level Network (FUNet). Based on MIT's Arctic technology, FUNet provides a packet-switched routing network with an acknowledgment/retry end-to-end flow control protocol. A hardware Fast User-level Network Interface (FUNi) provides access to FUNet for both message passing and remote direct-memory-access (DMA) block transfer between parallel peer processes. The FUNi hardware mechanisms allow direct low-overhead user-level access to FUNet while maintaining secure and transparent sharing of FUNet among multiple parallel applications. FUNi can be realized as SBus peripheral cards to allow compatibility with a variety of workstation platforms. It takes advantage of SBus's Direct Virtual Memory Access (DVMA) to circumvent performance limitations imposed by existing workstation and microprocessor designs. Simulation results have shown that FUNet with FUNi, when coupled with latency-hiding software techniques, is effective in supporting fine-grained parallel processing on a workstation cluster.

Thesis Supervisor: Gregory M. Papadopoulos

Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I would like to thank Professor Gregory Papadopoulos for advising this thesis. His guidance and encouragement during the past year and a half have been invaluable to the completion of this thesis. His insights, comments, criticisms, and suggestions added greatly to the final result. Thanks to Dr. Andy Boughton for explaining the design of Arctic and helping me with this thesis.

Many thanks to all the undergraduate students who have contributed to this thesis. Steve Chamberlin investigated the possibility of a FPGA implementation of custom network routers. Noah Rosen examined the different options in network link technology. Patrick Chan, as part of the Undergraduate Research Opportunity Program, has been developing the software for the FUNet simulator.

Thanks especially to my officemate, Derek Chiou, who has been a great friend through it all. He is the one person responsible for helping me overlook the frigid winters and humid summers and convincing me to join this wonderful institution. Thanks to all the others at the Computation Structures Group (CSG) who helped in various ways during the course of the project. A special thanks to Professor David Culler at U.C. Berkeley for introducing me to these wonderful people of CSG.

Thanks to the Largescale Parallel Software Group for providing the PROTEUS simulator. Thanks to Professor David Culler's research group at U.C. Berkeley for supplying the CMAM library.

My thanks to the Office of Naval Research for funding my graduate study. The fellowship has been a tremendous advantage.

As always, I am indebted to my parents for their love, support, and encouragement through the years. I am grateful for their hard work and sacrifices that have brought me to where I am today.

And I would like to thank Eva for her unconditional support in the last year and half. She encourages me when I am down, understands me when I am upset, and tolerates all of my many quirks.

Finally, I would like to thank Eva Chan, Janey Hoe, Kevin Lew, and Patrick Hu for their useful comments on the final drafts of this thesis.

— *To my parents*

Contents

List of Figures

Chapter 1

Introduction

This thesis describes the design and evaluation of a workstation-based parallel system enabled with FUNet, a low-overhead, user-level interworkstation communication network. This chapter first presents the motivation behind this thesis by examining existing parallel systems. Next, an overview of FUNet, the focus of this thesis, is presented. The organization of this thesis is outlined at the end of this chapter to direct readers to their areas of special interest.

1.1 Motivation for This Thesis

In the past two decades, the performance of microprocessors has increased by three orders of magnitude because of advances in integrated circuits and related technologies [11]. At the same time, the prices of microprocessors have remained affordable to the mass market. Given this performance-to-price advantage, microprocessor-based PC's and workstations have gained tremendous popularity in both industry and academia. Such widespread usage, in turn, has provided a strong positive feedback to the developments of each new generation of higher performing microprocessors and microprocessor-based machines.

In recent years, high-performance commercial microprocessors have also begun to play a major role in the supercomputing arena. By incorporating readily available commercial microprocessors, MPP manufacturers, such as Thinking Machines and Cray, have been able to produce higher performing systems in a shortened design cycle. However, despite the reduction in the design effort, the price of MPP systems remains far from affordable because of the engineering costs of proprietary hardware that still surrounds the microprocessors. A typical MPP processing node, resembling a stripped-down workstation, is priced two to three times more than a full-featured workstation of comparable performance [15, 14]. This provides a clear incentive to produce MPP systems with commercial workstation hardware as processing nodes.

More recently, both IBM and Hewlett-Packard have offered parallel systems based on a Local Area Network (LAN) cluster of workstations [12, 17]. Parallel programming tools, such as Network Linda [18] and Parallel Virtual Machine PVM [1], also exist for developing parallel applications for a LAN workstation cluster. These simple LAN-based solutions offer speedup in one of two ways. The simpler way is to execute, in batches, multiple independent scalar programs simultaneously. The second method, a more true form of parallel processing, divides a single large task among multiple workstations that execute in parallel. However, the size and granularity of parallelism in the second case is heavily restricted by the prohibitive cost of interprocessor communication over the LAN [12]. In order for workstation-based parallel systems to rival existing MPP architectures in speedup and performance, we need to provide a means for low-overhead interworkstation communication.

1.2 FUNet Parallel Cluster

This thesis proposes to augment a conventional LAN-connected workstation cluster with a second, high-performance user-level network (see Figure 1.1) dedicated to support interworkstation communication between parallel peer processes. The power of Fast User-level Network (FUNet) comes from its greatly reduced interprocessor communication overhead when compared with a LAN. By making assumptions about parallel-processing communication, the network features that have burdened inter-

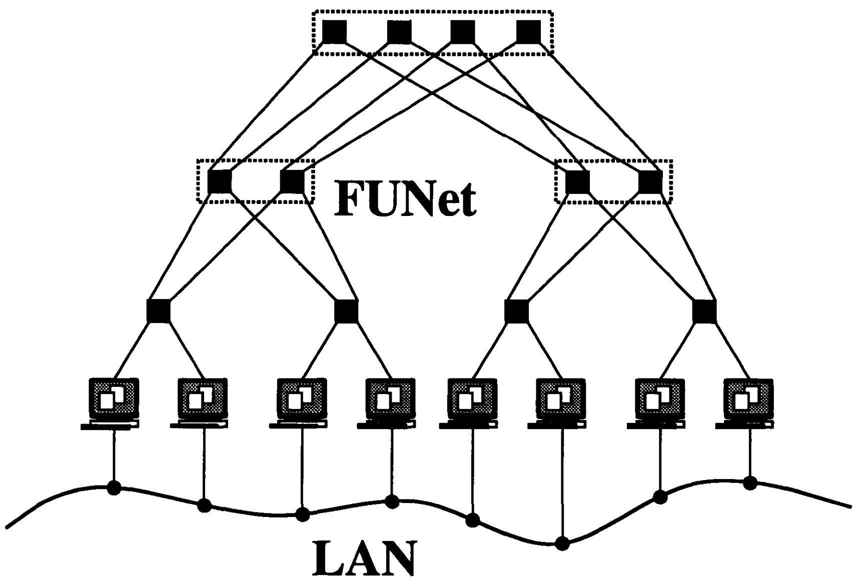

Figure 1.1: A FUNet Cluster with a Fat-Tree FUNet

processor communication in LAN-based parallel systems are left out in the FUNet design. Furthermore, the remaining features are optimized for parallel-processing specific usage. In this section, we present an overview of the FUNet system. We begin by establishing the parallel execution model that is supported by a FUNet cluster. Next we highlight the interesting features of FUNet, paying particular attention to the hardware Fast User-level Network Interface (FUNi).

1.2.1 Model of Execution

Each processing node of the FUNet cluster is a stand-alone workstation controlled by its own operating system. Sequential applications execute normally on individual workstations in a time-sharing environment. When a parallel application starts, a new time-shared process is created on each participating workstation. Identical copies of the executable binary are loaded into the virtual memory space of the started processes, and the peer processes begin independent execution of the binary at the same entry point.

The programming model for parallel execution on a FUNet cluster is multiple-instruction-stream/multiple-data-stream (MIMD) message-passing. Interworkstation communication between peer processes is provided by FUNet through FUNi. FUNet provides a reliable but unordered message delivery. Sending and receiving messages are under explicit control of the user program. Peer processes are named by integral node identifiers from 0 to N-1 where N is the number of participating workstations. During communication, a process can reference an object on a remote workstation by the same virtual address of the corresponding object in the local virtual address space since every process has the same program image.

Multiple parallel and sequential applications share both the network and processors. However, this sharing is transparent to the user-level processes, and protection mechanisms prevent interference and security violation among different applications. Each application is presented with the simplifying illusion that it is the sole user of the resources.

1.2.2 FUNet

FUNet is a packet-switched routing network based on the Arctic (A Routing Chip that is Cool) 4-by-4 packet-switched router chip [2]. The proposed FUNet will be implemented on a centralized network hub. Each FUNi will be given two connections (one in each direction) to an Arctic router. Each connection is 16-bit wide plus three bits of point-to-point flow control signals. The network will be clocked at 25 MHz, allowing up to 800 Mbit/sec of peak transfer bandwidth per channel.

An acknowledgment/retry end-to-end flow control protocol is implemented over the two-level FUNet. The protocol is conducted by the FUNi hardware and is not visible to the user-level processes. Under this protocol, when a data packet arrives at its destination, the receiving FUNi can accept the packet by returning a positive acknowledgment packet to the sender, or the interface can return a negative acknowledgment to reject the packet. Acknowledgment packets are transferred with a network priority higher than data packets to prevent deadlock. After receiving a negative acknowledgment, the FUNi that originated the rejected packet will automatically arrange for a retransmission of the undelivered packet. The option for rejecting a packet relieves the receiving interface from the burden of buffering all inbound packets. This allows FUNi to continuously absorb packets from the network to reduce network congestion. Negative acknowledgments also serve as an automatic rate control measure to keep fast-sending processes from swamping other processes with packets. Please see Section 2.8 for more information about the acknowledgment/retry protocol.

FUNet uses hardware mechanisms to ensure integrity of the shared system while exposing performance-critical hardware sections to direct user-level access. Privileged control registers on FUNi are protected from the user-level processes through address-translation schemes. Data protection of network packets is enforced through tagging. The operating system assigns each executing parallel application a unique Group Identifier (GID) that is global to the parallel cluster. Each packet entering the network is automatically tagged with the sending process's GID. Prior to delivering a packet, FUNi will check the GID of the executing process against the GID tag of the packet to prevent misdeliveries. Section 4.1.1 provides more detail about protection issues in the shared FUNet environment.

1.3 FUNi

The key design goal of FUNi is to minimize – within the design space allowed by existing workstation hardware – the overhead cost of sending and receiving messages between cooperating peer processes on different workstations.

FUNi will be implemented as peripheral cards for the SBus [9] on workstations. (Please see Figure 1.2.) Conforming to the SBus specification allows FUNi to be compatible with SBus-equipped commercial workstation platforms. The custom logic on the FUNi SBus card will be realized using Xilinx Field Programmable Gate Arrays

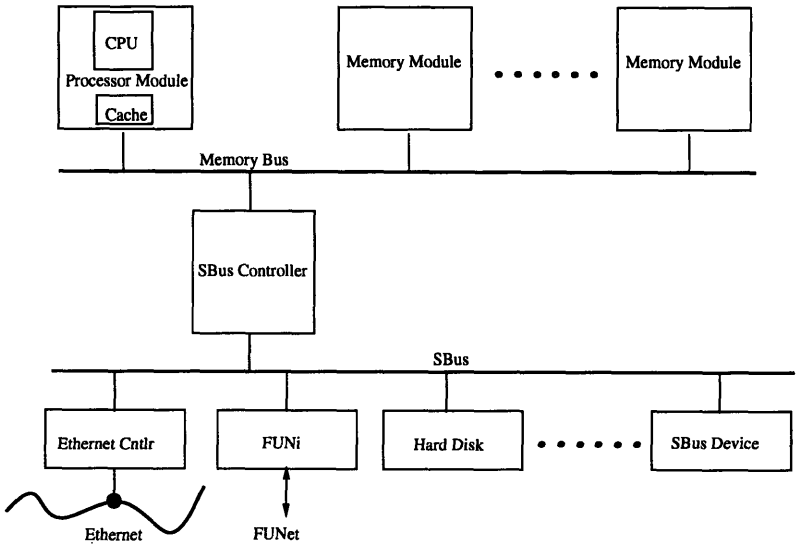

Figure 1.2: A Typical FUNet Processing Node

(FPGA) [24]. The reprogrammability of the FPGA firmware will allow rapid revisioning of the FUNi hardware during the hardware development and future studies. An architectural overview of the FUNi SBus card is presented in Chapter 5. Section 7.2.1 provides further detail regarding the proposed implementation of the SBus card.

To achieve the goal of minimizing communication overhead, user processes are given direct control of FUNi when possible. User-level processes directly invoke FUNi to send and receive packets in a message-passing style of communication. FUNi also provides a facility for a DMA-style virtual-memory-to-virtual-memory block transfer between workstations. The length of message-passing packets can vary from 0 to 21 32-bit words. (Memory-to-memory data transfers can only occur in burst sizes varying from 0 to 16 words in increments of 4 words.) Aside from allowing 512 user-defined packet types, the network interface also supports two packet priorities: reply and request, for constructing deadlock-free communication protocols in user programs. (FUNi's sending and receiving mechanisms always give precedence to packets with reply priority.) All packet types and priorities are available in both message-passing communications and DMA transfers.

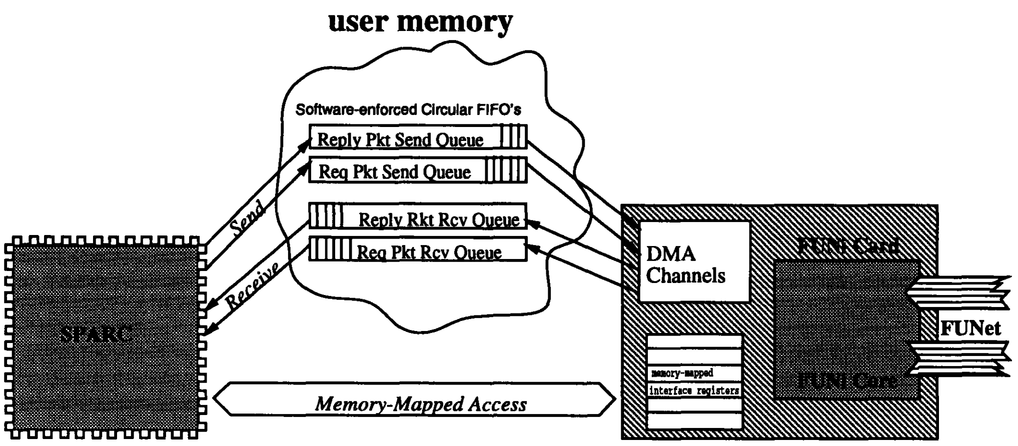

Figure 1.3: FUNi Programming Interface

User- and system-level processes control the operation of FUNi by reading and writing to FUNi's internal control registers through memory-mapped accesses. The sending and receiving interface is based on four software-enforced circular FIFO (first-in-first-out) packet queues jointly maintained by the user program and FUNi. (Please see Figure 1.3 for an illustration of FUNi's user interface.) The queues are allocated by the user process within the user's virtual memory space, and the size of the queue can be adjusted by the user process. When sending, the sending process writes the content of the outbound packet to the head of the send queue. FUNi uses SBus's Direct Virtual Memory Access (DVMA) [9] to retrieve outbound packets from the tail of the send queues in FIFO-order. Analogously, FUNi uses DVMA to deliver inbound packets into the receive queues. The user processes can then receive the inbound packets by reading from the tail of the receive queues. The full FUNi programming interface is discussed in detail in Chapter 3 with examples on its usage.

1.4 Summary

Key features of FUNet are summarized as follows:

1. Allows low overhead user-level interworkstation communication

2. Allows DMA-style virtual-memory-to-virtual-memory block data transfer between workstations
3. Requires no modification to existing workstation hardware
4. Supports large, variable length user packets of 0 to 21 32-bit words
5. Supports 512 user packet types and 2 packet priorities
6. Large, dynamically sizable send and receive queues in the user memory
7. Allows secure time-sharing of multiple applications
8. Implements hardware acknowledgment/retry end-to-end flow control protocols to reduce network congestion

1.5 Organization of This Thesis

In this chapter, we have presented the motivation behind FUNet, a dedicated user-level network for parallel processing in a workstation cluster. To provide a frame of reference for the work in this thesis, an overview of FUNet was presented. The remainder of the thesis presents the design, implementation, and evaluation of FUNet and FUNi. Chapter 2 describes the steering forces in the design of FUNet and FUNi. In Chapter 3, we present the user programming interface of FUNi and demonstrate its usage in interworkstation communication. In Chapter 4, we describe FUNet's underlying mechanism for supporting time-sharing of multiple parallel applications and the operating-system specific FUNi programming interface. Gang scheduling issues are also briefly discussed in this chapter. Chapter 5 gives an architectural overview of FUNi's hardware design. Chapter 6 evaluates the performance of a FUNet parallel cluster by comparing a simulated FUNet cluster with a contemporary massively parallel computer, CM-5. The thesis concludes with Chapter 7 in which observations made during the course of this thesis are presented. Related work in the field and future work extending from this thesis are also mentioned in the conclusion.

Chapter 2

Design Considerations of FUNet and FUNi

This chapter describes the various considerations that have influenced the design of FUNet and FUNi. It provides a high-level rationale for the design choices made for FUNet and FUNi. A more detailed examination of FUNi's programming interface and architecture are presented in the next three chapters. This chapter begins with a Local Area Network (LAN) and the UNIX Interprocess Communication (IPC) interface as the basis of our investigation on interworkstation communication. As this chapter progresses, different criteria are brought to attention, and the communication network and interface are refined step-by-step. Ultimately, we arrive at the final design of FUNet and FUNi. In Sections 2.2 and 2.3, we describe the design steps leading to a dedicated user-level network. In Sections 2.4, 2.5, 2.6 and 2.7, we discuss how the design of the network interface evolved. Finally in Sections 2.8, 2.9 and 2.10, we describe the acknowledgment/retry end-to-end flow control protocol and its implementation on FUNet.

2.1 Starting Point: LAN and UNIX IPC

The UNIX IPC interface offers an existing means for interprocessor communication among networked machines. IPC provides a system-call software interface that layers above TCP/IP, UDP/IP or raw IP protocol for communication over the physical network [7]. Thus, a LAN cluster of workstations already has the capability of a rudimentary parallel system in which peer processes of a parallel application can communicate within the LAN by relying on the existing software interface alone.

2.2 Dedicated Network

The generalized IPC interfaces and the underlying protocols were designed for general communication over the nationwide Internet system. The high-level software interface needs to hide all the implementation details of the physical interconnects which could be arbitrarily complicated. The interface must also implement the necessary security barriers since nothing can be assumed about the reliability of the participants on the Internet. When this same interface, with its full generality, is used for communication within a LAN cluster, interworkstation communication overhead also becomes unnecessarily burdened with the mechanism that deals with communication beyond the locality of the LAN cluster.

Whereas communication latency can be tolerated by performing useful computations during a communication delay, the communication overhead, consuming real processor cycles, cannot be similarly overlapped. If the overhead is large, the loss of computation cycles to communication overhead will quickly overwhelm any benefits from parallel processing. The overhead of interprocessor communication over a LAN through a standard UNIX IPC interface, which requires a few thousand cycles [6], prevents the possibility of fine-grained parallel processing.

However, this large overhead is not inherent in intracluster, interworkstation communication for parallel processing. For parallel processing within a cluster, we only need to communicate between a relatively small number of physically nearby workstations. Addressing and routing are much simpler issues, and stricter assumptions can be made about the reliability of the participating software and hardware. The

IPC interface over IP protocols provides a profuse amount of abstraction and generality that we cannot make use of at the cost of unwanted communication overhead. However, since many crucial services on UNIX workstations depend strongly on IPC and IP for secure network connectivity, we will endanger the system's integrity if we bypass the established interface and protocol. However, by implementing a dedicated and separate network, we could move away from the confines of established protocols and tailor an interface to the sole needs of parallel processing.

2.3 Direct User-Level Hardware Interface

In a typical multi-user, time-shared workstation environment, the network interface hardware is a critical and shared resource. Allowing any user-level process direct access to the network hardware would be disastrous to the system's integrity. In existing UNIX systems, network access is a privileged kernel-level operation; the user can only indirectly access the network by requesting kernel-level intervention through IPC system calls. This protection mechanism, requiring a software trap to the kernel plus whatever security measures implemented, adds significant overhead to the user-level interprocessor communication.

Taking advantage of the design freedom offered by our dedicated network, we can eliminate this software overhead by moving the necessary protections into hardware. By implementing a correct set of protection at the hardware level, we can safely let multiple time-shared user-level processes directly access the appropriate parts of the network interface hardware, and yet maintain the security and integrity between processes. By eliminating the kernel intervention for interprocessor communication, the overhead of interprocessor communication is reduced.

Another enabling factor of the user-level hardware network interface is the greatly simplified network interface. Our simple and specialized network does not require complicated addressing or a reliability protocol. A packet-based message-passing communication interface is not only simple enough to be easily implemented in hardware, but more importantly, also simple enough that we can trust the user-level process to invoke the interface correctly. There is no need to wrap up the already simple-minded interface with extra layers of system software procedure calls to provide a further simplified abstraction.

2.4 Memory-Mapped Register Interface

However, depending on the implementation, a hardware interface can still incur notable amounts of overhead. Ideally, one would like the interprocessor communication to incur zero overhead. With a user-level hardware interface, it is almost possible to achieve that. Joerg and Henry [13] proposed a network interface with interface registers mapped directly into the general purpose registers (GPR) of a RISC microprocessor. These interface registers, as part of the GPR file, can be used as the usual source and destination registers in an instruction. However, by issuing a special network access instruction, the contents of these registers become the contents of an outbound network packet. An efficiently coded program can cleverly manage register usage such that the contents of an outbound packet are placed into the appropriate registers as part of the computation. Conversely, the contents of an inbound packet can be loaded into these registers with a single instruction. Once received in the GPR, the content of the packet can be used immediately by subsequent instructions. Thus, the absolute overhead for sending or receiving a network packet is only a single instruction.

This idealized design, though efficient, is simply not available to us. The tightly coupled GPR-mapped network interface simply cannot be achieved without substantial modification to the microprocessor design at a fundamental level. This type of customization is against the goal of this thesis to produce a low-cost alternative MPP system by using commercial hardware where possible. None of today's commercially successful microprocessors provides any special hardware support for interprocessor communication. The current generation of RISC processors with aggressively super-

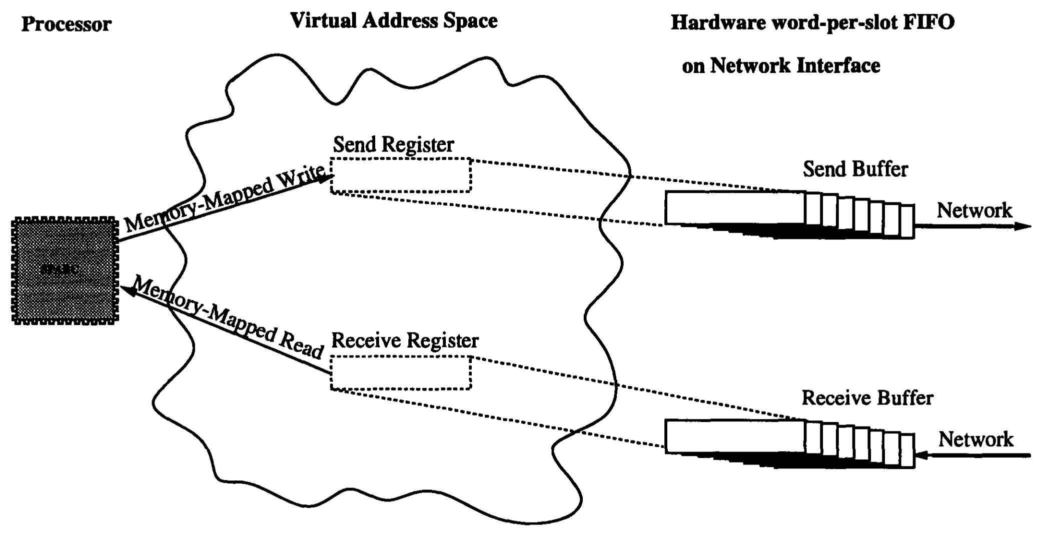

Figure 2.1: A Message Interface based on Memory-Mapped Registers

scalar and pipelined design usually does not even provide a tightly coupled coprocessor interface for attaching a network interface unit. Constrained by what is available in the mainstream microprocessor market, we are left with the option of implementing our network interface at the bus-level with memory-mapped interface registers.

A memory-mapped register interface will introduce a modest overhead, on the order of few tens of cycles, over the tightly coupled GPR-mapped interface. Given the non-negligible penalty of a memory-mapped register access, one needs to design the interface carefully to minimize memory-mapped accesses.

In the simplest interfacing scheme, such as one implemented for CM-5, one would use a single memory-mapped register for sending and another memory-mapped register for receiving [20]. (Please see Figure 2.1.) The single register implementation presents a FIFO abstraction. Sending is achieved by storing the packet header plus the contents of the packet, in packet order, to the same memory-mapped register, much like pushing into a queue. Similarly, receiving is accomplished by repeatedly reading the same memory-mapped register, much like popping from a queue. In this single register FIFO scheme, the contents of the outbound packet must be presented

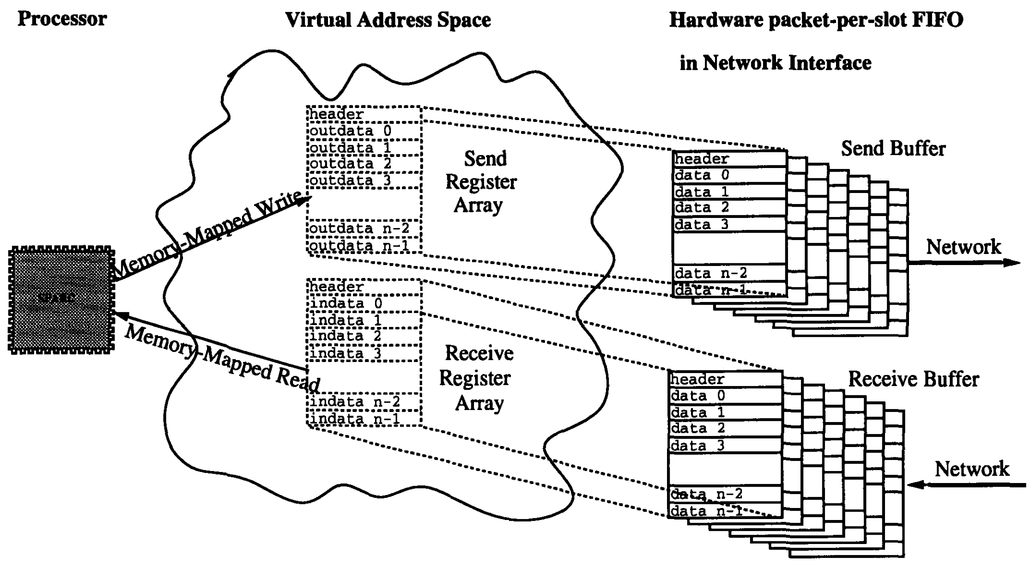

Figure 2.2: A Message Interface based on Memory-Mapped Register Arrays

to the network interface in strict order, and the contents of the inbound packet must be retrieved in order. This constraint on access ordering can impose extra overhead while handling the network packets.

The message interface could also be implemented as an array of memory-mapped registers similar to the alternative design suggested by Joerg and Henry [13]. (Please see Figure 2.2.) The set of memory-mapped registers is logically arranged as a small integer array in the memory-mapped address space. An outbound packet is composed by writing the packet header to the first word of the array and then the entire contents of the packet to subsequent registers. No particular ordering of writes needs to be enforced. The contents of the register are formatted as a packet when a launch command is issued by the processor. Conversely, user processes can receive an inbound packet into a register array by issuing a receive command. The separation of the send and receive register arrays allows task of sending to be interleaved with the task of receiving.

2.5 Memory-based Message Interface

In a message-passing model in which inbound messages are received by an explicit action of the user program, the user cannot guarantee to faithfully absorb the influx of inbound messages at all times. When polling is used, a process must neglect the pending inbound packets between polls. On a reliable network, since packets are never lost, the network must provide some finite buffering to hold these pending packets. The network interface must satisfy most of these buffering needs. Buffering allows the network interface of a negligent user process to continue, at least temporarily, to remove packets from the network and release the network resource occupied by the packets. Without this buffering, the user processes must poll and service inbound packets frequently to prevent pending packets from degrading the network performance by occupying the more critical network resources. Thus, buffering reduces the receiving overhead by reducing the frequency of polling.

On the sender's side, a network interface may not always be able to insert a new outbound packet into the network. A send is denied if the necessary resources are exhausted by the pending packets that are waiting for a negligent receiving process. Furthermore, when network traffic is heavy, the network may be too busy to accept a new packet on demand. Therefore, the network interface should also provide some outbound packet buffering to allow the network interface to continue accepting outbound packets from the user processes even when the outbound network channel is temporarily incapacitated. Buffering outbound packets reduces the sending overhead by reducing the incidence of when a send cannot proceed and the sending process must retry.

The effectiveness of these buffers in reducing communication overhead depends heavily on the size of these buffers. If the buffers overflow easily under normal usage, then they serve little purpose. The buffers need to be large enough to absorb the variation in the amount of network traffic so the user processes can perceive the network activity as stable. However, as described so far, arbitrarily enlarging the buffer

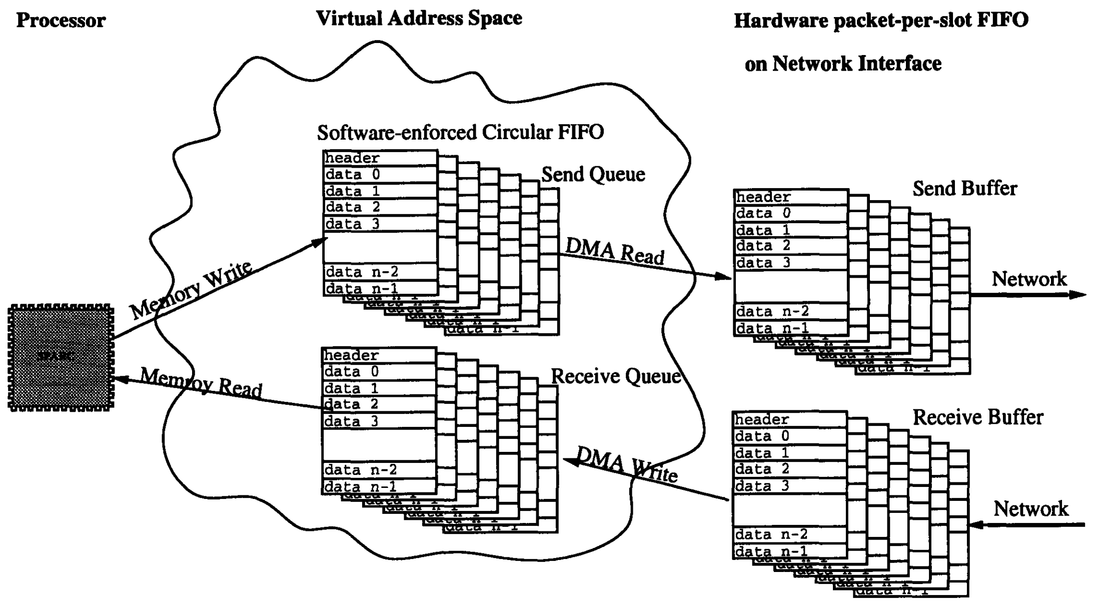

Figure 2.3: A Message Interface based on software FIFO's in User's Virtual Memory

to match the possible network variation is not possible. The buffering is provided as hardware FIFO's between the memory-mapped interface register array and the network. There is a practical constraint on the maximum buffer size that can be provided cost-effectively on the network interface. More importantly, these buffers are hardware states that need to be saved and restored when a process is context switched on our time-sharing system. An exceedingly large buffer would introduce an unacceptable context switching overhead.

To overcome this limitation, FUNi logically extends these buffers into the user virtual memory space where memory is cheap and plentiful. Figure 2.3 illustrates this idea. In the register array-based design shown in Figure 2.2, a user enqueues an outbound packet, through memory-mapped writes, into the hardware send FIFO. On the other hand, the network interface locates the outbound packet that is ready for dispatch by dequeuing it from the hardware send FIFO. A similar effect can be achieved with a software enforced circular FIFO queue in the user memory, which is maintained jointly by the user software and the network interface hardware. In operation, instead of enqueuing into the hardware FIFO through memory-mapped writes, the user processes would enqueue the outbound packet into the head of a circular FIFO queue in memory. FUNi, instead of popping an outbound packet from the hardware FIFO, would now retrieve, through DVMA, pending outbound packets from the tail of the circular FIFO queue.

A similar transformation can be made for the receive memory-mapped register array and buffer. A receive circular FIFO queue is maintained jointly by the network interface hardware and the user software. Through DVMA, FUNi enqueues inbound packets from the network into the head of the circular FIFO queues, and the user process dequeues the inbound packets from the tail of the circular FIFO queues.

The basic operations for sending and receiving are the same as the memory-mapped register-based design. For sending, instead of performing memory-mapped writes to compose outbound packets in a logical array of memory-mapped registers, the user process would store to a logically equivalent array at the head of the circular send queue in real memory. Instead of receiving packets with memory-mapped reads from a logical array of memory-mapped registers, inbound packets can be accessed from a logically equivalent array at the tail of the receive queue. The interpretation of the logically equivalent array structure for sending and receiving remains identical to the memory-mapped register array. The overhead for examining and maintaining the circular FIFO indices can also be equated to the overhead for issuing push or pop commands and checking the return status of the command in the memory-mapped register array implementation. The overall overhead for sending and receiving a packet when using the circular FIFO queue is comparable to the overhead for sending and receiving a packet through memory-mapped registers.

In fact, when coupled with appropriate hardware support, the software-enforced FIFO interface can lower communication overhead. With most workstation implementations, the compulsory latency for a memory-mapped read – not accounting for the latency resulting from a slow reacting device – requires on the order of a few tens of cycles. With the memory-mapped register interface, receiving a packet, which requires multiple reads, could accumulate a substantial overhead. However, in the case of receiving from a queue in the user memory, if the processor's data cache can snoop FUNi's DVMA transactions and cache the snooped values, the accesses to the inbound packets in the receive queues would hit in the cache, thus resulting in a reduction in overhead. (Unfortunately, this feature will not be available with the current generation of the SPARCstation since cache coherence is maintained by invalidating snooped cache lines.)

Moving the buffers into the user memory space allows for a much greater buffering capacity than in hardware because we are no longer constrained by the context switch overhead associated with the large hardware states. Regardless of the buffer size, the hardware state that needs to be maintained in the network interface is finite. The logical size of the buffers can be arbitrarily enlarged in the paged virtual address space. The memory-management unit manages the context switching of the queues with the rest of the user's virtual memory. This provides the buffer size necessary for the program execution to tolerate our highly distributed parallel system in which fine-grained coordination of peer processes is not possible. Placing the interface buffer into the memory system has the additional benefit of decoupling the software overhead of communication from the bandwidth and latency of accessing a network interface. The user process can enqueue and dequeue outbound and inbound packets at its own rate independent from the bandwidth that is available to the FUNi SBus device.

2.6 Message-Passing plus Memory-to-Memory

With a network interface that only supports message passing between processors, in order to perform a block memory transfer, the user process must divide the block into small chunks and transfer the chunks in individual packets. The sending process must explicitly copy, in verbatim, each byte of transfer from the source to the interface, and the receiving process must later explicitly copy, in verbatim, each byte from the interface to the destination location.

The overhead of data movement on the sending and receiving processors can be eliminated on our DMA-based message interface. FUNi has the ability to access the user memory. Since the transfer data is already sequentially formatted, requiring no further formatting, FUNi can compose the transfer packet directly from its source. Similarly, FUNi can use DVMA to write the data from inbound transfer packets directly to their destination location. This DMA-style remote block transfer will significantly reduce the transfer overhead by eliminating the data movement overhead on both the sending and the receiving nodes.

2.7 Cached Memory-mapped Status Registers

The user-level process reads FUNi's memory-mapped registers to inquire the status of FUNi. During polling, the user process checks a status register to detect the presence of pending inbound packets. The long latency of a memory-mapped access leads to a large polling overhead. For simplicity, let us suppose that a packet arrival is signaled by a bit in the FUNi status register. To poll, the user process repeatedly checks the status register through memory-mapped reads and waits for the bit to change from *empty* to *ready*. When the bit is *ready*, the poll succeeds and the overhead of polling is amortized by the useful computation resulting from the poll. However, while no new inbound packets arrive, the status bit remains unchanged as *empty*, The failed polls during that period are wasteful communication overhead because each costly memory-mapped read only produces the same information that has not changed since the last read.

Caching the memory-mapped status register can reduce this overhead by eliminating unnecessary memory-mapped reads. Caching hardware registers normally leads to incorrect behavior because the program does not see the current value in the hardware register. However, the program only needs to see the actual value if the content of the hardware register has changed since the last value was loaded into the cache. Thus, the user program can be guaranteed to see the correct value if FUNi invalidates the stale cached value of the status register whenever the content of the status register changes. Cache invalidation can be achieved by using the cache coherence protocol on the memory bus.

Again, with the example of polling, let us assume FUNi has just been serviced, and all inbound packets have been received, thus causing the status bit in FUNi to change from *ready* to *empty*. To expose this information, the network interface invalidates the cached line for the status bits. The next time the memory-mapped address of the status register is referenced by the processor, the *empty* value is loaded from FUNi into the cache. Prior to the arrival of a new packet, all subsequent memory-mapped reads of the status register hits in the cache, and each time the user sees the cached *empty*. The overhead from polling is reduced since unnecessary memory-mapped reads are replaced by low-latency cache accesses. When a new packet arrives, the value in the hardware status register changes from *empty* to *ready*, and FUNi again invalidates the cached value. Thus, on the next memory-mapped read, the cache will miss. The new *ready* value is loaded and correctly seen by the user process.

Please note that FUNi does not actually use a status bit to indicate the availability of pending packets. However, polling does indeed involve checking a memory-mapped register. Section 3.2.3 describes the actual scheme used in polling.

This section has presented the final refinement to FUNet and FUNi that is visible to the programmer. The remainder of this chapter discusses the ideas in the underlying hardware mechanisms of FUNet and FUNi.

2.8 Acknowledgment/Retry End-to-End Flow Control Protocol

In a simple network contract, when a process issues a send to the network interface, the interface attempts to insert the packet into the network. If the network cannot accept any more packets, the interface denies the send request. If the network has room, the outbound packet is accepted and inserted into the network. Once a packet is inserted into the network, the network interface and the sending process are no longer responsible for the packet. The network contract guarantees the packet will emerge at the destination network interface, and the responsibility of delivery belongs solely to the destination network interface.

This simple network contract has a few drawbacks. If one node is unable to absorb the influx of inbound packets, the network is forced to buffer the unreceived packets, and network routing resources can backup to all the senders of that node. This congestion would also block other sending processes whose packets need to use a part of the blocked path. This network protocol is inefficient in coping with congestion due to mismatched sender/receiver pairs since other unrelated sending processes can be blocked indirectly, and force their available receivers to wait.

This first inefficiency is a performance problem that can be avoided with proper programming. However, there is an unavoidable logistic problem when the sender is allowed to relinquish all responsibility once the packet enters the network. For example, because of the distributed nature of our proposed system, we must make provisions to allow for simultaneous execution of multiple application contexts on different workstations and thus must also allow the possibility that when a packet arrives at the destination workstation, the correct receiving processes may not be executing. What a receiving network interface should do with a undeliverable packet is not clear. The network interface must absorb the packet from the network so further inbound packets would not be blocked. The receiving network must hold on to the packet because it is solely responsible for its eventual delivery, but buffering all such packets could require an unbounded buffer resource.

An acknowledgment/retry end-to-end flow control protocol, similar to the Selective Repeat Protocol [19], is used to address both the congestion and buffering problems. The FUNi hardware conducts this protocol transparently from user programs. When FUNi absorbs an inbound packet from the network, it needs to return

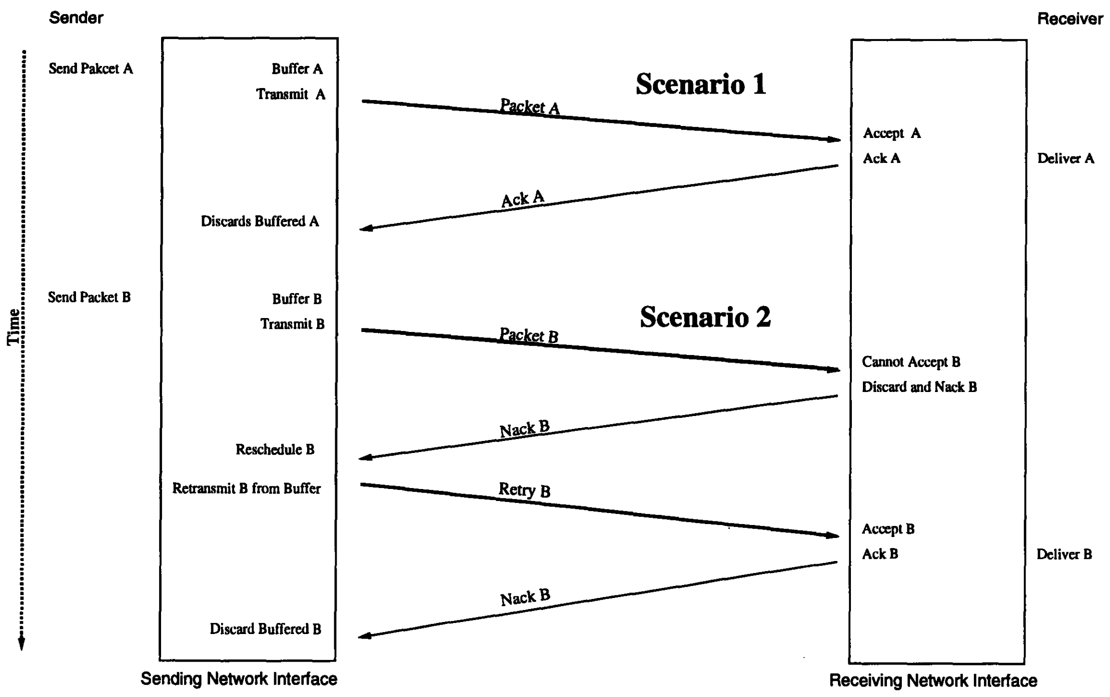

Figure 2.4: Two Scenarios in the Acknowledgment/Retry Protocol

an acknowledgment to the originating FUNi. If FUNi accepts the packet, a positive acknowledgment is sent back to acknowledge the acceptance, as shown in Scenario 1 in Figure 2.4. If FUNi cannot accept the packet for any reason, a negative acknowledgment can be returned to request the originating FUNi to re-send the packet, as shown in Scenario 2 in Figure 2.4. The sending FUNi needs to retain a copy of each outbound packet until its delivery is positively acknowledged.

Under this protocol, the receiving FUNi can simply reject and drop unacceptable packets and expect a retransmission from the sender. The protocol requires the originating FUNi to be responsible for buffering its outbound packets until the packets are accepted and acknowledged. The buffering requirement on the sender side can be well defined.

With the ability to reject packets, FUNi at each node can continuously absorb packets from the network, even when the packets cannot be accepted. Packets will flow on the network regardless of the behavior at each individual node. FUNi can never be indirectly blocked from communication by other misbehaving communication pairs. FUNi can only be denied from sending if its receivers are not accepting the inbound packets, thus causing the sending FUNi to run out of buffering resources for undelivered packets. Thus, this mechanism serves as an automatic rate control for throttling the network activities of over-active sending processes, thus preventing them from swamping other processors with messages.

2.9 Undelivered Packet Cache

The hardware acknowledgment protocol integrates well with our network interface design with message interface based on packet queues in user memory. The network interface could leave undelivered outbound packets in the send queues, and the queues would provide the buffering needed in this protocol. However, retrieving packets through DVMA from the user memory for retry is not only slow but also wastes DVMA bandwidth between the send queue and FUNi. The overall bandwidth of the system bus would also be degraded by the extra DVMA traffic.

To streamline the implementation of the acknowledge/retry protocol, FUNi could maintain a small hardware cache for undelivered packets. The network interface would only dequeue pending outbound packets from the send queues into the Undelivered Packet Cache when space are available in the cache. Each dequeued but undelivered packet remains in the packet cache and is transmitted repeatedly after each negative acknowledgment until a positive acknowledgment for its delivery is received. Since the Undelivered Packet Cache is implemented in hardware, retries of packets can be dispatched much more rapidly and fill up the idle network bandwidth.

2.10 Two-level Network

In the previous section that described the acknowledgment/retry end-to-end flow control protocol, we claimed that the acknowledgment protocol allows packets to flow on the network regardless of the behavior at each individual node. This statement requires further qualification.

To keep the network traffic flowing, all network interfaces have to continuously absorb inbound packets to release the network resource for new packets. However as the protocol dictates, for every data packet absorbed, the network interface must transmit an acknowledgment packet. This stipulation of the acknowledgment protocol enforces a dependency on the network traffic flow. A network interface must stop absorbing inbound packets when the outbound path is blocked. This dependency can lead to deadlock.

To demonstrate a possible scenario for deadlock, let us assume that node A is repeatedly sending packets at maximum rate to node B. Furthermore, let us also assume that node B is sending packets to an irrelevant third party. Since node B's outbound path is heavily utilized, it would not be able to absorb and acknowledge node A's packets as fast as node A is transmitting. Eventually a trail of packets will back up from node B to node A. Once the trail reaches from node B all the way to node A, node A would only be able to insert a new packet in its outbound path sometime after node B has a chance to absorb a packet from node A that is waiting on node B's inbound path.

At the same time, a more sparse trail of acknowledgment packets will run from node B to node A since node B transmits an acknowledgment to node A after absorbing each packet. Now, let us suppose suddenly all other nodes in the system begin to send to node A. All paths leading to node A will become heavily congested. With its outbound path blocked by the trail of packets going to node B, node A cannot freely absorb the influx of packets because it cannot return the acknowledgments. The inbound path leading to node A jams up with data packets, possibly blocking acknowledgment packets from node B. If we are unlucky, node B's outbound path will now be backed up with acknowledgment packets to node A. Node B can no longer absorb any packet until node A can absorb data packets to allow the node B's blocked acknowledgment packets to progress. However, node A cannot absorb any packet until node B absorbs a data packet to make room on node A's outbound path. The network is deadlocked.

FUNet supports two network packet priorities to ensure that FUNet will not deadlock under the acknowledgment protocol. The two-level FUNet always reserves resources for the higher priority packets' exclusive use. This can guarantee that the flow of high priority packets is never blocked by the lower priority packets. On FUNet, all data packets are transported as low priority packets, whereas acknowledgment packets are transported as high priority packets. Since the acknowledgment protocol does not enforce a dependency on acknowledgment packets, FUNi can always absorb the acknowledgment packets. Since the acknowledgment packets are continuously absorbed, the flow of acknowledgment packets can never be deadlocked by other acknowledgment packets. If the two-level FUNet ensures that the lower-priority data packets can never block acknowledgments, we can guarantee acknowledgment packets will never deadlock. Since the flow of acknowledgment packets will never block, FUNi will always be able to absorb data packets from the network. Thus, the flow of data packets can also never deadlock.

Chapter 3

FUNi Programming Interface

The basic message sending and receiving interfaces are based on software-enforced FIFO queues within user memory. The queues are jointly maintained by the user software and FUNi's hardware. Two sets of send and receive queue pairs are provided, one for each packet priority. Thus, there is a total of four queues: reply send queue, reply receive queue, request send queue and request receive queue. The user process assumes the role of the producer on the send queues and the consumer on the receive queues. FUNi performs the exact opposite. All FUNi transactions always give precedence to reply packets so that the reply packets' traffic is allowed to overtake request packets when competing for resources.

The circular queues rely on the standard convention of head and tail indices. The head index points to the head, the next free slot for enqueuing a new item. The tail index points to the tail, the next occupied slot to dequeue from. A queue is empty when both the head index and the tail index point to the same slot. A queue is full when the head is logically immediate before the tail. For each queue, the user software controls one end of the queue, and FUNi controls the other end. The two parties rely on a set of memory-mapped registers, called Queue Registers, to relay information about the indices. The producer of the queue uses one register to pass the head index to the consumer so the consumer knows how far to proceed in the queue. The consumer uses one register to pass the tail index to the producer so the

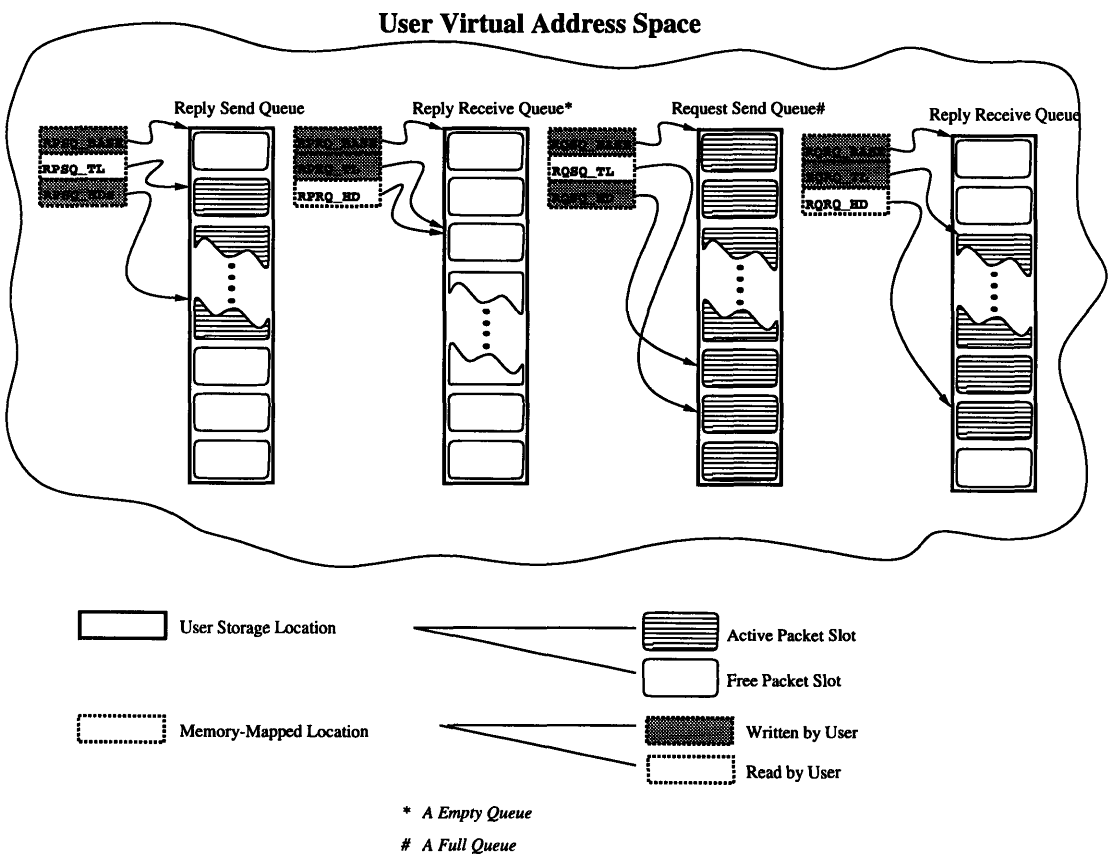

Figure 3.1: FUNi Message Interface

producer knows which slots are freed. There are also four other memory-mapped Queue Registers that FUNi uses to locate the base of the queues, plus one more register that the user uses to specify the size of the queues. Figure 3.1 depicts FUNi's message sending and receiving interface in a user's virtual memory space.

Three other memory-mapped registers fall into the category of status and control registers. The software reads and sets the registers to examine the status and to control the behavior of FUNi. The privileged controls are protected from the user processes by address mapping schemes. Please see Section 4.1.2 for the protection scheme.

The following section describes the FUNi programming interface. The section starts with the organization of the circular queue and then moves on to describe the memory-mapped registers. Following the presentation of the programming interface, explanations and examples on how to manipulate the FUNi registers and queues for communication on FUNet are given.

To clarify the description in the following sections, UPPER-CASE SANS SERIF TYPE is used when referring to FUNi's memory-mapped registers. `Typewriter Type` is used to refer to variables in the user programs. When the asterisk symbol, '*', appears in these fonts, it is used as a single wild-card character, much like the usage of '?' in the UNIX shell language, to refer to multiple items.

3.1 Interfaces

3.1.1 Send and Receive Queues

The user processes do not use memory-mapped accesses to directly interface with FUNi for sending and receiving network messages. Instead, a user process must interact indirectly through the memory system. The packets are held intermediately in software-enforced FIFO-ordered queues within the user's memory space. FUNi accesses these queues through DVMA operations.

The queues used for message passing are simply four software-enforced circular buffers that the user program allocates within the user's memory space. The queues must be of the same size, but the user can specify the queue size to range from 2 to 64K (in power of 2's) packet slots per queue. The queue size can be dynamically adjusted throughout the user process's execution.

Each queue is logically divided into slots of 32 words. Each slot is used to hold a single packet. The memory utilization of the packet queue is fairly poor. At most 24 words of the 32-word slot will contain useful information. It is possible to improve memory utilization by dividing the queues into smaller slots and allowing individual packets to occupy a variable number of slots depending on the packet size, but the improvement in memory utilization is not essential and cannot justify for the extra overhead introduced. A queue capable of holding 1K packets is only 128 KB which places very little burden on the memory system. Furthermore, the queue accesses only occur near the head and tail of the queue and thus have very good locality. Since the queue is in the user's virtual memory space, even in the case when large queues are employed, only the pages near the head and the tail need to be resident in the physical memory. Elaborate schemes to improve memory usage in the packet queues were experimented with and rejected because they overly complicate the management of queue indices, which leads to additional communication overhead.

The details regarding the usage and management of the interface queues are presented in Section 3.2. The next section will describe the memory-mapped registers used to relay information between the user programs and FUNi.

3.1.2 Memory-mapped Registers

The user programs access the memory-mapped registers by reading and writing to the memory-mapped addresses of the registers. A memory-mapped address is composed of two parts: the page address and the page offset. Bit[6:2] of the page offset determines which one of the sixteen FUNi memory-mapped registers is specified. This section first describes the Queue Registers and then describes the control and status registers.

Queue Registers

The interface registers described in this section are memory-mapped registers used for maintaining the coherence of the queue structures. These registers contain information used by FUNi and the user programs to determine the appropriate action on the queues. For more details regarding the usage of these registers during communication, please refer to Section 3.2.

FUNi Q_MASK 16-bit read/write

The user process initializes this register to specify the size of the four same-sized packet queues. For queue size of $2^{N(sizeof(slots))}$ bytes, the register need to be set to 2^N-1. Since, the FUNi Q_MASK register is only 16-bit wide, the queues can contain at most 2^{16} slots each, sufficient to hold 64K packets. In the current specification, each queue is logically divided into slots of 32 words. Therefore, the queues have a maximum size of 16 MByte each.

FUNi RPSQ_BASE
FUNi RQSQ_BASE
FUNi RPRQ_BASE
FUNi RQRQ_BASE 32-bit read/write

These registers contain the base addresses of four circular packet queues (RPSQ = reply send queue, RQSQ = request send queue, RPRQ = reply receive queue and RQRQ = request receive queue). The user process initializes these registers with pointers to the memory spaces allocated. Each address must be 32-word aligned, or will be automatically truncated for alignment. Memory-mapped writes to each of these four FUNi R**Q_BASE registers cause the corresponding FUNi R**Q_HD and FUNi R**Q_TL registers to automatically initialize to zero.

FUNi RPSQ_TL
FUNi RQSQ_TL 16-bit read/write
FUNi RPSQ_HD
FUNi RQSQ_HD 16-bit read/write

FUNi maintains the FUNi R*SQ_TL registers that hold the indices to the tail of the circular send queues, and the user process updates the FUNi R*SQ_HD registers with the indices to the head of the circular send queues. For the send queues, the user process is the producer while FUNi is the consumer. If (FUNi R*SQ_TL $\neq$ FUNi R*SQ_HD), i.e., (tail $<$ head), FUNi knows the corresponding queue is not empty and will dequeue the outbound packet from the tail of the queue. After FUNi retrieves the packet, FUNi increments the index in the corresponding FUNi R*SQ_TL by one and re-checks the queue for any more pending outbound packets. Since FUNi only processes pending packets up to the slot indexed by FUNi R*SQ_HD, the user needs to update FUNi R*SQ_HD after enqueuing outbound packets to make the newly enqueued packets visible to FUNi. For examples on the usage of these registers in sending a packet, please refer to Section 3.2.2.

FUNi RPRQ_HD		
FUNi RQRQ_HD	16-bit	read/write
FUNi RPRQ_TL		
FUNi RQRQ_TL	16-bit	read/write

FUNi maintains the FUNi R*RQ_HD registers that hold the indices to the head of the circular receive queues, and the user process updates the FUNi R*RQ_TL registers with the indices to the tail of the circular receive queues. For the receive queues, the user process becomes the consumer while FUNi acts as the producer. Each time an inbound packet arrives, FUNi determines whether there is sufficient space in the receive queue by comparing the corresponding FUNi R*RQ_HD against the FUNi R*RQ_TL. If there is enough space, FUNi delivers the packet to the queue through DVMA. After writing the packet to the user's memory, FUNi increments the index in the corresponding FUNi R*RQ_HD register to record the delivery of the new packet. After dequeuing inbound packets, the user needs to update FUNi R*RQ_TL before the slots occupied by the dequeued packets are released for reuse. For examples on the usage of these registers in receiving a packet, please refer to Section 3.2.3.

FUNi RPRQ_HD and FUNi RQRQ_HD are status registers that the user process reads to determine the presence of unreceived packets in the receive queue. Memory-mapped addresses of the two registers are on a cacheable page, separate from the other memory-mapped registers, to reduce the communication overhead as described in Section 2.7. For implementation reasons, the memory-mapped page address of FUNi RPRQ_HD and FUNi RPRQ_HD actually references a real page in memory that is prepared by the operating system. FUNi updates the values in memory with DVMA writes. The operating system uses the real memory-mapped addresses of the registers when reading and writing to the two registers during context switches.

Control and Status Registers

The following registers directly control the operation and report the status of FUNi.

FUNi CNTL 8-bit read/write

This is the main control and status register of FUNi. The FUNi hardware carries on three transactions: retrieval, delivery and dispatch. The retrieval transaction retrieves outbound packets from the send queues into FUNi, whereas the delivery transaction delivers inbound packets into the receive queues. The dispatch transaction is responsible for scheduling transmissions of outbound packets. The FUNi CNTL register is bit-fielded. Bits 3, 4, 5, 8 and 10 report the status of these three transactions. Bits 0, 1, 2, 6, 7 and 9 are mode switches that control the behavior of the three transactions. The modes can be set independent of one another. Bits 0, 1, 2 and 6 are privileged control bits. An attempt to set or clear the privileged control bits by user-level processes is ignored.

bit 0	MERR
bit 1	RESET
bit 2	CTXM
bit 3	CTXR1
bit 4	CTXR1'
bit 5	CTXR2
bit 6	DRAIN
bit 7	CRQM
bit 8	CRQR
bit 9	CSQM
bit 10	CSQR

bit 1: MERR Memory Error read /privileged write

This status bit becomes set when a FUNi-initiated SBus DVMA operation fails because of a memory error. When this bit sets, FUNi should generate an interrupt to the CPU for service. The operating system needs to clear this bit when the memory error is resolved. FUNi will suspend all DVMA operations while this bit is set.

bit 2: RESET Reset Mode read/ privileged write

FUNi enters Reset mode when the operating system sets this privileged control bit. In Reset mode, FUNi ceases all operations. The operating system needs to clear the RESET bit to restart FUNi. After resetting, FUNi enters the power-up state, in which FUNi will be in Context-Switch mode with all registers uninitialized.

bit 3: CTXM Context Switch Mode read/privileged write

FUNi enters Context-Switch mode when the operating system sets this privileged control bit. Once this bit is set, all of FUNi's transactions (retrieval, delivery and dispatch) will halt to bring FUNi to a state, ready for context switching. FUNi will also reject all further inbound packets by returning negative acknowledgments to their senders. After setting CTXM, the operating system must wait for the CTXR1 and CTXR2 bits before proceeding to modify the states of the FUNi registers. Violation of this condition will lead to corrupted states and unpredictable behavior.

After the context switch, the operating system clears the CTXM bit to restart FUNi's normal operation. For details regarding the steps in context switching, please refer to Section 4.2.

bit 4: CTXR1 Context Switch Ready 1 read only

This status bit is used to indicate, in response to CTXM, that FUNi's retrieval and dispatch transactions have halted and that all packets in the Undelivered Packet Cache are not outstanding. This leaves FUNi in a state in which it is safe to modify the FUNi GID and the FUNi TICKET registers.

bit 5: CTXR1' Context Switch Ready 1 Prime read only

The condition indicated by this status bit is a subset of CTXR1. Specifically, packets in the Undelivered Packet Cache can be outstanding. After setting CTXM, if CTXR1 is not set after a sufficient amount of time has elapsed for the acknowledgments of all undelivered packets under the assumption of failure-free network, the operating system can conclude that some network failure has occurred and that the missing acknowledgments will never return. In this case, the operating system should proceed with the context switch after only CTXR1'. This allows the operating system to switch out a process and continue with other processes in the presence of a transient network failure. If a process is context switched while some of its packets are still waiting for their return acknowledgments, the execution state of the process is irrecoverably damaged. Successful continuation cannot be guaranteed.

bit 6: CTXR2   Context Switch Ready 2   read only

This status bit is used to indicate, in response to CTXM, that FUNi's delivery transaction has halted and that the internal receive buffers are emptied. FUNi SBus card contains two small high-speed hardware receive buffers for keeping pace with the higher-bandwidth Arctic network. At any time, there may exist some packets waiting in the buffer for the current context. The operating system must wait until CTXR2 becomes set, indicating the hardware buffers have been drained completely, before turning FUNi over to the next context.

bit 7: DRAIN   Drain Buffer Mode   read/privileged write

FUNi enters Drain-Buffer mode when this control bit is set. The operating system uses this mode to drain the SBus card's internal receive buffers. In this mode, instead of delivering remote DMA transfer packets directly to their destinations, FUNi enqueues the transfer packets into the receive queues. The delivery of message-passing packets is unchanged.

bit 8: CRQM   Change Receive Queue Mode   read/write

FUNi enters Change-Receive-Queue mode when this control bit is set. FUNi's delivery transactions will halt so the FUNi R*RQ_BASE and FUNi Q_MASK registers can be modified to change the location or the size of the receive queues. The contents of these registers must not be modified until CSQR becomes set, or unpredictable behavior results. To modify the FUNi Q_MASK register, the user must set both CSQM and CRQM, and wait for both CSQR and CRQR to become set because both the send queue and receive queue are affected.

bit 9: CRQR   Change Receive Queue Ready   read only

This bit is used to indicate that FUNi's delivery transaction has halted in response to CRQM. When the CRQR bit is set, the user can modify the FUNi R*RQ_BASE registers safely without corrupting the receive queues. If CSQR is also set, the FUNi Q_MASK register can be modified.

bit 10: CSQM   Change Send Queue Mode   read/write

FUNi enters Change-Send-Queue mode when this control bit is set. FUNi's retrieval transaction will halt so the FUNi R*SQ_BASE and FUNi Q_MASK registers can be modified to change the location or the size of the send queues. The contents of these registers must not be modified until CSQR becomes set, or unpredictable behavior results. To modify the FUNi Q_MASK register, the user must set both CSQM and CRQM, and wait for both CSQR and CRQR to become set.

bit 11: CSQR   Change Send Queue Ready   read only

This bit is used to indicate that FUNi's retrieval transaction has halted in response to CSQM. When the CSQR bit is set, the user can modify the FUNi R*SQ_BASE registers safely without corrupting the receive queues. If CRQR is also set, the FUNi Q_MASK register can also be modified.

FUNi GID   11-bit   read/privileged write

This register holds the 11-bit Group Identifier (GID) of the current application. FUNi uses the identifier to tag network packets for data protection among time-sharing applications. The lower bits of the FUNi GID register are also used to index the current context of the multi-context Undelivered Packet Cache. Only the operating system is allowed to write to the privilege FUNi GID register. The operating system needs to be careful when modifying the FUNi GID register during a context switch. Please see Section 4.2 for detail.

FUNi TICKET	8-bit	read/privileged write

The eight bits of the FUNi TICKET register bitmap the availability of the eight lines in the Undelivered Packet Cache for the current context. The dispatch transaction unit uses the content of this register (and other internal registers) to determine which Undelivered Packet Cache line contains undelivered packet that is waiting for dispatch. When FUNi's retrieval transaction retrieves a packet from the send queue, the packet is stored in an available line in the packet cache, and the bit corresponding to that line is cleared. The bit becomes set again only after the packet is delivered and positively acknowledged.

FUNi is defined to allow up to 32 undelivered packets per context. The width of this register should expand according to the cache size.

3.2 Usage

3.2.1 Initialization

When each peer-process of a parallel application starts up, the operating system on each workstation prepares its FUNi for access. This preparation entails first swapping the pre-existing context if another process was using FUNi. Then, the operating system assigns a new GID to the FUNi GID register and resets the FUNi TICKET register. Since the user-controlled Queue Registers are left uninitialized, the operating system must also leave FUNi in the Change-Receive-Queue mode and Change-Send-Queue mode to prevent FUNi from executing haphazardly based on the uninitialized register contents. (At power-up, FUNi automatically enters Context-Switch mode for analogous reason). The program must first properly initialize the registers before activating FUNi to start network communication.

During the user startup initialization, the program first allocates the memory space for the four queues and initializes the FUNi R**Q_BASE registers with the base addresses of the four queues. (The FUNi R**Q_HD and FUNi R**Q_TL registers will automatically initialize to zero as the side effect of initializing the FUNi R**Q_BASE registers.) A appropriate queue size should be determined based on the expected program behavior and hardware configuration. The queues can be of size $(2^N slots)(32\frac{word}{slot})(4\frac{byte}{word})$ where $1 <= N <= 16$. The user program sets the FUNi Q_MASK register to 2^N-1 to indicate the queue size. Finally, the user program clears the CSQM and CRQM bits in the FUNi CNTL register to activate FUNi for sending and receiving.

To allow a user program the flexibility to adjust the amount of buffering space, the FUNi R**Q_BASE and FUNi Q_MASK registers can be modified throughout the program execution. Prior to modifying the FUNi R**Q_BASE and FUNi Q_MASK registers, the user must set the appropriate control bits (CRQM and/or CSQM) in FUNi CNTL and wait for the appropriate conditions (CSQR and/or CRQR). (Section 3.1.2 describes the semantics of these bitfields in the FUNi CNTL register.) Improper initialization or adjustment of these queue parameters results in incorrect program behavior.

The following is an example of user startup initialization of FUNi with a queue size of NUM_SLOT*32*4 bytes.

```
/* set CSQM and CRQM if not already set */

/*       ______memory_mapped_address_______
/*       _base_addr__+ __register_select___          */
*(int *)(MM_FUNi_BASE+(NIO_CNTL<<NIO_REG_OS)=
  (NIC_CSQM_MASK | NIC_CRQM_MASK);

/* wait for FUNi transactions to halt */
while (!(*(int *)(mm_FUNi_base+(NIO_CNTL<<NIO_REG_OS)) &
                (NIC_CSQR_MASK | NIC_CRQR_MASK)));

        /* initialize Q_MASK register */
        *(int *)(MM_FUNi_BASE+(NIO_QMASK<<NIO_REG_OS))=NUM_SLOT-1;


        /* allocating and initializing reply send queue */
        (*(int **)(MM_FUNi_BASE+(NIO_RPSQ_BASE<<NIO_REG_OS)))=
          (int *)malloc(NUM_SLOT*32*SIZEOF(INT));

        /* allocating and initializing reply receive queue */
        (*(int **)(MM_FUNi_BASE+(NIO_RPRQ_BASE<<NIO_REG_OS)))=
          (int *)malloc(NUM_SLOT*32*SIZEOF(INT));

        /* allocating and initializing request send queue */
        (*(int **)(MM_FUNi_BASE+(NIO_RQSQ_BASE<<NIO_REG_OS)))=
          (int *)malloc(NUM_SLOT*32*SIZEOF(INT));

        /* allocating and initializing request receive queue */
        (*(int **)(MM_FUNi_BASE+(NIO_RQRQ_BASE<<NIO_REG_OS)))=
          (int *)malloc(NUM_SLOT*32*SIZEOF(INT));

        /* clear CSQM and CRQM to restart FUNi */
        *(int *)(MM_FUNi_BASE+(NIO_CNTL<<NIO_REG_OS)=0;
```

3.2.2 Sending Network Messages

Once FUNi has been initialized and activated, the user process can send packets by enqueuing to the head of the send queues. This section describes the steps involved in sending a reply-priority message-passing packet. The steps required for sending a request packet are identical except for the substitution of the corresponding registers and queues. (Steps involved in sending a special XFER-type, direct memory-to-memory transfer, packet are discussed in Section 3.2.4)

Let us assume `(int*)Send_Rp_Q_base` is a program variable pointing to the base of the reply send queue, and `(int)Q_mask` holds the value, one less than the number of slots in each queue, that was assigned to the FUNi Q_MASK register. Let us also assume **(int)Send_Rp_Q_head** is the index to the head (the next open slot) of the circular send queue. (This variable should be initialized to 0x0 when the reply send queue is initialized). Please refer to Figure 3.2 for the relationship among the different software and hardware pointers and indices.

Before enqueuing a new outbound packet, the user process must first determine if the queue has space for the new packet. If FUNi RPSQ_TL == ((**Send_Rp_Q_head** + 1) & **Q_mask**), the queue is full, and an attempt to enqueue another packet will overflow the queue. Overflowing the circular queue misleads FUNi into incorrect behavior and can irrecoverably destroy the queue data structure. If the queue is full, the user process must wait until FUNi retrieves a packet from the queue and releases the slot occupied. Later in this section, we will describe a way to eliminate the wait by operating from a virtual send queue while the true send queue is full.

If the FUNi RPSQ_TL register must be checked prior to sending every packet, the memory-mapped read latency will introduce a significant additional overhead to the send operation. Luckily, this memory-mapped read penalty can be avoided through memoization. The user process can read the register once using a memory-mapped read and memoize the index. From then on, the user only needs to compare **Send_RP_Q_head** with the memoized index. Because the true value of FUNi RPSQ_TL could only have incremented during the execution, as long as **Send_RP_Q_head** does not overlap the memoized index, **Send_RP_Q_head** cannot overlap FUNi RPSQ_TL. The user process only needs to reread FUNi RPSQ_TL to update the memoized tail index when **Send_RP_Q_head** is logically immediately before the memoized index. Usually, for a sufficiently large queue size, one needs to invoke the costly memory-mapped read to FUNi RPSQ_TL only once each time **Send_RP_Q_head** circulates the queue.

Once the user process is certain that the head slot indexed by **Send_Rp_Q_head** is available, the user can compose the outbound packet in the packet slot. The user starts by writing the packet header to the first word of the head slot at address **(int*)Send_Rp_Q_base** + (**Send_Rp_Q_head**<<5). The header word is composed of

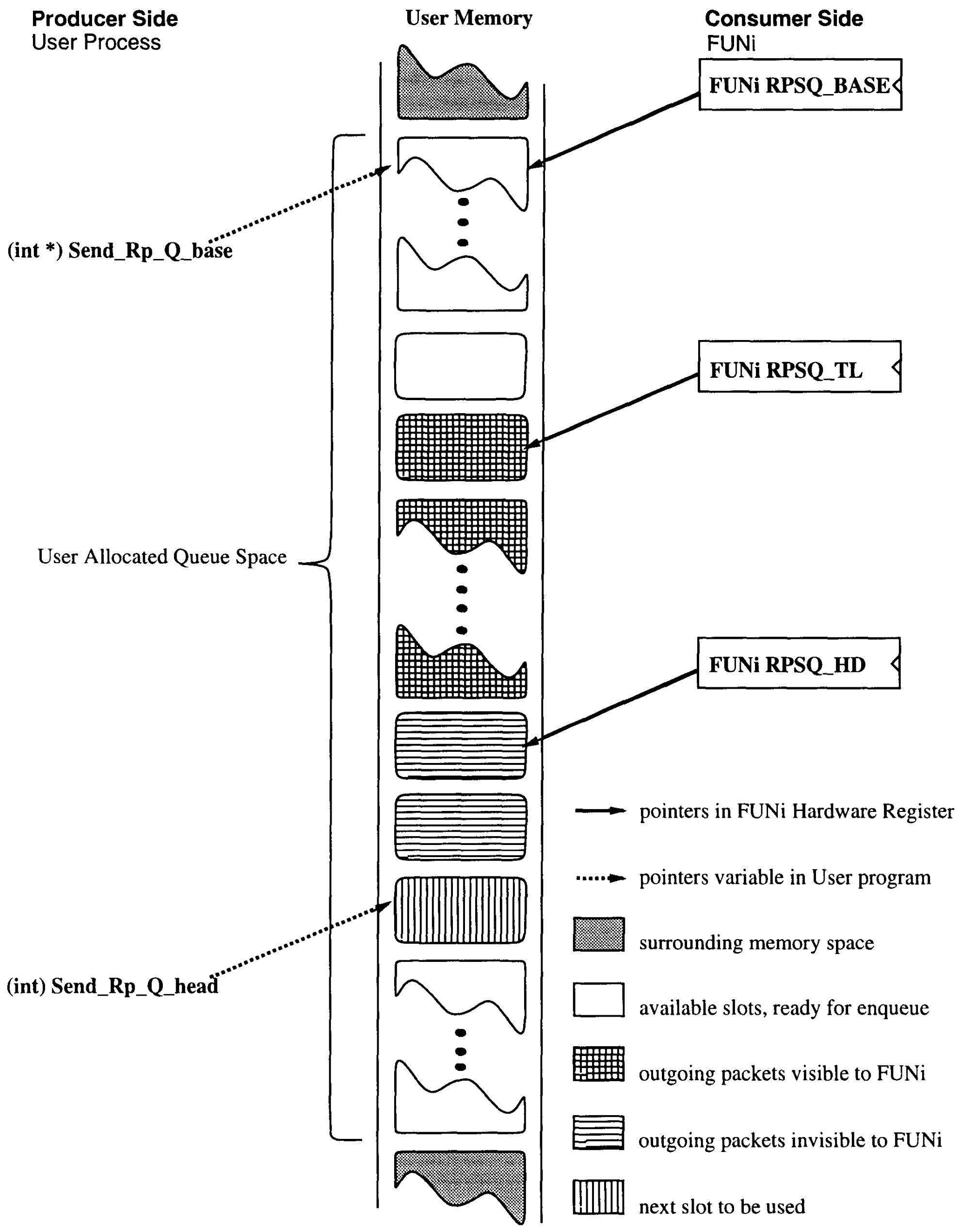
Producer Side
User Process
User Memory
Consumer Side
FUNi
FUNi RPSQ_BASE
FUNi RPSQ_TL
FUNi RPSQ_HD
(int *) Send_Rp_Q_base
User Allocated Queue Space
(int) Send_Rp_Q_head
pointers in FUNi Hardware Register
pointers variable in User program
surrounding memory space
available slots, ready for enqueue
outgoing packets visible to FUNi
outgoing packets invisible to FUNi
next slot to be used

Figure 3.2: Organization of Reply Send Queue

four fields. (Bit[31] is set to 0 for message-passing packets. Bit[30] is not used. Bit[29:21] specifies one of the 256 user packet types. Bit[20:16] of the header word specifies the length of the packet in words. Length of 0 to 21 words can be specified. If a length longer than 21 is specified, the packet is assumed 21-word long. Bit[15:0] is used to name the receiver by its integral Node ID.) After writing the header word to the head of the send queue, the user process can then store the content of the outbound message in the successive word addresses following `(int*)Send_Rp_Q_base` + `(Send_Rp_Q_head` << 5). There is no restriction on the particular order that the data and the header are written, as long as the user process does not make the current packet visible to FUNi until the packet has been completely composed.

After the packet is completely composed, the user process should record the event by incrementing `Send_Rp_Q_head` modulo the queue size. The packet enqueued does not become visible to FUNi's retrieval transaction until the FUNi RPSQ_HD register is updated with the new incremented `Send_Rp_Q_head`. If the user wishes to minimize the communication latency, the user process should update the FUNi RPSQ_HD register with the incremented value of `Send_Rp_Q_head` immediately after each enqueue. However, when a large number of packets are sent in succession, such as in a block data transfer, the packets can be grouped into sub-blocks to be sent in a pipelined fashion. The user process only needs to update the FUNi RPSQ_HD register once for each sub-block. This optimization amortizes the overhead cost of updating the FUNi RPSQ_HD register over the sub-block without seriously affecting the latency of large data transfers.

Earlier in this section, we stated that when the send queue is full, the user process should wait for FUNi to make more room. However, the disadvantage to this simplistic solution is the large additional overhead due to idle cycles if no useful work is found while waiting. As an alternative to waiting, the user can allocate another block of memory of the same size as the current queue and use the newly allocated memory as a virtual send queue. The user process needs to maintain a separate head index for this virtual queue. When FUNi has emptied the real send queue (i.e., when

FUNi RPSQ_TL == Send_Rp_Q_head), the user can switch in the virtual send queue as the new send queue. The user only needs to update the FUNi RPSQ_HD register and Send_RP_Q_head with the head index kept for the virtual queue to continue normal operations with the new send queue.

The following is a simplified code section that demonstrates sending a 5-word data packet of type TYPE to the processor PROC. The packet contents are d0,d1,d2,d3,d4. The program uses the same program variables as described in this section. The code does not make use of any of the optimization tricks. It is meant to be a straight forward example. For a more elaborate example, please refer to the C source code for the ported active message communication primitives included in Appendix A.

```
{
  int *addr;

  /* wait until there is enough room in the send queue */
  while ((*(int *)(mm_FUNi_base+(NIO_RPSQ_TL<<NIO_REG_OS)))==
         ((Send_RP_Q_head+1)&Q_mask));

  addr=(int *)Send_Rp_Q_base+Send_Rp_Q_head<<5;

  /* header for 5-word packet to PROC */
  *(addr++)=((TYPE<<21) |
             (5<<16) |
             (PROC));

  /* 5 data words in successive addresses */
  *(addr++)=data0;
  *(addr++)=data1;
  *(addr++)=data2;
  *(addr++)=data3;
  *(addr)=data4;

  /* increment head index */
  Send_RP_Q_head=(Send_RP_Q_head+1)&Q_mask;

  /* update head register to send packet */
  *(int *)(mm_FUNi_base+(NIO_RPSQ_HD<<NIO_REG_OS))=Send_RP_Q_head;
}
```

3.2.3 Receiving Network Messages

The receive operation is the exact complement of the operation for sending a packet. The receive operation is the simpler of the two operations since the user process plays the role of the consumer. When a packet arrives from the network, an initialized FUNi uses DVMA to deliver the packet to one of the two receive queues according to the packet's priority. With just normal memory references, the user can then receive the packet from the receive queue. The following paragraphs describes the steps involved in polling and receiving a reply packet. The steps involved in receiving a request packet are analogous except for the substitution of the corresponding registers and queue.

Again, let us assume `(int*)Rcv_Rp_Q_base` is a program variable pointing to the base of the reply receive queue, and the variable `(int)Q_mask` holds the value, one less than the number of slots in each queue, that was assigned to the FUNi Q_MASK register. Let us also assume `(int)Rcv_Rp_Q_tail` indexes the next slot to dequeue from. (This variable should be initialized to 0x0 when the reply receive queue is initialized). Please refer to Figure 3.3 for the relationship among the different software and hardware pointers and indices.

The receive operation begins with polling for new packets. FUNi RPRQ_HD holds the index to the next empty slot where FUNi will write to. The user can detect the availability of unreceived packets by comparing the content of FUNi RPRQ_HD with `Rcv_RP_Q_tail`. If the two indices are equal, then there are no unread packets in the receive queue. However, if `Rcv_RP_Q_tail` is logically less than FUNi RPRQ_HD, then the slots between the two indices contain live packets that are ready to be received.

To receive the packet in the tail slot indexed by `Rcv_RP_Q_tail`, one first reads the header word from the first word in the slot at the address `(int*)Rcv_Rp_Q_base` + (`Rcv_Rp_Q_tail`<<5). The header has the same format as the header used for sending, except for the destination field bit[15:0] which is set to the receiving workstations own node ID.

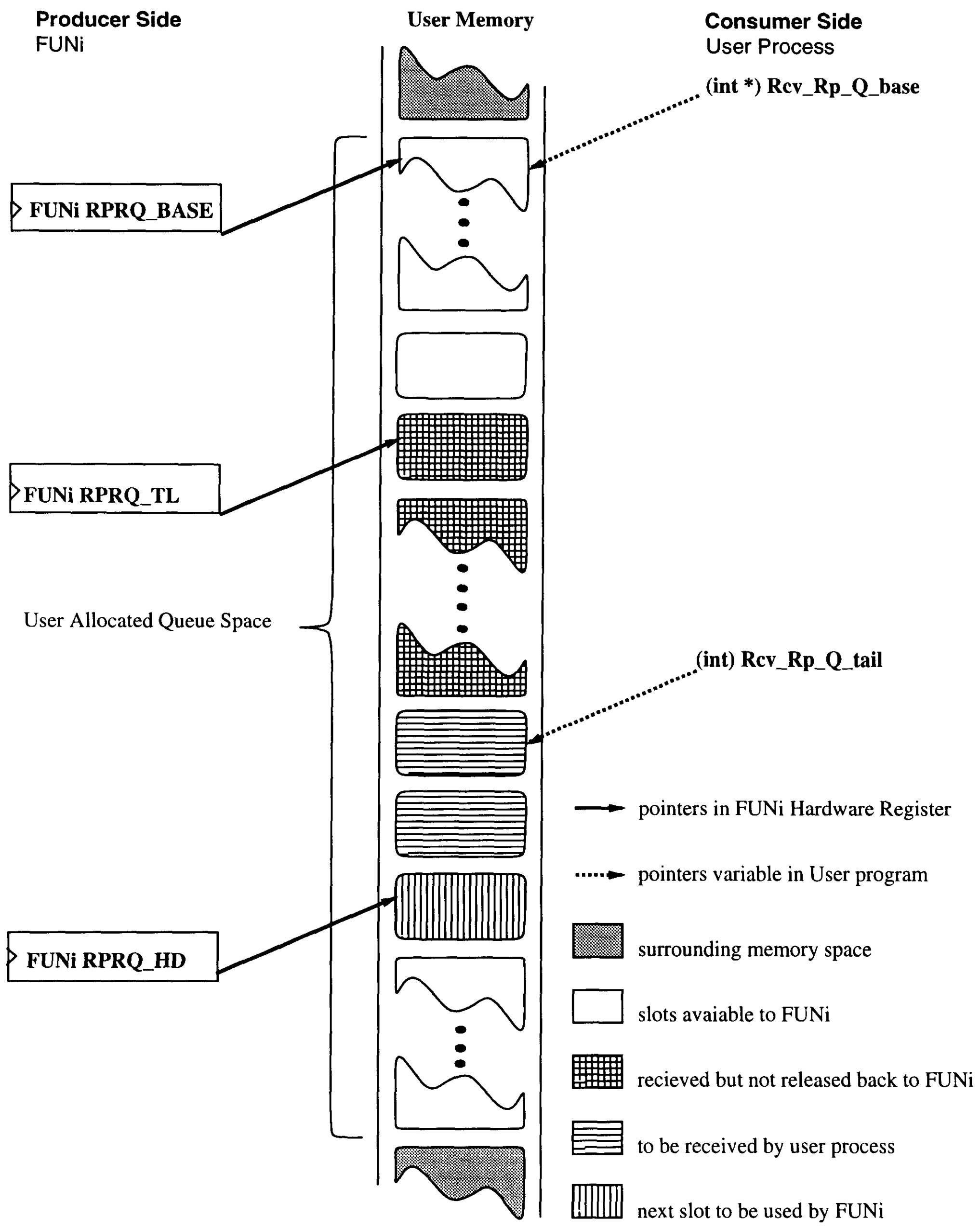
Producer Side
FUNi
User Memory
Consumer Side
User Process
(int *) Rcv_Rp_Q_base
FUNi RPRQ_BASE
FUNi RPRQ_TL
User Allocated Queue Space
(int) Rcv_Rp_Q_tail
FUNi RPRQ_HD
pointers in FUNi Hardware Register
pointers variable in User program
surrounding memory space
slots avaiable to FUNi
recieved but not released back to FUNi
to be received by user process
next slot to be used by FUNi

Figure 3.3: Organization of Reply Receive Queue

After parsing the header to determine the nature and the size of the packet, the user process can retrieve the content of the packet from the successive word addresses that follow the header. After completely retrieving the content, the user program increments Rcv_RP_Q_tail by one slot to record the receive. FUNi, as the producer in the receive operations, compares FUNi RPRQ_HD against FUNi RPRQ_TL to determine whether the head slot is free. Therefore, the user process needs to update FUNi RPRQ_TL with the incremented value of Rcv_RP_Q_tail to release the slots occupied by the received packets. For large queue sizes, the user process only needs to update FUNi RPRQ_TL as frequently as necessary to allow FUNi's delivery transaction to progress.

The following is a simplified code section that demonstrates polling and receiving a reply packet into an integer array packet[22]. The code uses the same program variables as defined above. Again, the code is meant to be a straight forward example and does not make use of any optimization tricks. For a more elaborate example, please refer to the C source code for the ported active message communication primitives included in Appendix A.

```
{
  /* check to is if the queue is not empty */
  if (Rcv_Rp_Q_tail!=
      *(int*)(mm_FUNi_base+(NIO_RPRQ_HD<<NIO_REG_OS))) {

    /* not empty */

    int *addr=(int*)Rcv_Rp_Q_base+Rcv_Rp_Q_tail<<5;
    int header=*addr;                   /* retrieve header */
    int length=NIH_DEC_LEN(header);     /* extract length */
    int i;

    if (header&NIH_XFER_MASK) {
      /* receive packet */
      for(addr++,i=0;i<length;i++,addr++) {
        packet[i]=addr;
      }
    }
```

```
      Rcv_RP_Q_tail=(Send_RP_Q_head+1)&Q_mask;

      /* update tail register to release slots */
      *(int *)(mm_FUNi_base+(NIO_RPRQ_TL<<NIO_REG_OS))=Rcv_RP_Q_tail;
    }
  }
```

3.2.4 XFER Mode: Directly Memory Accesses Message Interface

The difference between XFER-type packets and message-passing packets is where FUNi retrieves and delivers the contents of the packets. As illustrated in Case 1 of Figure 3.4, for message-passing packet, the sending FUNi retrieves the content of the outbound packet from the send queue, and the receiving FUNi delivers the content of the received packet into the receive queue. Whereas, for direct memory-to-memory XFER packet (shown by Case 2 of Figure 3.4), the sending FUNi gathers the payload of outbound XFER packet from the source location. The receiving FUNi delivers the payload of the XFER packet directly to the destination location specified by the sender.

The following paragraphs describes the usage of sending and receiving a reply-priority XFER packet. The steps for sending the lower-priority request XFER packets are identical except for the substitution of the corresponding registers and queues.

Similar to sending a message-passing packet, the sending process must first check for an available slot before enqueuing a XFER request block. Each XFER packet requires a full packet slot even though only a four-word XFER request block is entered. Prior to enqueuing each XFER request block, the sending process makes sure FUNi RPSQ_TL != ((Send_Rp_Q_head + 1) & Q_mask). Then, the sending process stores the header for the XFER packet at (int*)Send_Rp_Q_base + (Send_Rp_Q_head<<5) as usual. The header word for the XFER packet uses the same format as the message-passing packets, except for bit[31] which needs to be b'1

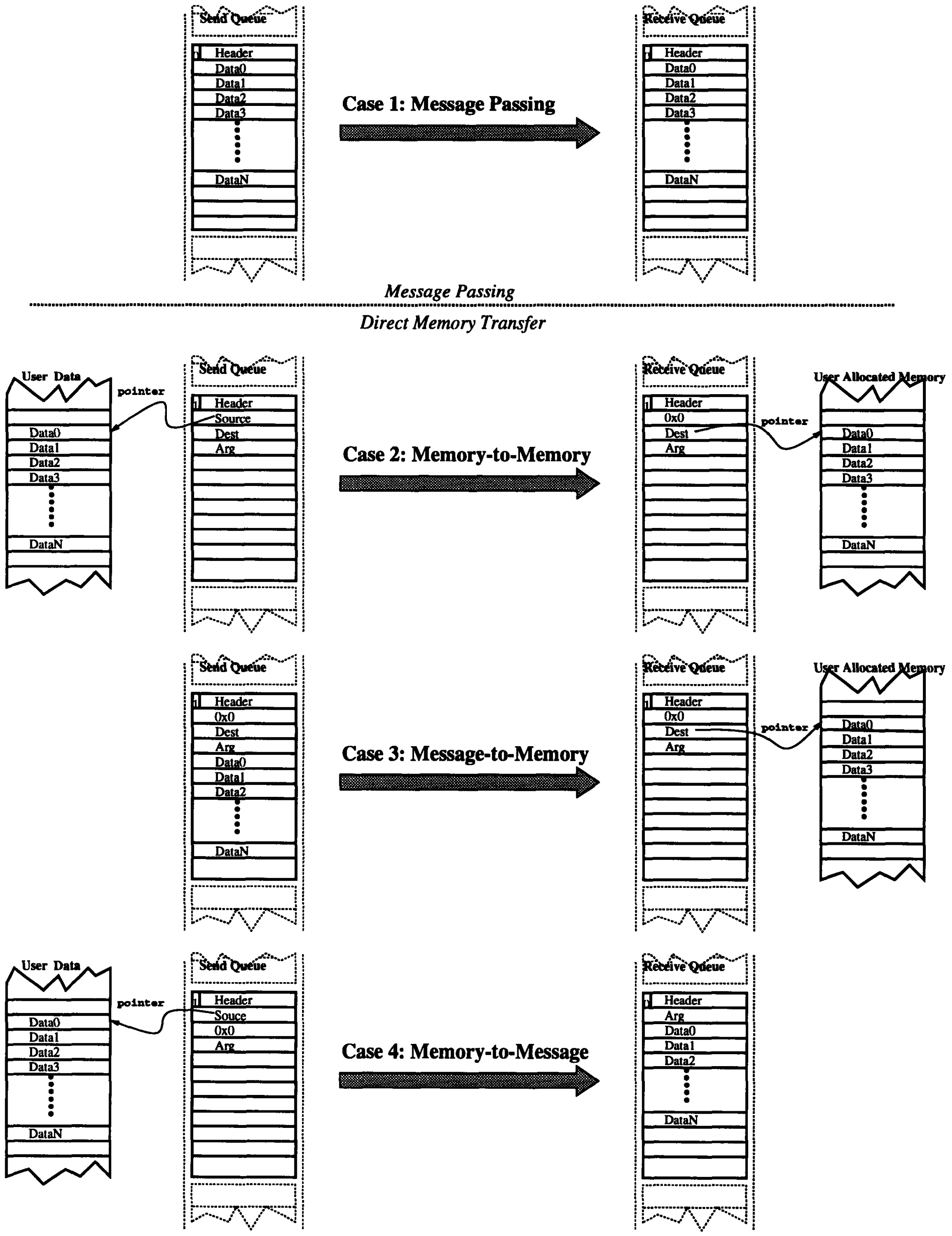

Figure 3.4: FUNi Block Data Transfer Modes

to instruct FUNi to interpret the packet slot as a XFER request. Bit[20:16] of the header word specifies the number of words this packet will transfer. The size of the transfer per XFER packet must be a multiple of the DVMA burst-transfer size (4 in this case) and cannot be more than 16 words. Automatic packetization for larger block transfers is not implemented. For message-passing packets, the sending process, in the following steps, would write the content of the packet in the successive addresses. However for XFER, the sending process only needs to write, in the next word address, the base address for the source data block. When FUNi processes this XFER request block, FUNi will fetch the specified number of words from the specified source address. In the next word of the XFER request block, the user process provides the remote destination address of the transfer. To simplify the implementation of the network hardware, both the source address and the destination address must be four-word aligned. (Misaligned addresses will be automatically truncated). Finally to completed the XFER request block, the user can provide a single argument, usually a transfer ID, that will be carried with the XFER packet and will appear in the receive queue of the destination workstation.

For coherence of data, the source for the data transfer must not be modified until the sender is certain that FUNi has retrieved the data. The user process can examine the FUNi RPSQ_TL register to check the progress of FUNi and determine whether the data have been retrieved yet. If the source is modified before FUNi retrieves the data, the transfer will reflect the modifications.

Figure 3.4 illustrates three XFER modes utilizing DMA. We have described the generic memory-to-memory XFER mode. FUNi provides another XFER mode (shown by Case 3 of Figure 3.4) in which it retrieves the transfer data from the send queue so that the user may compose the body of the packet in the send queue as in message passing and invoke only the DMA feature on the receiving node. To use this option, the sending process would specify the source address to be (int*)0x0 in the XFER request block and enqueue the transfer data in the addresses following the XFER request block, i.e. (int*)Send_Rp_Q_base + (Send_Rp_Q_head<<5) + 4.

By specifying (int*)0x0 as the destination address, the user can invoke the third mode to use DMA to compose an outbound packet and have the packet emerge at the receiving node as a message-passing data packet. (Please see Case 4 in Figure 3.4.) In this mode, the delivered packet is composed of the single word argument that the sender provides in the XFER request block, followed by the payload data.

At the receiving node, the transfer process is nearly transparent to the user code. When FUNi receives a XFER-type packet from the network, the transferred data is written directly to the destination address specified in the XFER packet. A four-word XFER notice block is enqueued into the receive queue of the corresponding priority. The first word is the header word which uses the same format as the receive header for message passing except bit[31] is b'1 to distinguish the header as the header of an XFER packet. The word immediately following the header is always set to zero. The next word is the destination address that the transfer data is delivered to. Finally, the fourth word holds the single argument that the originator of the transfer provided.

The following is a section of code that demonstrates a memory-to-memory data transfer from the source address SOURCE to the destination address DEST on the workstation RCVR. A total of 4*L words is transferred via possibly multiple XFER packets. All transfers are of type TYPE and carry ID as their argument.

```
{
  int *addr;
  int toGo,*source,*dest;
  int last;

  /* initiate data transfer in 16-word XFER packets */
  for(toGo=4*L,source=SOURCE,dest=DEST;
      toGo>=16;
      toGo-=16;source+=16;dest+=16) {
    /* wait until there is enough room in the send queue */
    while ((((*(int *)(mm_FUNi_base+(NIO_RPSQ_TL<<NIO_REG_OS))+1) &
           Q_mask) ==
          Send_RP_Q_head)

    addr=(int*)Send_Rp_Q_base+Send_Rp_Q_head<<5;
    /* header for 16-word XFER packet */
    *(addr++)=(0x80000000 |
               (16<<16) |
               (TYPE<<21) |
               (RCVR));
    *(addr++)=source;
    *(addr++)=dest;
    *(addr)=ID;


    /* increment head index */
    last=Send_RP_Q_head=(Send_RP_Q_head+1)&Q_mask;

    /* update head register to send packet */
    *(int *)(mm_FUNi_base+(NIO_RPSQ_HD<<NIO_REG_OS))=Send_RP_Q_head;
  }

  /* transfer remaining words (< 16 words) */
  if (toGO) {
    /* wait until there is enough room in the send queue */
    while (((*(int *)(mm_FUNi_base+(NIO_RPSQ_TL<<NIO_REG_OS))) &
            Q_mask) ==
           Send_RP_Q_head);

    addr=(int*)Send_Rp_Q_base+Send_Rp_Q_head<<5;

    /* header for the last XFER packet */
    *(addr++)=(0x80000000 |
               (toGo<<16) |
               (TYPE<<21) |
               (RCVR));
    *(addr++)=source;
    *(addr++)=dest;
    *(addr)=ID;

    /* increment head index */
    last=Send_RP_Q_head=INC_BY(Send_RP_Q_head,1,Q_mask);

    /* update head register to send packet */
    *(int *)(mm_FUNi_base+(NIO_RPSQ_HD<<NIO_REG_OS))=Send_RP_Q_head;
  }
}
```

3.3 Special Purpose Interface

3.3.1 Route Table RAM

In the FUNi programming model, remote workstations are named by an abstract integral node ID. FUNi uses a route table to translate the integral ID to the 30-bit network route bits. Two identical route tables are implemented as a pair of RAM's on the FUNi SBus card. The operating system can load the contents of the RAM's using memory-mapped writes to the memory-mapped page address associated with the route table RAM's. In a write, the lower thirty bits of the data word are stored into the RAM's at the address specified by the page offset of the memory-mapped write's target address.

3.3.2 Interrupt

In the current implementation plan, FUNi will only interrupt when an inbound packet is tagged with the privileged system-level GID. The system-level packet is still rejected if the FUNi GID register does not contain the privileged system-level GID. After the interrupt, the system's interrupt handler should then initiate a context switch to gain possession of FUNi in order to receive the retransmitted packet. This special class of interface is for the operating system alone. A user process must use polling to detect the presence of new packets. We will not support a general interrupt-driven operation because the existing operating system for SPARCstations does not support user-level interrupts. An interrupt-driven message interface that uses system-level interrupts for user packets would result in an even larger overhead than polling.

Chapter 4

Time-sharing Support

FUNet is designed to allow multiple parallel applications to time-shared the network and processor resources while providing each application the illusion of a dedicated and reliable network. The following sections describe the mechanisms that allow FUNet to expose its hardware to direct user-level access without sacrificing the necessary protection and security. Then, a general guideline for secure, user-transparent context switching of FUNi is specified. Finally, a cluster scheduling strategy that can make efficient use of a FUNet cluster is suggested.

4.1 Protection Mechanisms

The primary concern on network security is the privacy of data. With different application contexts sharing various parts of the FUNet resources, we need to have a mechanism to prevent one application context from accidentally, or consciously, seeing the private communication of another context. Similarly, an application context must not be able to falsify a delivery to another context.

4.1.1 Network Security

To avoid the overhead from system-level intervention, data protection for user-level network communication is implemented in the FUNi hardware. The operating systems on all participating nodes of FUNet collectively assign a unique eleven-bit Group Identifier (GID) to every executing parallel application on FUNet. When the process of a parallel application is switched in on a workstation, the operating system writes the corresponding GID into the FUNi GID register as part of the context switch. During the time-slice of the process, every outbound packet is automatically tagged with the content of the FUNi GID register. When the packet arrives at the destination, the receiving FUNi compares the GID tag of the inbound packet against the local FUNi GID register. The inbound packet is delivered to the executing process only if a match is made. In the case of a GID mismatch, meaning the correct receiving process is not presently executing, the inbound packet is not delivered. Under the acknowledge/retry end-to-end flow control protocol, FUNi will drop the undeliverable packet and return a negative acknowledgment to the packet's originator. Thus, a process is only able to communicate with its peer processes who shares the same GID.

4.1.2 Privileged Access to FUNi Hardware

The hardware protection scheme for network security is based on the content of the FUNi GID register. To maintain the authenticity of the network GID tags, only trusted system-level processes are allowed to write to the FUNi GID register. FUNi needs a way to differentiate between the user- and system-level memory-mapped accesses to protect the FUNi GID and other privileged control registers. This can be achieved by using the virtual to physical address translation as an access barrier.

A memory-mapped address is composed of two parts, the page address and a page offset. During a memory-mapped access, the memory-management system translates the memory-mapped page address into a physical page address. If the physical page address maps to the FUNi SBus device, the physical page address, concatenated with the original page offset, is passed to the FUNi SBus device as part of the bus transaction.

The SBus allows multiple physical page addresses to map to a single device. The FUNi SBus device can associate the address of a particular page with user-level accesses and the address of another page with system-level accesses. With exclusive control of the memory-management system, the operating system can ensure that only its own system-level memory-mapped accesses are translated to the system-level physical page address. Thus, FUNi can determines the privilege level of an access by examining the physical page address of that access.

4.2 Context Switch

While context switching parallel applications on a FUNet processing node, the operating system must also swap the FUNi hardware states. With the send and receive packet queues residing in the user's virtual memory, the amount of FUNi hardware states that need to be swapped – sixteen hardware registers plus two small hardware packet FIFO's – is fixed. The operating system needs to carefully follow the guidelines for setting and checking the bits in the FUNi CNTL register when swapping the FUNi hardware registers. (Section 3.1.2 describes the semantics of the bitfields in the FUNi CNTL register.)

To initiate a context switch, the operating system sets the CTXM bit of the FUNi CNTL register after first saving the original content of the FUNi CNTL register. FUNi must be given time to bring itself into a state that can be swapped without corruption. CTXR1 of the FUNi CNTL register indicates whether FUNi's retrieval and dispatch transaction units have arrived at a swappable state. The operating system must not modify the FUNi GID or the FUNi TICKET register until CTXR1 becomes set. When CTXR1 is set, the operating system can also swap other FUNi registers that are associated with the send queues.

The FUNi registers associated with the receive queues must not be modified until CTXR2 of the FUNi CNTL register becomes set. The CTXR2 status bit indicates the readiness of the FUNi delivery transaction unit. A little coercion by the operating system may be necessary to bring about the setting of CTXR2 during a context switch.

When the Context-Switch mode is enabled, FUNi will reject all further inbound packets. However, FUNi's hardware receive buffers may contain some already accepted packets that are waiting to be delivered to the user's receive queues. The buffers, containing hardware states, must be drained before FUNi can set CTXR2. In an attempt to empty FUNi's hardware receive buffers, the delivery transaction continues to operate during the Context-Switch mode. However, the operating system cannot rely on the delivery transaction alone to drain the buffers. If the user's receive queues were full or if the user had set the CRQM bit before context switch began, the delivery transaction would not be able to move the buffered packets out to memory. Therefore, the operating system must explicitly set up a new pair of receive queues with sufficient space to drain the contents of the buffers. The queues only need to be as large as the hardware buffers. This pair of queues can be kept as part of the exiting process's context block. When the context is restored, the drained packets need to be returned onto the network.

When both CTXR1 and CTXR2 of FUNi CNTL become set, all of FUNi's transactions have halted, and FUNi will no longer initiate any more DVMA to the virtual memory of the exiting context. The operating system can now perform other non-FUNi-related context switching tasks. To conclude the context switch after successfully swapping the CPU, memory-management unit and FUNi register states, the operating system restores the saved content of the FUNi CNTL register to reactivate FUNi for the next context.

A simple-minded instantiation of the context switching operation written in the C programming language is included in Appendix B.

4.3 Scheduling

Saving and restoring FUNi states significantly increases the overhead of a context switch. The granularity of existing time-sharing strategies becomes impractical because of the new larger context switch overhead. Furthermore, attempting to maintain the same fine granularity of time-sharing adds to the difficulty in gang scheduling the execution of parallel applications on a distributed system. For the execution of a thread to make progress on one workstation, most of its communication peers on remote workstations must also be executing. Otherwise, in the worst case, a parallel application could completely stall when all senders in the communication pairs are switched in only when their receivers are switched out. Synchronizing context switches across a loosely coupled parallel-processing workstation cluster is a difficult task since each workstation is running under its own operating system and under potentially different work loads of sequential programs. Time sharing parallel applications at a fine granularity that is common to the uniprocessor workstations is nearly impossible. Seemingly, one must compromise the performance and efficiency of the workstations as the granularity of time-sharing is raised to to a more manageable level. However this is not entirely true.

Since the control of FUNi does not change hand when a parallel process is displaced by another sequential process, the FUNi states do not have to be swapped out. The operating system only needs to place FUNi in Change-Receive-Queue and Change-Send-Queue modes to disable its DVMA operations while the uniprocessor processes are executing. Therefore, only the parallel applications need to time-share at a coarser granularity (on the order of seconds) to amortize the overhead of context switching and to ease the task of gang scheduling. However, during the scheduled time slice of a parallel application, each process of the parallel application can time-share its CPU with other sequential processes in finer granularity to maintain good CPU utilization.

Chapter 5

FUNi Architectural Overview

The SBus card was chosen as the target FUNi implementation primarily for the SBus's DVMA (Direct Virtual Memory Accesses) feature that is crucial to FUNi' programming interface. The ease of implementation was also a major consideration that stood in favor of the SBus. The SBus compatibility allows FUNi to work directly in a wide range of SBus-equipped workstation platforms without modification to the stock workstation hardware. Section 7.2.1 provides a discussion regarding the implementation plan of the FUNi SBus card.

The FUNi architecture can be divided into seven principal components: SBus Interface Module, FUNi Core Module, Route Table RAM's, Undelivered Packet Cache, Synchronization FIFO Group, Router Interface Module, and Differential Signal Converters. Figure 5.1 diagrams the high-level datapath of FUNi. The rest of this chapter presents the functional-level description and the preliminary implementation notes for each of the seven components.

5.1 SBus Interface Module

The SBus is a multi-mastered, virtually addressed backplane bus designed to support the I/O needs of a workstation. A bus transaction can occur in single beats of a

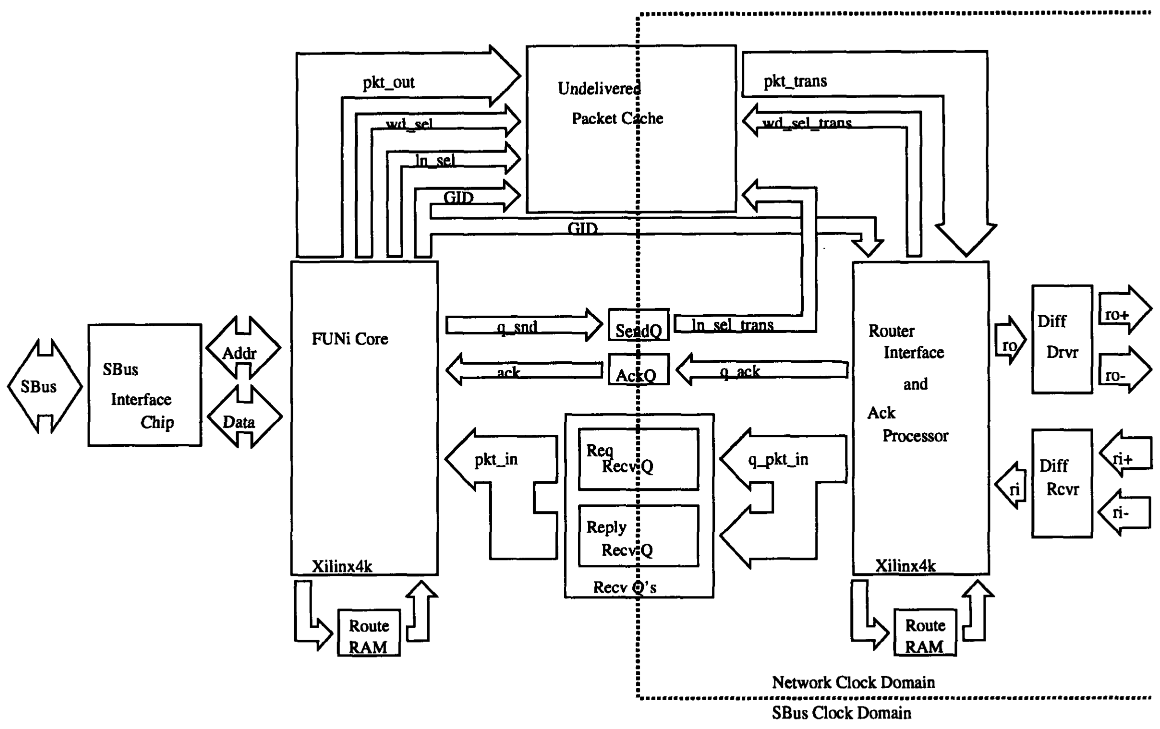

Figure 5.1: FUNi SBus Card Datapath

byte, halfword or word. Burst transfers of 2 to 16 words are also supported. Under the 25 MHz limit on the clock cycle, the architectural specification of SBus allows a hypothetical peak transfer bandwidth of 80 MByte/sec. A peak bandwidth of 72 MByte/sec can be achieved on a SPARCstation2 by performing 16-word burst transfers on a 25 MHz SBus.

Multiple master devices can coexist on a single bus. A centralized bus controller arbitrates the bus requests from the bus masters. Once control of the bus is granted, the succeeded master presents a virtual address for the bus controller to translate. After the translation, a slave device is identified as the target and a read or write bus transaction occurs between the master and slave devices. The slave device's participation on the bus is passive; the slave only reacts when selected by the SBus controller.

The main task of the SBus Interface Module is to provide the FUNi Core Module a simplifying veneer of the SBus protocol. The FUNi SBus card will be both a master and a slave device. The FUNi card behaves as a slave device in response to memory-mapped accesses from the CPU. When identified as the target slave device, the SBus Interface Module performs the necessary handshake on the SBus to accept the request and to inform the FUNi Core Module of the pending request. FUNi assumes the role of a master device to perform DVMA to access user's packet queues. The SBus Interface Module serves as an intelligent DVMA channel for the FUNi Core Module. In a DVMA transaction, the FUNi Core Module, through a simple handshake, presents a 32-bit virtual address, together with the size and the nature (read or write) of the transfer. When the request is for a write, the data words would follow. It is up to the DVMA channel to carry out the appropriate bus transaction to fulfill the request.

If the SBus Interface Module encounters a memory error on the SBus during DVMA, an interrupt is generated so the operating system can resolve the error. The SBus Interface Module should also notify the FUNi Core Module to terminate the unfinished DVMA transaction. The FUNi Core Module will set the MERR bit in the FUNi CNTL register and postpone all further DVMA transactions. The FUNi Core Module retries the failed transaction when the operating system clears the MERR bit in the FUNi CNTL register.

5.1.1 Implementation Notes

VLSI parts that meet the previous description are available from commercial suppliers. The L64853A SBus DVMA Controller is available from LSI Logic, and Motorola has announced a SBus DVMA interface controller, MC92001.

L64853A is available in a 120-ping plastic quad flat package (PQFP). Originally designed for system-level functions, the upper eight bits of virtual addresses in DVMA are forced to 0xFF. Incorporating the device into the FUNi SBus card would require board-level modifications to restore full addressability of DVMA transactions. Also, L64853A limits the available SBus bandwidth by supporting only four-word burst transfers. The 16-bit peripheral interface with FUNi further reduces the bandwidth.

MC92001 from Motorola is a complete implementation of the SBus slave and master interface plus a DVMA controller with a peak transfer rate of 160 MByte/sec (in extended mode). All transfer modes, including the 64-bit extended SBus mode, are supported. Unfortunately, at the time of this writing, the part is not available yet.

In light of the obstacles involved with incorporating commercial parts, a custom implementation may be the only viable alternative in the near future. A custom implementation fashioned after the simple and effective design of L64853A should be achievable in the current FPGA technology (over 10K available gates and 192 I/O pins on XC4013) [24]. In the custom implementation, the drawback associated with L64853A can be corrected.

5.2 FUNi Core Module

The FUNi Core Module contains the three main transaction units that coordinates packet movements through FUNi. The Retrieval Unit is responsible for retrieving the pending outbound packets from the user's two send queues, and the Dispatch Unit is responsible for scheduling sends and retries of the retrieved packets. The Delivery Unit is responsible for delivering inbound packets to the user's receive queues. The FUNi Core Module also contains sixteen memory-mapped FUNi registers. The functions of these registers are outlined in Section 3.1.2. A block diagram with the relevant signals is presented in Figure 5.2.

5.2.1 Interface Glue Unit

The Interface Glue Unit serves as the front end for the sixteen registers during a memory-mapped access from the CPU. The unit decodes the register field that is

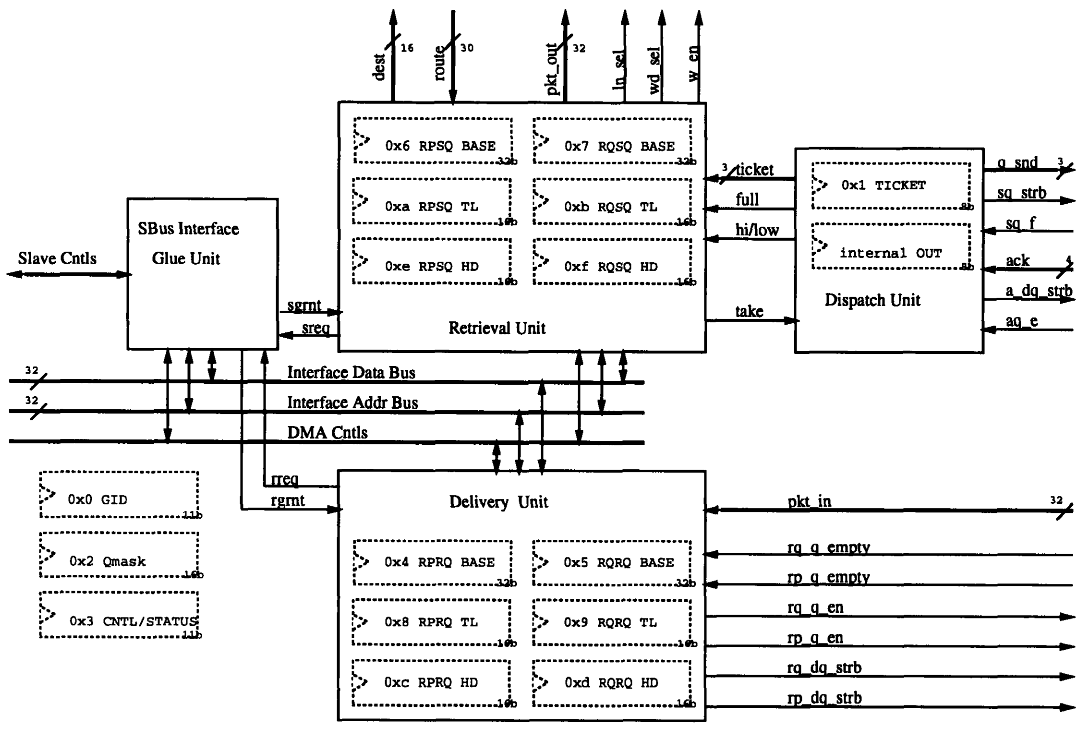

Figure 5.2: FUNi Core Module Block Diagram

encoded in the memory-mapped address, and performs the read or write request to the FUNi hardware registers. In a write access to a privileged register such as FUNi GID, before loading the designated register with the submitted value, the lowest bit of the physical page address is checked to determine the issuer's access privilege. Privileged system-level control registers are updated only if the lowest bit of the page number is b'1.

The Interface Glue Unit is also the front end for memory-mapped writes to the Route Table RAM's. Writes to the Route Table RAM's are also privileged accesses that require validation. In a validated write to the Route Table, the entire 16-bit page offset is used to index the RAM's, and the lower thirty bits of data are stored into the RAM's.

The Interface Glue Unit also arbitrates for the use of the **Interface Data and Address Bus**. The Retrieval and Delivery Units must request for the bus by asserting their ***req** signal and wait for their grant lines to be asserted before proceeding with a

DVMA access.

5.2.2 Retrieval Unit

The Retrieval Unit examines the send queue index registers to determine the presence of pending outbound packets and retrieves these packets into the Undelivered Packet Cache through DVMA.

After completing each retrieval transaction, the Retrieval Unit returns to the default state in which it repeatedly checks for one of the following two conditions:

1. FUNi RPSQ_HD!=FUNi RPSQ_TL and **full** from the Dispatch Unit is not asserted.
2. FUNi RQSQ_HD!=FUNi RQSQ_TL and both **full** and **hi** from the Dispatch Unit are not asserted.

The first condition corresponds to when at least one pending packet is waiting in the reply send queue and when there is at least one free line in the Undelivered Packet Cache. The second condition is similar to the first except it is applied to the request send queue. Note that condition 2 must also ensure **hi** is not asserted. **hi** indicates that the current cache line is only for high priority reply packets.

When one of the two conditions is met, the Retrieval Unit starts a transaction for retrieving a packet from the corresponding send queue into the available line in the Undelivered Packet Cache. If both conditions are met, precedence is given to the reply packet. In any case, if the CTXM or the CSQM bit of the FUNi CNTL register is set, the Retrieval Unit remains in the default state.

When a transaction starts, the Retrieval Unit asserts **take** to lock the current cache line selection on **ln_sel**. **sreq** is asserted to request the Interface Glue Unit for a DVMA read.

When the Retrieval Unit is granted control of the **Interface Bus** by **sgrnt**, the Retrieval Unit de-asserts **sreq** and presents FUNi R*SQ_base + (FUNi R*SQ_TL<<5) on the address bus to read the first four words from the tail slot of the send queue. When the DVMA read returns, the first word – corresponding to the header word of the queued packet – is parsed for the transfer mode, length, packet type and destination of the packet. From the parsed information, three network header words are generated and stored into the Undelivered Packet Cache.

If the parsed header indicates that the current packet is a message-passing packet, the next three words from the DVMA read are stored in the Undelivered Packet Cache as the message content. If the length of the packet is greater than three, the Retrieval Unit sets an internal counter to three and asserts **sreq** to request for another DVMA read for the remainder of the message. The length of this DVMA will vary depending on the availability of the burst mode and the length of message body. The Retrieval Unit repeats DVMA reads until the entire queued message has been moved into the Undelivered Packet Cache.

If the parsed header belongs to a XFER request block, the second word of the DVMA read is latched as the source address of the XFER packet. If the second word contains the pointer 0x0, the XFER packet payload will be loaded from the send queue following the XFER request block. The third word is checked to determine the destination address. If the destination supplied is not 0x0, the third and fourth words are stored into the Undelivered Packet Cache as the XFER header words. However, if the third word equals 0x0, indicating the current packet should be delivered as a normal message-passing packet, new network headers are generated. The new network header words, plus the fourth word of the XFER request block are stored into the Undelivered Packet Cache, overwriting the original header words for XFER. After composing the header, the Retrieval Unit performs DVMA reads from the source addresses to gather the XFER payload.

When the retrieval into the Undelivered Packet Cache is complete, the Retrieval Unit de-asserts **take** to release the current cache line to the Dispatch Unit for dispatching. The transaction is officially completed when the Retrieval Unit increments the corresponding FUNi R*SQ_TL register to record the transaction.

5.2.3 Delivery Unit

The Delivery Unit delivers inbound packets, that are accepted by the Router Interface Module, to their destination location through DVMA.

After completing each delivery transaction, the Delivery Unit returns to a default state in which it repeatedly checks for one of the following two conditions:

1. (FUNi RPRQ_HD+1)&(FUNi QMASK)!=(FUNi RPRQ_TL) and **rp_q_empty** from Synchronization FIFO Group is not asserted.
2. (FUNi RQRQ_HD+1)&(FUNi QMASK)!=(FUNi RQRQ_TL) and **rq_q_empty** from Synchronization FIFO Group is not asserted.

The first condition corresponds to the case when at least one pending packet is waiting in the reply hardware receive buffer, and there is at least one available slot in the reply receive queue in the memory. The second condition is similar to the first except it is applied to request packets.

When one of the two conditions is met, the Delivery Unit will start a delivery transaction for delivering a packet from the corresponding hardware buffer into the user memory. If both conditions are met at the same time, precedence is given to the reply packet. In any case, if the CTXM or the CRQM bit of the FUNi CNTL register is set, the Delivery Unit stays in the default state.

If a transaction is started, the Delivery Unit will select the appropriate FIFO to drive the **pkt_in** data bus by asserting either **rq_q_en** or **rp_q_en**. The network header words are strobed into the Delivery Unit by toggling the appropriate **r*_q_strb** signal. The network header words are parsed to determine the transfer mode, length, and packet type. A packet header for the receive queue is constructed from the parsed information.

If the parsed header indicates a message-passing packet, the Delivery Unit initiates a DVMA transfer into the receive queue by first asserting **rreq** to request the Interface Glue Unit for a DVMA write. When the Delivery Unit is granted control of the **Interface Bus** by **rgrnt**, the Delivery Unit de-asserts **rreq** and presents FUNi R*RQ_base + (FUNi R*RQ_HD<<5) on the address bus to write the queue packet header followed by the message content into the head of the user's receive queue. The Delivery Unit may repeat DVMA several times depending on the burst modes available.

However, if the parsed header indicated a XFER-mode packet, the required processing is more complicated. There are two additional words of the XFER headers following the normal network headers. The first of the two XFER header words needs to be latched as the destination address for the transfer. The second of the two headers is the single word argument allowed in XFER-mode packets. The Delivery Unit generates a four-word XFER notice block composed of: 1. the queue packet header, 2. (int *)0x0, 3. the destination address, and 4. the argument word. Next, the Delivery Unit requests a DVMA write to store the XFER notice block to the head of the receive queue at address FUNi R*RQ_base + (FUNi R*RQ_TL<<5). Except when the DRAIN bit in the FUNi CNTL register is set, after storing the header block to the receive queue, the Delivery Unit strobes in the payload of the transfer and writes them to their destination address by one or more DVMA steps. When the DRAIN bit is set, the Delivery Unit will ignore the destination address and transfer the payload into the receive queue following the XFER notice block instead.

Once the current packet has been completely written to the user memory, the Delivery Unit completes the transaction by incrementing the corresponding FUNi R*RQ_HD register to record the transaction. However, as described in Section 3.1.2, since the FUNi R*RQ_HD registers are cached status register, the change in the hardware register will not be visible to the user process running on the CPU.

The Delivery Unit must initiate one additional DVMA write to the cached memory-mapped address of FUNi R*RQ_HD to invalidate the stale cached value.

5.2.4 Dispatch Unit

The Dispatch Unit schedules cached packets for transmission and conducts the hardware acknowledgment/retry protocol. Within the Dispatch Unit, there is the 8-bit FUNi TICKET register that bitmaps the availability of the Undelivered Packet Cache lines. **hi** is asserted when the lower six bits of the FUNi TICKET registers are set to indicate that only the cache lines reserved for reply-priority packets are available. Only reply packets are cached while **hi** is asserted. **full** is asserted when all eight bits of the register are set to indicate that all eight Undelivered Packet Cache lines are occupied. No more packets can be cached until some cached packet have been positively acknowledged.

By using a priority encoder, the Dispatch Unit continuously drives the position of the lowest b'1 bit in the FUNi TICKET register through a flow-through latch onto **ticket**. When the Retrieval Unit asserts **take**, the latch is closed to lock the last position on **ticket**. On the high-to-low transition of **take** after the Retrieval Unit has completed the transfer into the cache line indexed by the locked **ticket**, the bit position in the FUNi TICKET register corresponding to the locked **ticket** is cleared to mark the cache line as occupied.

Along with the 8-bit auxiliary FUNi TICKET register, there is also an internal 8-bit FUNi OUT register that is not visible to the user. When a cache line has been selected for dispatch, its corresponding bit in the FUNi OUT register is cleared. The bit will remain cleared until the packet's acknowledgment packet returns.

The Dispatch Unit will check each of the eight bit positions of the FUNi TICKET and FUNi OUT registers one-by-one. For bit positions where the ticket bit is set, no action is taken because the corresponding cache lines do not contain live data.

If the ticket bit and the outstanding bit are both cleared, the packet held by the corresponding cache line has already been dispatched for transmission, and thus no action can be taken. If the ticket bit is cleared and the outstanding bit is set, the packet held by the corresponding cache line is pending delivery, but not dispatched. If **sq_f** is not set, the Dispatch Unit will present the bit position on **q_snd** and toggle **sq_strb** to dispatch the cache line for transmission. When a packet is dispatched, the Dispatch Unit clears the corresponding bit position in FUNi OUT to record the dispatch.

The Dispatch Unit constantly checks **aq_e** for returned acknowledgments in the Acknowledgment Queue. When **aq_e** is de-asserted, the four-bit value on **ack** is examined. The first three bits indicate the cache line that is acknowledged. The fourth bit indicates whether the acknowledgment was positive or negative. The bit in the FUNi OUT register that corresponds to the returned acknowledgment is set to record the return. If the acknowledge is positive, the corresponding bit in the FUNi TICKET register is also set to free the cache line occupied by the acknowledged packet. Otherwise, the FUNi TICKET register is left unchanged so that the cache line becomes a candidate for dispatch again.

5.2.5 Implementation Notes

The FUNi Core Module is the most complicated piece of custom logic on the FUNi SBus Card. The implementation of this module abandons the traditional schematic capturing process. Instead, designs will be entered in Verilog Hardware Description Language to be compiled into the appropriate netlists by Synopsys HDL Compiler. The current plan calls for implementation of the module in the Xilinx 4000 Family of Field Programmable Gate Arrays (FPGA).

This module will most likely need to be partitioned, according to the subunit boundary, for implementation as a multiple FPGA chip set. Fortunately, the Delivery Unit and the Retrieval Unit have no interaction except for the sharing of the **Interface**

Bus, and the Dispatch Unit's interaction with the Retrieval Unit requires only six signals. Partitioning the module into a chip set should not pose any great obstacle. The Interface Glue Logic needs to be replicated in each FPGA in the multi-chip implementation to serve as the front-end of the memory-mapped registers within each unit.

The idea of using FPGA instead of Application Specific Integrate Circuit (ASIC) for the implementation of custom logics has existed since the beginning of the design. For that reason, the design of FUNi has evolved toward simplicity in favor of the FPGA implementation. Some of the features implemented in the current FUNi design may appear incomplete or awkward. Nonetheless, the low cost and ease of revision offered by FPGA is a tremendous asset to a prototype system intended for a proof of concept.

5.3 Multi-context Undelivered Packet Cache

The Multi-context Undelivered Packet Cache holds outbound packets until they are positively acknowledged by their recipients. The Retrieval Unit retrieves pending outbound packets from the send queues in user memory into the cache. The usage of the cache is managed by the Dispatch Unit with the help of the FUNi TICKET register. The cache is multi-contexted so cached packets do not need to be flushed between context switches.

The cache will be implemented as two side-by-side IDT7025S/L 8Kx16 dual-port static RAM's to achieve the 32-bit datawidth. The 8K words of the cache, addressed by 13 address bits, will be logically organized into 32 contexts of eight lines each. Each line will be 32 words long, enough to hold the packet with the maximum packet length.

IDT7025S/L is dual-ported with separate data and address pins to allow the Retrieval Unit and the Router Interface Module completely independent and asyn-

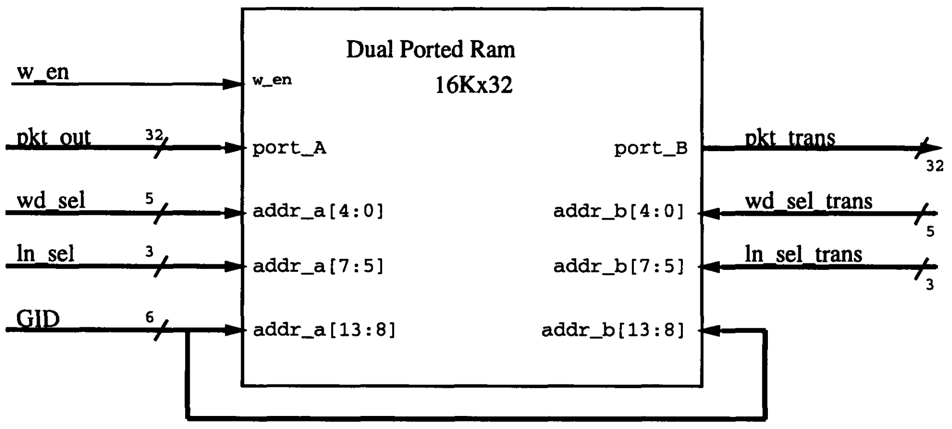

Figure 5.3: FUNi Undelivered Packet Cache Block Diagram

chronous accesses on their private ports. The upper five address bits of both ports are hardwired to the lower five bits of the FUNi GID register to select the current context.

The Router Interface port is hardwired for read only. Bit[7:5] of the address pin is hardwired to **ln_sel_trans** and is driven by the output of the Send Queue to select a cache line. A counter in the Router Interface Unit drives **wd_sel_trans**, which is connected to bit[4:0] of the address bits, to read the content of a cache line sequentially during a transmission.

Similarly, the address bits of the Retrieval Unit port are divided into word and line selects and are driven by **wd_sel** and **ln_sel**. The Retrieval Unit must also drive **w_en** which controls the write enable of the Retrieval Unit port. When writing into the RAM, **w_en** is asserted only during the first phase of a clock and is purposely delayed relative to the clock transition. Address transitions are allowed only on the first edge of the clock and must settle before the delayed **w_en** signal is asserted. This allows consecutive writes to occur back to back in every cycle.

5.4 Route Table RAM

Each Arctic network packet header carries thirty bits of routing information. In the maximum network configuration, 2^{16} nodes can be connected in the network. The packet header from the user program names the destination workstation abstractly via an integral ID. Before the packet is transmitted, the integral ID needs to be converted into the corresponding route bits that the network routers understand.

When formatting the network packet, the Retrieval Unit drives the node ID to the Route Table RAM to generate the route bits. The Router Interface Module uses a second identical Route Table RAM to generate the route bits for returning acknowledgment packets. The Interface Glue Unit in the FUNi Core Module loads the contents of the RAM's in response to the memory-mapped writes from the operating system.

The specification defines a 64K-by-30-bit route table to allow full compatibility and utilization with the Arctic router. However, it is unlikely any FUNet cluster will approach the 64K node size. The configuration which FUNet is interested in, does not require all thirty bits of route information. Thus, in an actual implementation, a narrower Route Table with fewer entries will suffice.

5.5 Synchronization FIFO Group

The Synchronization FIFO Group is made up of four hardware uni-directional FIFO's, each allowing independent asynchronous enqueue and dequeue operations.

The Send Queue and Acknowledgment Queue are two small queues of 3 and 4 bits wide respectively. The depth of the queues is bounded by the number of Undelivered Packet Cache lines available for each context. When the Dispatch Unit schedules a packet from one of the eight cache lines for dispatching, a three-bit index of the cache line is enqueued into the Send Queue. The Router Interface Module, on the other

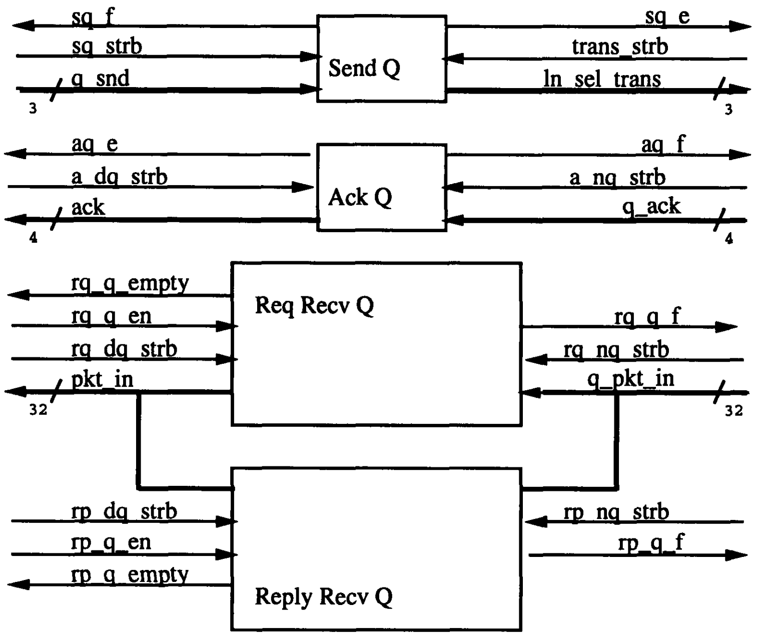

Figure 5.4: Synchronization FIFO Group Block Diagram

end, waits for the Send Queue to become nonempty. When a valid index appears at the output of the Send Queue, it selects the cached line to be transmitted by the Router Interface Module. When possible, the Transmit Scheduler in the Router Interface Unit will read the packet from the Undelivered Packet Cache for transmission on the network. After the transmission, the Transmit Scheduler dequeues the last index by toggling **trans_strb** and looks for the next cache line that is scheduled for transmission.

The Router Interface Module passes returned acknowledgments to the Dispatch Unit through the four-bit wide Acknowledgment Queue. When the Network Packet Preprocessor in the Router Interface Unit receives an acknowledgment packet, the packet is summarized into four bits, three bits for the cache line index and one bit for positive/negative. The acknowledgment is then enqueued into the Acknowledgment Queue. The acknowledgment controller in the Dispatch Unit constantly polls the output of the Acknowledgment Queue for returned acknowledgments to update the FUNi TICKET and FUNi OUT registers.

When the Router Interface Module receives a data packet, the packet is enqueued into one of the two Receive Queues according to the packet's priority. Two separate queues are required to buffer reply and request packets separately because higher priority reply packets must not be blocked by request packets. Each of the two queues is 32-bit wide. The depth of the queues is not important since the user receive queues in memory provides the main buffering. However, since the bandwidth at which the Delivery Unit can move packets out of FUNi is slower than the bandwidth of the network, the FIFO queue does need to have some buffering capacity to handle a momentary pile up of inbound packets. LH5420, the 256 by 32-bit FIFO from Sharp, allows buffering of up to ten network packets of maximum size. To prevent the FIFO from overflowing, the Router Interface Unit rejects subsequent inbound packets when the FIFO is full.

Although the FIFO's provide some buffering on the FUNi card, their primary purpose is to allow asynchronous operations at the two ends of the FUNi card. The SBus end of the FUNi card is required to execute synchronously with the SBus clock. However, one cannot reasonably require the network clock to also be synchronized with the SBus since doing so would require the synchronization of all SBus clocks on every participating workstation. The isolation provided by the FIFO's allows the Arctic Network to operate at its own maximum clock rate despite the maximum 25 MHz SBus clock limit imposed on the SBus end of FUNi. The clocking isolation also allows workstations with different SBus speed to connect to the same network.

5.6 Router Interface Module

The task of the Router Interface Module consists of three parts. First, the Transmitter and Input Port Buffers implement the necessary handshake with Arctic to transmit and receive packets on the network. Second, the Network Packet Preprocessor participates in the acknowledgment/retry flow control protocol. Lastly, the Transmit Scheduler coordinates the sharing of the Transmitter between the data

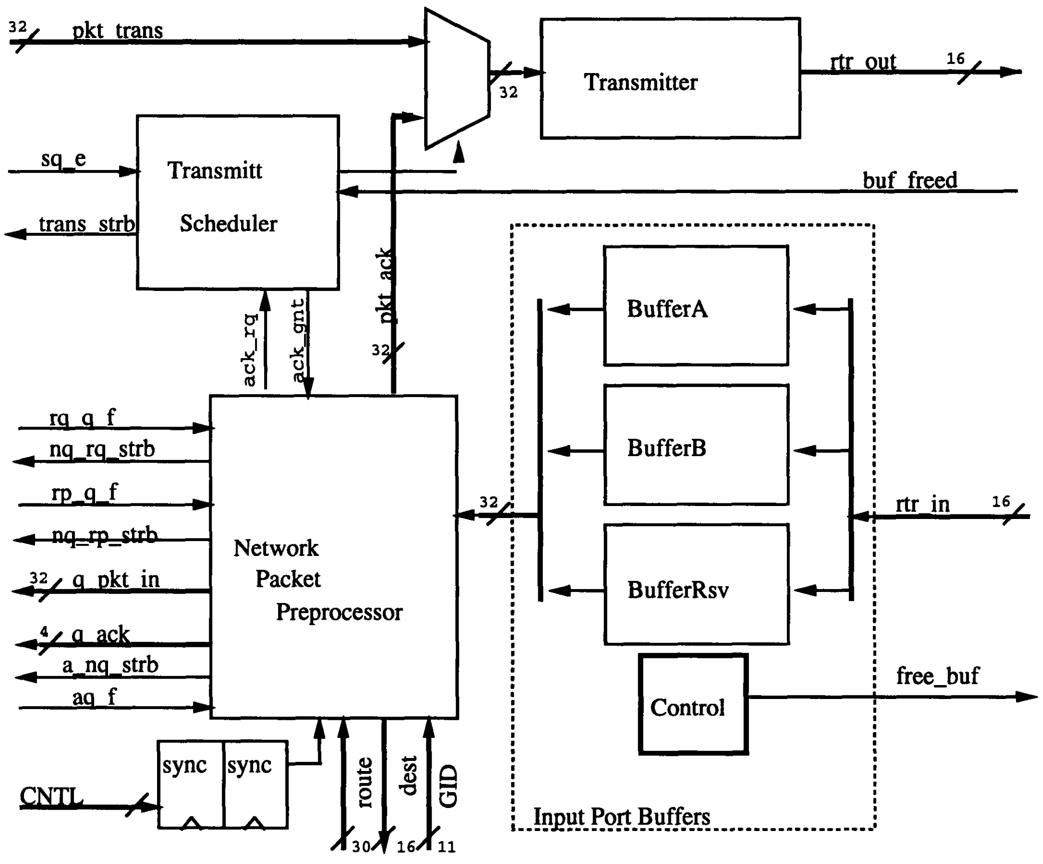

Figure 5.5: Router Interface Module Block Diagram

traffic from the Undelivered Packet Cache and the acknowledgment traffic from the Network Packet Preprocessor.

5.6.1 Input Port Buffer and Transmitter

The Input Port Buffer and the Transmitter section correspond to the input and output section of an Arctic router. They interface directly with the transmission handshake of Arctic. The Input Port Buffer provides buffering for three packets. One is reserved for high priority network packets. The output section of Arctic keeps a count of free buffers remaining on its corresponding input section. When the count equals one, only high priority packets may occupy that last buffer. The output section also uses this count to stop transmissions completely when the corresponding input section has run out of buffers. The input section informs the output section when a buffer is released so the count can be incremented.

5.6.2 Network Packet Preprocessor

The Network Packet Preprocessor carries out the low-level processing of the acknowledgment protocol. The Network Packet Preprocessor continuously checks the Input Port Buffers for the arrival of packets.

When an acknowledgment packet is found in the buffer, it is immediately processed. The first step in processing any packet is to check the GID tag of the packet against the contents of the FUNi GID register. In the case of an acknowledgment packet, mismatched GID's would indicate that either some network error has caused a misdelivery, or the operating system has, purposely or inadvertently, performed a context switch without waiting for all the acknowledgments for the exiting context. A mismatched acknowledgment is discarded which is the correct behavior if the operating system did indeed purposefully proceed through a context switch without waiting for all the acknowledgments. (Please see the description of FUNi CNTL register's CTXR1' bit in Section 3.1.2 for clarification.)

After an acknowledgment has been validated, the acknowledgment packet is parsed to extract the three-bit cache line index and to determine whether the acknowledgment is positive or negative. The acknowledgment is then summarized into four bits and driven onto **q_ack** to be enqueued into the Acknowledgment Queue.

When the Network Packet Preprocessor cannot locate an acknowledgment to process, it will, in turn, try to locate a data packet in the Input Port Buffers. When a data packet is located in the buffer, again, the Network Packet Preprocessor will first perform a check on the GID tag of the packet. If a mismatch is detected, the Network Packet Preprocessor will reject this packet. Besides from mismatched GID's, the Network Packet Preprocessor can also elect to reject inbound data packets if the

CTXM bit of the FUNi CNTL register is set, or if the synchronization Receive FIFO corresponding to the priority of the packet is full as indicated by **r*_q_f**. To reject a packet, the Network Packet Preprocessor sends a negative acknowledgment packet to the originator of the rejected packet. If none of the above conditions is met, the Network Packet Preprocessor can safely enqueue the packet into the appropriate Receive FIFO and return a positive acknowledgment packet to accept the packet.

To transmit the acknowledgment packet, the Network Packet Preprocessor asserts **ack_req** to request the Transmit Scheduler for a transmission window. Until a transmission window is granted for transmitting the acknowledgment, the Network Packet Preprocessor cannot process any more data packets from the Input Port Buffer. Returned acknowledgment packets can still be processed normally.

After processing a packet from the Input Port Buffer, the buffer occupied by the packet is released. This information is propagated back to the sending section of an Arctic router by driving **free_buf** through the correct transitions as specified by the Arctic design.

5.6.3 Transmit Scheduler

The Transmit Scheduler manages the sharing of the Transmitter between the data and the acknowledgment traffic. When the Network Packet Preprocessor requests the Transmit Scheduler by asserting **ack_rq**, the Transmit Scheduler will grant the control of the Transmitter to the Packet Preprocessor at the first opportunity. When the Transmitter has idle cycles between the transmission of acknowledgment packets, the Transmit Scheduler will examine **sq_e** to determine if any data packet are waiting for transmission. If an outbound packet is found and the Network Packet Preprocessor is not using the Transmitter, the Transmit Scheduler transmits the packet from the Undelivered Packet Cache line that is selected by the output of the Send Queue.

The Transmit Scheduler maintains a counter that is initialized with the number of buffers in the Arctic Input section that are connected to the Transmitter. Each time a packet is transferred, the count is decremented by one. Each time a correct transition appears on **buf_freed** from the Transmitter's opposite input section, the count is incremented by one. Whenever the count equals one, the Transmit Scheduler will stop scheduling data packet transmissions. Thus, the last buffer in the input section is reserved for the acknowledgment packets only. When the count equals zero, no more packet transmissions will be allowed.

5.6.4 Implementation Notes

The Router Interface Module is another piece of custom logic that needs to be implemented as FPGA. The greatest concern is the maximum clock rate that can be achieved in FPGA implementation of this complexity. As discussed in Section 5.5, the network side of the FUNi card is allowed to execute at a maximum clock rate independent of the 25 MHz SBus clock. Arctic is designed for operations at up to 50 MHz. The speed of the Router Interface Module will most likely become the limiting factor in the network clock speed. A comforting fact is that even at a comfortable 25 MHz, the Arctic Network is still capable of delivering 800 Mbit/sec per channel. This bandwidth exceeds the expected bandwidth between the CPU and FUNi.

5.7 Differential Drivers and Receivers

To overcome the effect of ground variation, differential wires will carry the signals that are running between the FUNi SBus card and the centralized network hub. The drivers and receivers are off-the-shelf parts to convert between single-ended and differential signals. At the Arctic Network Hub, another pair of drivers and receivers is used to convert the differential signals to GTL signal levels that Arctic uses.

Chapter 6

FUNet Cluster Performance Evaluation

This chapter assesses the quality of the network interface design. This assessment is based on two benchmark programs executed on a simulator of a hypothetical FUNet system. We first describe the simulator to establish confidence in the results of the experiments. Next, we explain the two benchmark programs and analyze the results of the simulations. Similar benchmarks were also executed on a successful contemporary MPP system, CM-5. We observe that FUNet cluster is able to achieve comparable processor utilization and scalability as CM-5.

6.1 The Simulator

The simulator used in this chapter is based on the PROTEUS simulation engine which allows rapid development of event-driven simulators of parallel architectures [3, 4, 5]. The PROTEUS simulation engine is a collection of C source files for an abstracted core system of a simulator. To minimize the overhead for creating a new custom simulator, the simulation engine includes many convenient features for a parallel simulator such as a performance monitoring tool, a GUI-based simulation initialization, etc.

The PROTEUS engine, when combined with user application source code and optional custom hardware description modules, can be compiled to an uni-workstation executable that simulates the execution of the given application on the target parallel architecture. The program execution is simulated at the granularity of individual hardware events such as machine instructions, network accesses, etc. The fidelity of a customized simulator can be arbitrarily improved – at the expense of simulation speed – for any target system. User applications are coded in a superset of the C programming language that has been extended with simulation related function calls and primitives. The use of C greatly extends the usefulness of PROTEUS by easing the development of applications.

The FUNet simulator was created by incorporating a custom simulation of FUNi and FUNet into the PROTEUS simulation engine. Steps were taken to ensure the fidelity of the simulator when possible. This section visits the different facets of the simulated FUNet cluster. For each area, we describe the parameters and assumptions that are relevant to the accuracy of the simulation and also those parameters that were overlooked in the simulations.

6.1.1 Processing Nodes: 40 MHz SPARCstation2

During compilation, the user application is augmented with additional cycle-counting code. The cycle-counting data in our version of the PROTEUS engine is derived from the SPARCstation2 with the SPARC instruction set. During a simulation, the PROTEUS engine keeps an accurate account of the application's execution cycle down to the instruction level. (Since the PROTEUS simulator keeps track of time only in terms of clock cycles, we arbitrarily assigned 40 MHz as the clock rate for our simulated processing nodes.)

The augmentation process in PROTEUS uses an optimistic model for the instruction execution on a SPARC microprocessor that is fully pipelined with register bypass. Instruction fetches are assumed to always hit in the instruction cache, and interlocks due to data dependency are ignored. Thus, all arithmetic and logical instructions, both scalar and floating-point, contribute only one cycle to the total cycle count. Flow control instructions take two cycles, but the second cycle is a delay slot that another instruction can occupy. The augmentation process also does not account for the effect of data caching. Cache hits are assumed for all normal memory accesses. Thus, all load and store instructions are considered single cycle instructions, with the exception of load-double-word which takes two cycles.

To accurately emulate the interaction between the CPU and FUNi, added details about the memory and processor I/O are incorporated into our simulation. The concept of a memory bus is included to bring forth the effect of bus contention due to the demands of FUNi's programming interface. To perform bus transactions, devices, such as the CPU and FUNi, must wait until the bus is freed. (The CPU is given priority in a bus arbitration.) Once control of the bus is granted, the device occupies the bus for the entire duration of the transaction, thus sequentializing the different transactions from FUNi and the CPU.

A memory-mapped read latency is approximately 28 CPU cycles (based on experimental results), plus any additional cycles for acquiring the bus. The simulator assumes the CPU will buffer the memory-mapped writes, and thus, a memory-mapped write contributes only two cycles to the program execution. For each memory-mapped access, a bus acquisition and a transaction are simulated. The appropriate latency is introduced between the time when the memory-mapped access is issued and the time when the access arrives at the memory-mapped device. Loads and stores to the user's send and receive queues never hit in the cache. Their latency and effect on bus contention must also be accounted for in the simulation. Load and store misses use the same latency model as memory-mapped accesses.

6.1.2 User Programming Environment

User applications are coded in a superset of the C programming language. The FUNet cluster maintains the MIMD message-passing programming model stated in Section 1.2.1. When the simulator starts at time zero, a process is created for each simulated node, and all started processes begin execution at the `main()` procedure of the user application. During the parallel execution of the application, peer processes can communicate explicitly with each other through FUNi.

There is one inconvenience in PROTEUS's C programming environment which involves the use of global variables in the user programs. In an actual FUNet cluster, the process on each node executes a private copy of the executable in its own virtual address space. Thus, a C global variable declared in a program is replicated at the same virtual address on each workstation. However, the PROTEUS simulator compiles user C code into a single executable used by all simulated nodes. In the PROTEUS executable, only a single storage is created for each global variable declared in the user program. Thus , the execution on all simulated nodes references the same storage when accessing the global variable.

To emulate the correct behavior of an FUNet cluster on the simulator, all global variables must be extended into an array to give each simulated node a private copy of the global variable. When referencing a global variable on the local node, the user program needs to index into the global variable array with the node index. Because every process now uses a different memory location for the same global variable, when referring to a global variable on a remote node through a pointer, the user program must add an additional offset to the local virtual address of the global variable. The additional level of dereferencing adds a small penalty to the simulated performance.

6.1.3 Operating System

In a PROTEUS system, the application does not execute under a true operating system. Instead, the PROTEUS simulation engine provides the basic operating system functionalities such as memory management, interrupt handling, etc. PROTEUS presents a single-machine view of execution. A single application starts on all simulated nodes when the simulation begins. The simulator terminates when all the processes on all the simulated nodes have terminated. The basic PROTEUS system does not support the distributed multi-tasking view taken by the FUNet cluster. However, the PROTEUS simulation engine does contain a multi-threading package which future studies can use to implement a mock multi-tasking environment.

In our effort to assess the effectiveness of the FUNi design, we ignored the effect of time sharing. The benchmark programs were executed alone without interference from other applications.

6.1.4 Physical Network: Hypercubic Arctic Hub

The FUNet cluster simulator incorporates a custom network simulation for a hypercubic direct-routing network based on Arctic. The operation of the Arctic network is accurately depicted in all relevant details in the simuation. The network is simulated at the estimated network clock rate of 25 MHz. The 4-by-4 Arctic router is simulated with three buffers at each input section, with one reserved for high priority packets. An output section stops the flow of packet traffic when its corresponding input section runs out of packet buffers. The flow-through latency of the simulated Arctic is five network cycles. The transfer bandwidth through an established path is two 16-bit halfwords per network cycle. The wire delay between the routers is one network cycle.

6.1.5 FUNi

FUNi hardware events are accurately accounted for in terms of latency and resource utilization. The simulator supports the full programming interface defined in Chapter 3. User processes access FUNi's internal control registers through simulated memory-mapped reads and writes, as described in Section 6.1.1. The simulated FUNi uses DVMA accesses in bursts of 1, 4 or 8 words to access the user memory. The DVMA bus transactions are sequentialized with bus transactions from the CPU. Fifteen bus cycles are allotted for the bus transaction overhead (not including the cycle waiting to acquire the bus), and a transfer bandwidth of one 32-bit word per two cycles is used in the simulation.

6.2 Benchmark and Analysis

Two benchmark programs based on University of California at Berkeley's version of the Connection Machine Active Message (CMAM) communication library [21, 22] were executed on the FUNet simulator to evaluate FUNet and FUNi. The CMAM library was ported to the FUNet cluster by rewriting the low-level primitives that dealt with the network interface. A few extensions were made to the original CMAM library to take advantage of the features of FUNet and FUNi. A new set of primitives which efficiently supports single-packet active messages with up to twenty arguments was added. A new set of data transfer primitives was also added to take advantage of FUNi's low-overhead remote DMA data transfer. The description of the benchmarks is presented below, followed by the results of the experiments.

6.2.1 CMAM Primitives Benchmark

The first benchmark is used to quantify the performance of FUNi. Instead of measuring idealistic raw throughput by sending and receiving meaningless messages, we

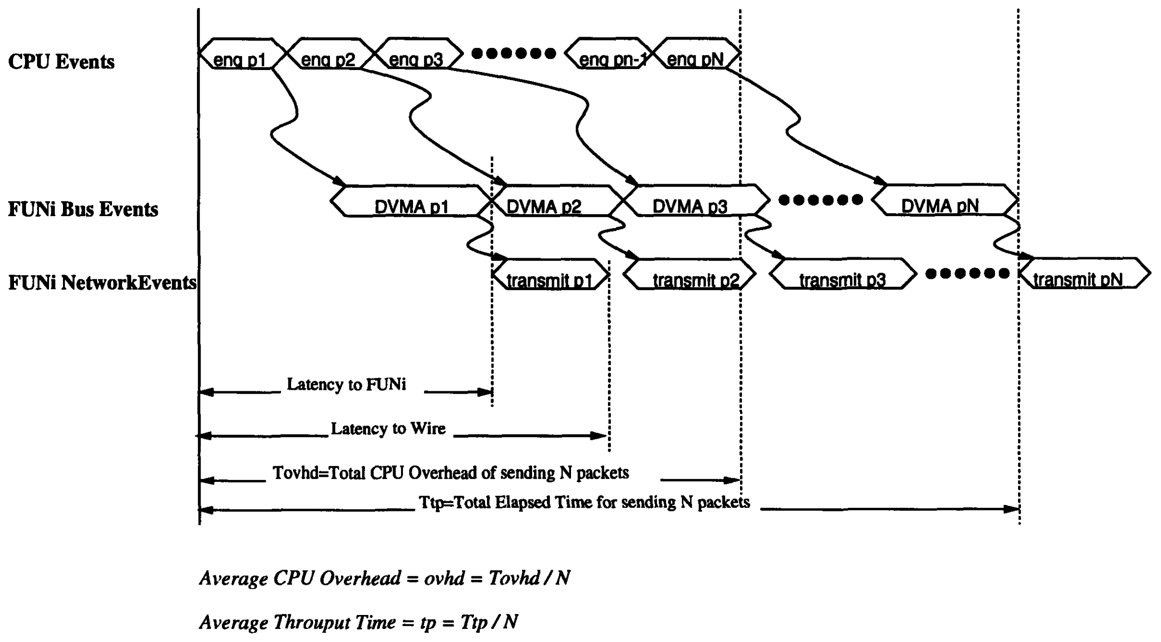

Figure 6.1: Average Throughput Time and CPU Overhead Determination

measure the performance of FUNi when coupled with the CMAM library. The CMAM communication layer provides a simple and non-obtrusive veneer for the underlying FUNi hardware. However, the communication primitives contain sufficient functionalities for it to be useful in real applications. Measuring the performance of the CMAM primitives over the FUNi hardware yields a better representation of FUNi's performance in real programs.

The benchmark suite that was included in the CMAM library distribution has been adapted for FUNet's ported version of the CMAM library. The original suite was modified and augmented with additional items to give a better assessment of FUNi's performance. For some of the primitives, many items were measured twice under two different catagories: once under the heading of throughput time (*tp*) and once under the heading of CPU overhead (*ovhd*). The overhead measures how fast the user program can carry out a certain primitive on the CPU. The throughput time measures how fast FUNi can actually complete the transactions requested by the primitive in question. The two numbers differs significantly in most cases.

User programs interact with FUNi indirectly through the memory system for sending and receiving messages. The scheme effectively decouples the actual speed of FUNi from what is visible to the user program. For example, a user sends a message by enqueuing a packet into the send queue in the user's memory. Once the packet is enqueued, the network access is considered complete from the program's point of view. FUNi alone, without incurring any more CPU overhead, completes the remainder of the sending process. Therefore, the absolute overhead visible to the user program is simply the CPU cycles spent to enqueue the packet into the send queue. The rate at which FUNi actually satisfies the send request on the network only appears to user programs as latency and bandwidth. Thus in the case of sending a packet, the overhead would correspond to how long it takes for a user process to enqueue a packet into the send queue, whereas the throughput time would indicate how long it takes for FUNi to transmit an enqueued packet. Please see Figure 6.1 for an example of determining the average overhead and throughput time for sending a message.

The adapted CMAM primitive benchmark suite is executed on a 32-node FUNet cluster simulation. For reference, a similar suite is also executed on 32 nodes of CM-5 with 32 MHz SPARC 601 processors. A subset of the results from the benchmark is tabulated in Table 6.1. The table is divided into four sections: active message primitives, block data transfer primitives, shared memory library calls, and global synchronization barrier. In each section, the headings beginning with the lower case *cmam* denote the results from the FUNet simulation; headings beginning with upper-case *CMAM* denote results belonging to CM-5.

Active Message Primitives

The first section of the table presents the results of the active message primitives in the CMAM library. The primitives presented in this section are FUNet's `cmam_4()` and `cmam_reply_4()` which both send active messages of the corresponding priority with four arguments. `CMAM_4()` and `CMAM_reply_4()`, the CM-5 equivalents, are also

Active Message Passing Primitives										
	send tp		send ovhd		recv tp		recv ovhd		round-trip	
	usec	cyc	usec	cyc	usec	cyc	usec	cyc	usec	cyc
cmam_4	5.5	219.4	2.8	113.2	8.7	348.2	5.5	221.3	29.0	1160.8
cmam_reply_4	5.4	215.8	2.5	98.2	8.4	336.1	5.0	201.3		
cmam_n=1	5.5	218.0	3.4	134.2	9.4	377.8	6.3	251.3	28.7	1147.9
cmam_n=20	16.1	645.2	6.1	244.2	24.2	969.7	14.7	587.3		
cmam_reply_n=1	5.4	215.0	3.0	119.2	9.1	363.8	5.8	231.3		
cmam_reply_n=20	16.2	646.8	5.7	229.2	23.9	955.6	14.2	567.3		
CMAM_4	1.5	50.7			1.6	52.2			12.5	413.9
CMAM_reply_4	1.3	42.8			1.6	52.2				

Block Data Transfer Primitives								
	send tp		send ovhd		recv tp		recv ovhd	
	MB/s	cyc	MB/s	cyc	MB/s	cyc	MB/s	cyc
cmam_xfer_4	2.9	218.5	5.1	126.6	1.8	365.0	2.7	235.6
cmam_reply_xfer_4	3.0	215.8	5.7	111.7	1.9	345.2	3.0	215.6
cmam_mfer_n	10.1	63.6	25.7	24.9	7.5	85.5	14.4	44.3
cmam_reply_mfer_n	9.9	64.4	28.4	22.5	7.0	90.9	15.9	40.3
CMAM_xfer_4	7.2	73.2			8.5	62.0		
CMAM_reply_xfer_4	9.8	54.1			10.0	52.6		

Shared Memory Library Calls								
	read_i		read_d		write_i		write_d	
	usec	cyc	usec	cyc	usec	cyc	usec	cyc
cmam_*	30.7	1227.2	31.0	1238.2	30.2	1209.6	30.5	1219.8
CMAM_*	13.7	450.8	14.5	480.3	12.8	421.5	13.2	434.7
	i ovhd		i lat		16 ovhd		16 lat	
	usec	cyc	usec	cyc	usec	cyc	usec	cyc
cmam_get_*	13.8	553.7	30.1	1205.0	20.0	797.8	36.6	1457.6
cmam_put_*	3.4	134.5	30.9	1236.0				
CMAM_get_*	4.3	141.3	13.4	443.0	11.1	364.9	20.5	678.9
CMAM_put_*	2.0	65.7	16.2	533.8				

Global Synchronization Barrier		
	barrier	
	usec	cyc
cmam_barrier	95.6	3825.2

Table 6.1: Performance Comparison between CMAM and FUNet Active Message Library Primitives

shown as a comparison. FUNet's extensions to active message primitives, `cmam_n()` and `cmam_reply_n()`, are also shown in this section. `cmam_()` and `cmam_reply_n()` were measured for the case of a single argument and the case of the maximum of twenty arguments.

For each primitive, five parameters are measured. The first two columns are under the categories of *send tp* and *send ovhd.* The send throughput time measures the total time required for a node to send to other nodes. The send overhead measures the execution time of the primitive on the CPU. Because of the nature of the CM-5 network interface, there is no difference between the network throughput time and the CPU overhead; only one number is shown for each CM-5 primitive. The *send tp* column shows that `cmam_4()` requires approximately four times as many cycles as `CMAM_4()` to complete a transmission. However, in the next column, we see that `cmam_4()` actually requires only approximately twice as many cycles as `CMAM_4()` in terms of CPU overhead. (The extra cycles in the *send tp* column are attributed to the limiting bandwidth of DVMA transfers on the SBus.) Also, FUNet's active message library is written in C and compiled by GCC without optimization, whereas the low-level primitives in the CMAM library were custom crafted in assembly code. Thus, hand coding the performance-critical primitives for FUNi can further reduce the difference in the overhead. The next two columns are under the categories of *recv tp* and *recv ovhd.* The receive throughput time shows the total time required for a node to receive an inbound active message from the network. The receive overhead measures the time spent by the CPU for receiving an already arrived and queued message. `cmam_4()` performs considerably worse than `CMAM_4()` on the CM-5. Over seven times as many cycles are needed to completely receive an inbound packet through FUNi, and at least four times as many cycles are required on the CPU. The large receiving overhead is inherent to the FUNi's interface design. Since inbound packets are written to the receive queues by FUNi through DVMA, the processor will miss in the cache when accessing the contents of the packets. The cache miss latency not only increases the receive overhead, but the resulting bus transactions for loading the cache also decrease the bus bandwidth available to FUNi for DVMA. Thus, both the throughput time and overhead are affected.

The final column of this section shows the round-trip time of a request-priority active message and a returning reply-priority message. Each round-trip involves two sends and two receives. By taking half of the difference between the round-trip time and twice the sum of the *send ovhd* and *recv ovhd*, we are able to arrive at a figure that represents the transfer latency inherent to the network interface hardware. In the case of FUNet, we can see that FUNi introduces a total of nearly 250 processor cycles of latency on the sending and receiving node pairs. By comparison, the network transit latency is insignificant.

Block Data Transfer Primitives

The next section of the table presents the results from the data transfer primitives. The primitives tested are `cmam_xfer_4()` and `cmam_reply_xfer4()` which both transfer data in blocks of four words. `CMAM_xfer_4()` and `CMAM_reply_xfer_4()` are the equivalent primitives tested on CM-5. `cmam_mfer_n()` and `cmam_reply_mfer_n()` are the FUNet extensions which use DVMA and perform transfers in packets of up to 16 words. For these primitives, the send and receive throughput time and overhead, in similar context as in the last section, are measured.

In the send throughput time column, we can see that both `cmam_xfer_4()` and `cmam_reply_xfer_4()` can only achieve one third of the transfer bandwidth of their CM-5 equivalents because of the lower bandwidth of the FUNi interfacing scheme. However, in the same column we can also observe that by taking advantage of the DVMA feature and the larger packet size of FUNi, `cmam_mfer_n()` and `cmam_reply_mfer_n()` can achieve a transfer bandwidth comparable to `CMAM_xfer()` and `CMAM_reply_xfer()` despite of the lower bandwidth of FUNi. The send overhead column shows the rate at which the CPU can enqueue data transfer requests to the FUNi interface queues. In this case we can observe the decoupling effect of FUNi's large interface queues. A FUNet processing node can enqueue requests for data transfer using `cmam_xfer_4()` and `cmam_reply_xfer_4()` at up to 5 MByte/sec, even though the actual data transfer occurs much slower.

The next two columns present the throughput time and the overhead of the data transfer on the receiving node. A similar phenomenon as on the sending node can be observed on the receiving node. The throughput time of the data transfer by `cmam_xfer_4()` and `cmam_reply_xfer_4()` is much lower than the CMAM equivalent primitives. However, `cmam_mfer_n()` and `cmam_reply_mfer_n()` are able to make up for FUNi's drawback in bandwidth and perform respectively. The decoupling effect of FUNi can also be observed on the receiving node.

Shared Memory Library Calls and Global Synchronization Barrier

The second to the last section measures the performance of calls to a shared-memory library which implements a shared-memory coherence protocol in software. These high level communication interfaces are constructed from the CMAM primitives. In each case, the calls on the FUNet system can achieve about half of the performance of their counterparts on the CM-5. Finally the last section presents the time required for a global synchronization barrier. The FUNet system does not have a special control network for global synchronization. Barriers must be emulated in software with message passing. The software-emulated barriers are quite expensive. At nearly 100 microseconds per barrier on a thirty-two node system, synchronization barriers must be used sparingly in order to maintain good processor utilization.

6.2.2 Matrix Multiply

This particular version of matrix-multiply is taken from von Eiken et al. in their paper describing active messages and Split-C [22]. The C code for the matrix-multiply loop is included in Appendix C. In this algorithm, both the multiplicand matrices A_{N*R} and B_{R*M} and the product matrix C_{N*M} are partitioned into blocks of columns across the participating nodes. The algorithm begins by having each node update its own columns of C based on its own columns of A and B. Next, each node fetches each successive remote column of A and updates its own columns of C accordingly. A total of $2NMR$ floating point operations is performed for every matrix multiply.

The example is well-suited for a FUNet cluster because the algorithm pipelines each remote fetch of the columns of A with the computation based on the last fetched column. The overlapping of communication delay with useful computation hides the effect of FUNi's relatively high communication latency. With the latency hidden, FUNi's lowered overhead for communication allows high utilization of CPU cycles and achieves good scalability.

Two experiments were performed with the matrix multiply program. The first experiment is designed to demonstrate that a FUNet cluster can achieve good CPU utilization despite the relatively low bandwidth and high latency in interprocessor communication. The second experiment demonstrates the scalability of the FUNet system. For each experiment, three runs were made. One run is made on a CM-5 using University of California at Berkeley's version of CMAM library. Next, another run is made on the FUNet cluster using an identical version of matrix multiply as the one used for CM-5. Finally, the other run is made on the FUNet cluster, this time allowing the use of the FUNet extensions to the CMAM library for improved performance.

In both experiments, performance is measured in million floating point operations per second (MFLOPS). MFLOPS in this case is the algorithmic MFLOPS calculated by dividing $2NMR$, the number of floating point operations in the matrix multiply, by the execution time. Peak single processor performance is the algorithmic MFLOPS for running the algorithm on a single processor, thus eliminating all communication costs.

Latency Hiding and Overhead Amortization

As explained previously, the algorithm used in this experiment is able to hide the effect of communication latency of fetching remote columns of A by pipelining the fetches with computations. By increasing the amount of computations resulting from each data fetch, the communication overhead is amortized to achieve very good CPU utilization. This experiment was performed by von Eiken, et al. for CMAM on a 128-node CM-5 [22]. The experiment is scaled down for execution on a 32-node FUNet simulator. In the different trials, the dimensions of the matrices were varied to control the ratio of computation versus communication while maintaining the total number of floating point operations. To fetch all remote columns of A, each node must transfer $(R - \frac{R}{p})N$ doublewords, where p is the number of processing nodes. In this experiment, N, the number of rows in A, is kept constant at 128 while R, the number of columns of A, is varied from 64 to 2048 to control the amount of communication in each trial. M is adjusted accordingly from 1024 to 32 to keep the total number of computation, $2NMR$, constant at 16 million floating-point operations.

Figure 6.2 plots the results of the experiment. The Y-axis represents the percentage of CPU utilization in each run, and the X-axis shows M/p, the number of columns of C held on each processor. Three curves are plotted. A curve is plotted for matrix multiply on FUNet using the original CMAM library only, and another curve is plotted for matrix multiply on the FUNet that utilized the FUNet extensions to CMAM. As a reference, another curve is plotted for the result from executing matrix multiply on CM-5. Comparing the diamond and triangle marked curves, we see that FUNet exhibits normalized behavior similar to CM-5 when the original CMAM primitives are used in both cases. In each case, as M increases (or R decreases), processor utilization quickly approaches optimal. Over eighty percent of peak performance is achieved for $M \geq 16p$. When the FUNet extensions are allowed, better than eighty percent of peak performance is achieved for M, as little as $4p$, and higher processor utilization is achieved overall.

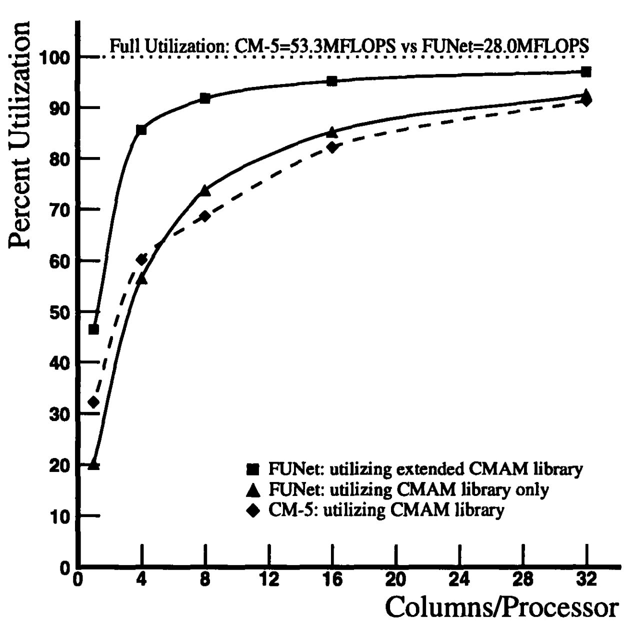

Figure 6.2: Utilization vs. Columns per Processor for Matrix-Multiply

Scalability

In this next experiment, square matrix multiplies of increasing dimensions are carried out on systems of varying size to determine the scalability of the FUNet cluster. Again, three runs of this experiment are carried out. Two experiments are carried out on the FUNet cluster, once with the FUNet extensions to CMAM and once without. One more run is carried out on CM-5 to serve as reference. Figures 6.3, 6.4, and 6.5 plot the result from multiplying two square-matrices of 64-by-64, 128-by-128 and 256-by-256, respectively.

In each plot, the X-axis indicates the number of processing nodes used in each run and the Y-axis represents the performance achieved in each run normalized to peak single processor performance. In each curve, a dotted line with slope 1 is drawn to represent the case of perfect scalability. In Figure 6.3 for square matrix multiply of dimension 64-by-64, we see that in all three runs, the curves break from linear speedup for a system size greater than 16. This is because the problem size is simply

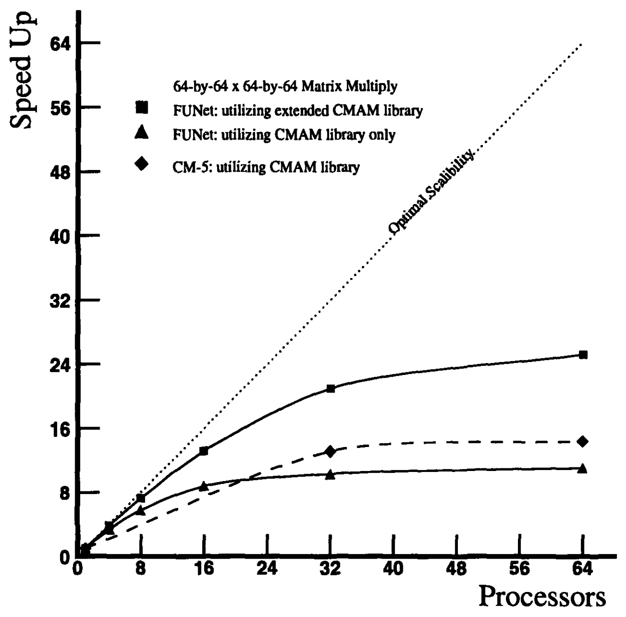

Figure 6.3: Processor Scalability: Multiplying 64-by-64 Square Matrices

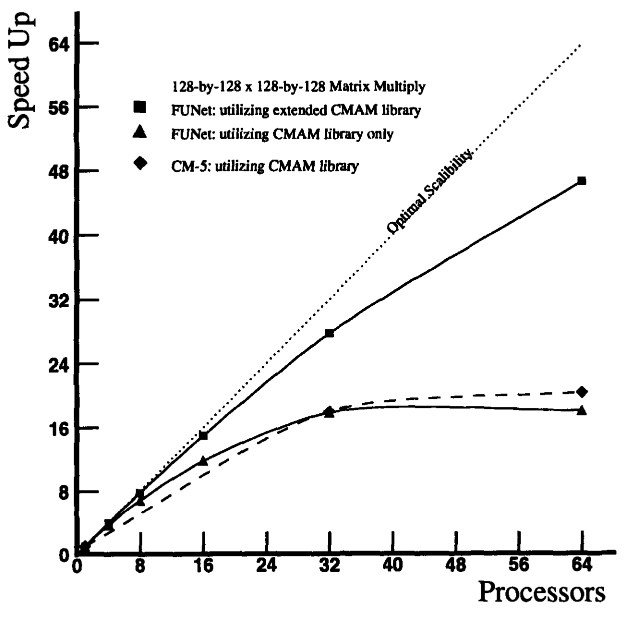

Figure 6.4: Processor Scalability: Multiplying 128-by-128 Square Matrices

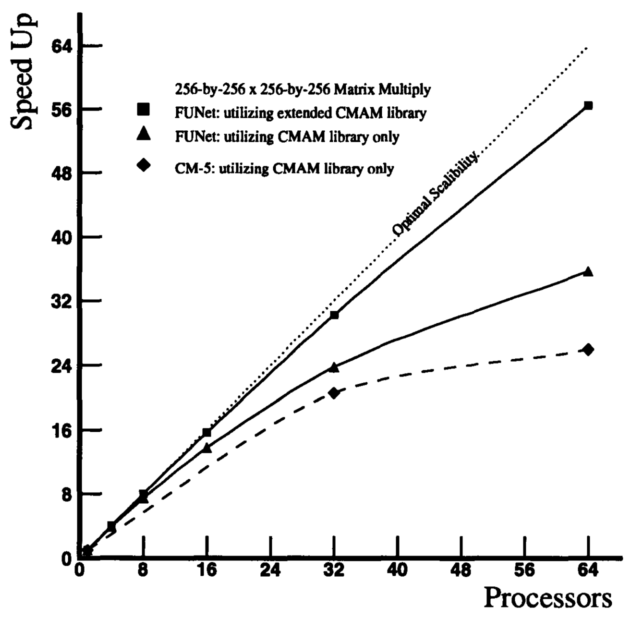

Figure 6.5: Processor Scalability: Multiplying 256-by-256 Square Matrices

too small for the computation to amortize the communication overhead on larger systems. Also from this plot (and also plots in Figures 6.4 and 6.5), we can see similar normalized behavior between FUNet and CM-5 when FUNet is restricted to the original CMAM library. In all three cases, significantly better normalized behavior is observed on the FUNet cluster when FUNet-specific extensions to CMAM are included to lower the communication overhead. In Figure 6.4 for square matrix multiply of dimension 128-by-128, we begin to see improvement in the linearity of speedup from the increased problem size. In Figure 6.5 for square matrix multiply of dimension 256-by-256, matrix multiply is able to scale up to 64-node systems in all three cases. In particular, the diamond curve, representing the case of the FUNet cluster with an extended CMAM library, closely follows the ideal scalability curve. Thus, by overlapping communication latency with useful computation, the lowered overhead of communication enables the FUNet cluster to achieve comparable processor utilization and scalability as a contemporary MPP system.

Chapter 7

Conclusion

In this thesis, we set out to design a user-level network that enables a cluster of workstations to carry out finer-grained parallel processing. In doing so, we had hoped to produce a cost-efficient alternative to existing MPP supercomputing systems to encourage the popularization of parallel computing. To this end, we have produced the design of FUNet and FUNi, a user-level network and its network interface, that allow low overhead interprocessor communication on a cluster of SBus-equipped workstations.

In an attempt to design a network interface that would retrofit the design of existing microprocessors and workstations, the design freedom available is limited. We recognized, early on, that the latency and bandwidth of communication will suffer from a network interface that is far removed from the microprocessor. Unable to resolve this problem in hardware without an extensive redesign of microprocessors and workstations, we compromised for a software solution. By overlapping communication delays with useful computations, the effect of communication latency can be masked, thus leaving the communication overhead as the only true penalty for interprocessor communication. Thus, we concentrated on minimizing the overhead of communication through FUNi. As shown in Section 6.2.2, by keeping the overhead low, the simulated FUNet cluster is able to successfully execute a relatively fine-grained parallel program with good performance and scalability, despite the moderately long communication latency experienced in our system.

Judging from the simulation, we believe we have produced a satisfactory design that can, at least temporarily, enable efficient and scalable fine-grained parallel processing on a cluster of workstations. However, observing the present rate of improvement in microprocessor performance, in the long run, no network interface design, if constrained by the bus bottleneck, will be able to keep up with future microprocessors' communication demands. Eventually, scalar microprocessor performance will plateau because of the physical limitations of the materials from which microprocessors are built. At that point, computer architects must turn to parallel processing as the remaining avenue for continuing performance gain. Therefore, future generations of microprocessors need to start considering the incorporation of a tightly coupled network interface as an integral part of the processor design to minimize both the latency and overhead of interprocessor communication.

The following section briefly describes a few of related projects in the field of workstation-based massively parallel processing. Finally, the last section of this chapter discusses the future of the FUNet cluster project.

7.1 Related Work

The field of workstation-based parallel processing appears to be highly active. Two commercial ventures, IBM SP1 and Dolphin Technology SCI, are especially noteworthy because of their close resemblance with this thesis.

7.1.1 IBM 9076 SP1 Supercomputer

IBM, which in the past had offered parallel systems based on LAN cluster of RS/6000 workstations, has introduced their next generation of workstation-based parallel system, the 9076 SP1 [10]. To minimize the design effort, the SP1 parallel system is based on existing 62.5 MHz RS/6000 processor boards for the RS/6000 family of workstations. To further leverage the engineering in the workstation sector, the SP1 system is designed to be easily upgradable as the processor and the processor board improve. A dedicated high performance network switch is used to support internode communication in parallel applications. The SP1 is available in configurations between 8 to 64 processing nodes; a 128-node experimental system is under development. An 8-node SP1 is listed for $312,000; a 64-node costs $2.75 million. (At an average of $40,000 per node, the cost per node of SP1 remains high when compared to $25,225 for a top of the line RS/6000 Powerstation375.)

This system closely resembles the spirit of this thesis. The SP1 system attempts to simplify the design and reduce the cost of parallel systems by incorporating fully engineered workstation hardware as processing nodes. Interprocessor communication for parallel processing is supported by a dedicated, high-performance network switch. The difference between SP1 and FUNet is in the execution of the concept. First of all, the design and implementation of the dedicated network and network interface differ significantly. Secondly, SP1 strongly retains the single-machine view of execution whereas FUNet follows a more distributed, loosely-coupled philosophy.

7.1.2 Dolphin SCI Technology

The Scalable Coherent Interface (SCI) standard, IEEE-1596, defines a uni-directional point-to-point interconnect technology with a ring topology and an incorporated directory-based cache coherence scheme. This interconnect standard was originally defined for the purpose of interconnecting processing nodes in a scalable shared-memory multiprocessor system. SCI allows interconnection for up to 64,000 nodes.

Dolphin SCI Technology developed a 500 MHz GaAs implementation of the SCI controller chip [23]. The SCI chip can be connected directly to 10-meter cables with 2-byte wide ECL differential interconnects. A minimum data rate of 128 MByte/sec can be expected. The controller is available as a core design that can be matched with various bus and processor glue logic to work on many different platforms, including workstations.

Thus, instead of FUNi, one could implement a network interface card based on Dolphin's SCI chip to construct a parallel-processing communication network for a shared-memory parallel cluster of workstations. In comparison with FUNi, the SCI interface, closely resembling a local area network, does not easily support user-level accesses. The ring topology of SCI also introduces concerns in scalability of SCI-based parallel systems. The current industry outlook expects SCI to become a high-performance backplane bus or LAN replacement, instead of its original purpose in parallel processing.

7.2 Future Work

Up to this point, the study of FUNi and FUNet remains a paper study based on simulations. Much work, in both hardware and software, needs to be completed for the realization of a FUNet cluster.

7.2.1 Hardware Implementation

Hopefully, the study conducted in this thesis has been convincing enough to justify further efforts in the construction the FUNet hardware for future studies. The hardware construction includes the implementation of FUNi and the Arctic Network Hub. The current plan is to implement FUNi as SBus cards to be used with a cluster of SUN SPARCstations. The FUNi SBus cards will have a bi-directional channel to the Arctic Network Hub that will provide the interconnections for the workstations. The logic design will abandon the traditional schematic capturing process. Instead, designs will be entered in Verilog Hardware Description Language to be compiled into the appropriate netlists by Synopsys HDL Compiler. The current plan calls for the implementation of FUNi in Xilinx 4000 family of Field Programmable Gate Arrays (FPGA) to facilitate future revisioning of the network interface after the current design has been realized with hardware. The following section describes the two hardware projects.

FUNi SBus Card

FUNi will reside on a printed-circuit board (PCB) SBus card. SBus, the I/O bus used in Sun SPARCstations, is defined with three major goals: ease of implementation, low power consumption, and small form factor [16]. While we benefit from the ease of implementation due to the simple synchronous bus protocol and the CMOS compatible electrical interface, the limit on power consumption and form factor poses some difficulties. A standard single-width card for SBus is specified to be only slightly larger than a 3-by-5 index card and is limited to 10 watts of average power consumption [9]. Using primarily CMOS components will help us remain under the power constraints, and the surface mount technology on multiple-layered PCB design can relieve the problem of over-crowding. However, most likely, we will compromise for the less elegant double-width SBus card to allow for more implementation versatility.

An important design goal of our SBus card is revisibility. FPGA's are used in the implementation not only to achieve high integration of logic but also to allow the possibility of reusing the hardware in future revisions of FUNi. For this idea to succeed, the SBus card must be designed with sufficient expandability and flexibility in its datapath since everything outside of the FPGA's cannot be changed once the PCB is fabricated. Thus the various design choices for the SBus card need to be biased toward flexibility rather than elegance. Therefore, even though it is possible to produce a completely functional network interface on a single-width SBus card, it is more profitable in the long run to fabricate a double-width card with a more flexible and complete datapath design.

Arctic Network Hub

Currently a centralized direct routing hypercubic network hub comprised of Arctic routers is being considered. The Arctic router, developed for the *T project, is a four-by-four high-performance packet-switched router. Implemented in a 50 MHz gate array, Arctic is capable of 1600 Mbit/sec per channel. Because of the speed limitation of interfacing logic on FUNi, we only expect to clock the network at 25 MHz, reducing the bandwidth down to 800 Mbit/sec per channel.

The Arctic project, although near completion, is still actively modified to ensure successful fabrication. The instability in the Arctic design introduced uncertainty in the actual network topology of the routing hub. To combat the uncertainty, the connection topology of the dedicated network is abstracted from the design of FUNi. Each SBus interface card has one bi-directional connection to the network. The actual structure of the network is unknown to FUNi. FUNi will be outfitted with a RAM-based route table to translate node addresses into the appropriate network route information.

The RAM-based route table provides additional benefits. The content of the route table is loaded by the operating system. By modifying the content of the route table, the operating system can exactly control the mapping of the abstract node ID to the physical workstations. This gives the operating system the ability to, for each workstation, individually determine who and where its peer workstations are. Thus, it is possible to partition and cluster into non-interfering sub-clusters for more flexible usage. Individual nodes can also be remapped or excluded for fault tolerance or load balancing.

7.2.2 Software Development

This thesis has dealt mainly with the hardware design of the proposed workstation-based parallel system. However, several software issues also need to be dealt with as part of the FUNet project. The first task involves coding the device driver for the FUNi SBus card. Because of the popularity of SBus devices, coding a SBus device driver is a relatively simple task which we will not discuss in this section. In the rest of this section, we will first discuss the availability of user applications for the FUNet cluster and then discuss operating system level work.

Active Message Communication Layer and Software Availability

To simplify the network and network interface, FUNet will only support simple packet types. This network environment is well suited for active messages. In active message passing, each packet carries the pointer to its packet handler, and the contents of the packet are used as arguments to the packet handler at the destination processor. An active message incurs a relatively small communication overhead and works well with small-sized packets. More elaborate communication abstractions can be constructed from basic active message primitives.

University of California at Berkeley has developed an active message communication library for the Thinking Machines Corporation's CM-5 [22, 21]. The library can be ported for our FUNet system. This immediately opens up a large source of applications. Most software that is based on the Connection Machine Active Message (CMAM) library for the CM-5 can be ported with minimal modifications. Furthermore, University of California at Berkeley has also developed the Split-C compiler [8] that will compile C programs with extended parallel primitives to run on the CM-5 using CMAM primitives. We will also be able to take advantage of the Split-C compiler to develop software for our FUNet parallel system.

Context Switching and Protection

To make more efficient use of computing resources in a FUNet cluster, we need to allow multiple applications to time-share the processing nodes and the network. To maintain data security between the different applications on the same network, FUNi makes use of Group Identifier (GID) tags to maintain data security of packets sent on FUNet. However, during a parallel context switch, FUNi requires assistance from the operating system to manage the GID assignment and to swap FUNi hardware states.

Time-sharing multiple applications is recognized as an important aspect of our parallel system, and the interface hardware is designed with the necessary security mechanisms to support time-sharing. However, we will not be able to demonstrate the full time-sharing capability of FUNi without making detailed modifications to the operating system kernel. Therefore, initially, we only plan to support our proposed parallel system to run in a single parallel application mode under the existing UNIX operating system. The time-sharing issues can still be studied by using a parallel version of light-weight multiprocessing in which multiple parallel applications time-share the network and other resources but appear to the operating system as a single application.

7.2.3 Other Miscellaneous Ideas

Numerous other important issues remain open for further study pending on the completion of the FUNet project. This section describes the most interesting ones that may be investigated as work continues after the completion of the FUNi and FUNet projects.

Gang Scheduling

With the possibility of time-sharing parallel applications, gang scheduling becomes an important issue in the operating system design. For an application to progress, each sending and receiving pair of processes must be the executing processes of their respective nodes at the same time. Otherwise, the packets sent are rejected and have to be repeatedly retried by the sending interface. This is counter-productive since the retries increase network traffic but do not accomplish any useful work until the receiving process is swapped in. Since each workstation in the proposed system is under independent control of its own operating system, global coordination is not trivial. Furthermore, interactions with sequential applications must be considered. Intelligent scheduling is necessary for an efficient execution on a loosely coupled distributed parallel system like the proposed FUNet system.

Interrupt-Driven Message Handling

In the current design, user programs need to check for the presence of the network packet by polling the network interface. This adds unnecessary overhead to the cost of communication. An interrupt-driven system would eliminate the communication overhead spent on testing for inbound packets. We are unable to implement an interrupt-driven system because the existing operating systems for the SPARCstations do not support user-level interrupts. An interrupt-driven system based on a system-level interrupt for user packets would incur an even larger overhead than a simple polling system. Thus, in the current design, only system-level network packets cause interrupts. Before an efficient interrupt-driven system can be implemented, the operating system and the microprocessor hardware need to support proper handling of user-level interrupts.

Heterogeneous Parallel Systems

To allow ultimate flexibility in the configuration of parallel systems, one would like to be able to construct parallel systems out of a heterogeneous group of workstations, possibly each with a different operating system and of different performance. This poses a greater challenge in software than in hardware. As long as a fixed network interface protocol is agreed upon, interconnecting heterogeneous workstations simply requires separate implementations of the network interface designed to fit each system.

However, the possibility of differing performance across the system introduces the task of load balancing to the already perplexing problem of gang scheduling.

Bibliography

[1] A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. A user's guide to PVM parallel virtual machine. Technical report, Oak Ridge National Laboratory, July 1991.

[2] A. Boughton, G. Papadopoulos, B. Greiner, S. Asari, S. Chamberlin, J. Costanza, R. Davis, T. Durgavich, D. Faust, E. Heit, T. Klemas, J. Kwon, E. Ogston, G. Rao, and R. Tiberio. *Arctic User's Manual.* CSG, LCS, 1993.

[3] E. A. Brewer. Aspects of a parallel-architecture simulator. Technical report, MIT Lab. for Comp. Sci., January 1992.

[4] E. A. Brewer and C. N. Dellarocas. *Proteus: User Documentation.* MIT Lab. for Comp. Sci., 0.5 edition, 1992.

[5] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical report, MIT Lab. for Comp. Sci., September 1991.

[6] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of tcp processing overhead. *IEEE Communication Magazine*, June 1989.

[7] D. E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture.* Prentice Hall, 1988.

[8] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. *Introduction to Split-C.* University of California at Berkeley, 1.5 edition, 1993.

[9] E. H. Frank and J. D. Lyle. *SBus Specification B.0.* Sun Microsystems, Inc, 1990.

[10] A. Gillen. AWS shoots for the stars. *MIDRANGE Systems*, February 1993.

[11] J. L. Hennessy and N. P. Jouppi. Computer technology and architecture: An evolving interaction. *Computer*, September 1991.

[12] The HP cluster computing program. pamphlet, 1992.

[13] C. F. Joerg and D. S. Henry. A tightly-coupled processor-network interface. Technical report, MIT Lab. for Comp. Sci., March 1992.

[14] G. Leopold. Cray T3D couples MPP, vector technology. *Electronic Engineering Times*, October 1993.

[15] D. Lieberman. Teraflop engine is unveiled. *Electronic Engineering Times*, November 1991.

[16] J. D. Lyle. *SBus: Information, Applications and Experience.* Springer-Verlag, 1992.

[17] NASA launches RS/6000 cluster. *MIDRANGE Systems*, February 1993.

[18] Scientific Computing Associates Inc. Network linda system performance enhancement software. product announcement, May 1991.

[19] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 1989.

[20] Thinking Machines Corporation. *The Connection Machine: CM-5 Technical Summary*, January 1992.

[21] T. von Eicken and D. E. Culler. Building communication paradigms with the CM-5 active message layer (CMAM). Technical report, University of California at Berkeley, July 1992.

[22] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. Technical report, University of California at Berkeley, March 1992.

[23] Ron Wilson. SCI scheme gets interface hardware. *Electronic Engineering Times*, December 1992.

[24] Xilinx. *The Programmable Logic Data Book*, 1993.

Appendix A

CMAM Primitives for FUNet

```
/*********************************************************************
 * source code for FUNet's version of CMAM  primitives:
 *   cmam_4()
 *   cmam_indirect_4()
 *   cmam_xfer_4()
 *   cmam_reply_4()
 *   cmam_reply_indirect_4()
 *   cmam_reply_xfer_4()
 * and FUNet's extensions to CMAM
 *   cmam_n()
 *   cmam_indirect_n()
 *   cmam_mfer_n()
 *   cmam_reply_n()
 *   cmam_reply_indirect_n()
 *   cmam_reply_mfer_n()
 * plus other interface depended handlers and supporting functions.
 *
 * base on:
 *
 * CMAM - CM-5 Active Message layer, V1.99
 *
 * "Copyright (c) 1992 The Regents of the University of California.
 * All rights reserved.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY
 * PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
 * DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS
 * DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE
 * SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE
 * UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE,
 * SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * NOTE on PROTEUS:
 * The @ symbol appearing the code below is equivalent in semantic to
 * the * symbol in C for memory deference.  The @ is a token used by
 * the PROTEUS system to mark memory-references that needs to be
 * simulated.
 ******************************************************************/


/******************************************************************
 * include files
 ******************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#define CMAMSCA
#include "user.h"
#include "sim.h"
#include "funi.user.h"
#include "mmio.h"
/*
 * mmio.h declares int mm_funi which is set to contain the
 *   base memory-mapped address of the FUNi device.
 * (int *)(mm_funi-0x20000) is the base memory-mapped address of
 * the cached memory-mapped registers.
 */
#include "cmam_int.h"


/******************************************************************
 * global variable declaration
 * NOTE: To emulate the correct behavior on a real FUNet cluster, the
 *       global variables have been extended into array to give each
 *       node in the simulation a private copy of the global variable.
 ******************************************************************/
int *p_sq[NO_OF_PROCESSORS], *p_rq[NO_OF_PROCESSORS];
int p_sq_tail[NO_OF_PROCESSORS], p_sq_head[NO_OF_PROCESSORS];
int p_rq_tail[NO_OF_PROCESSORS], p_rq_head[NO_OF_PROCESSORS];
int *q_sq[NO_OF_PROCESSORS], *q_rq[NO_OF_PROCESSORS];
int q_sq_tail[NO_OF_PROCESSORS], q_sq_head[NO_OF_PROCESSORS];
int q_rq_tail[NO_OF_PROCESSORS], q_rq_head[NO_OF_PROCESSORS];


/*******************************************************************
 * pointers to global variables uses in the code body.
 * each pointer corresponds to an global variable above.
 * PROTEUS will automatically maintain the pointers so the correct
 *   elements in the global variable array is accessed by each
 *   simulated  nodes.
 *******************************************************************/
/* base pointer of the reply send queue and receive queue */
int **cmamp_sq,**cmamp_rq;
/* reply send queue tail index and head index */
int *cmamp_sq_tail,*cmamp_sq_head;
/* reply receive queue tail index and head index */
int *cmamp_rq_tail,*cmamp_rq_head;

/* base pointer of the request send queue and receive queue */
int **cmamq_sq,**cmamq_rq;
/* request send queue tail index and head index */
int *cmamq_sq_tail,*cmamq_sq_head;
/* request receive queue tail index and head index *//
int *cmamq_rq_tail,*cmamq_rq_head;


/*******************************************************************
 * CMAM message handlers
 *******************************************************************/

/*******************************************************************
 * basic active message: 4 args
 */
void cmam_handler(int header, void (*fun)(), longlong i1, longlong i2)
{
  /* call *fun with args */
  (*fun)(i1,i2);
}
/*******************************************************************
 * variable-length request active message: 0 to 20 args
 */
void cmam_n_handler(int header, void (*fun)(),
                    longlong i1, longlong i2)
```

```
{
  register int length=NIH_DEC_LEN(header)-1;
  register int count=length-4;
  longlong buffer[12];
  register longlong *ptr=buffer;
  register longlong *addr=
    (longlong*)(ADDR(*cmamq_rq,DEC_BY(*cmamq_rq_tail,1,CMAM_Q_MASK))+
               3);

  /* prepare argument buffer */
  *ptr++=i1;
  *ptr++=i2;
  for(;count>0;count-=4) {
    *ptr++=@addr++;
    *ptr++=@addr++;
  }
  /* call *fun with pointer and size of argument buffer */
  (*fun)(&buffer,length<<2);
}
/*******************************************************************
 * variable-length reply active message: 0 to 20 args
 */
void cmam_reply_n_handler(int header, void (*fun)(),
                          longlong i1, longlong i2)
{
  register int length=NIH_DEC_LEN(header)-1;
  register int count=length-4;
  longlong buffer[12];
  register longlong *ptr=buffer;
  register longlong *addr=
    (longlong*)(ADDR(*cmamp_rq,DEC_BY(*cmamp_rq_tail,1,CMAM_Q_MASK))+
               3);

  /* prepare argument buffer */
  *ptr++=i1;
  *ptr++=i2;
  for(;count>0;count-=4) {
    *ptr++=@addr++;
    *ptr++=@addr++;
  }
  /* call *fun with pointer and size of argument buffer */
  (*fun)(&buffer,length<<2);
}
```

```
/*********************************************************************
 * indirect active message: 4 args
 */
void cmam_indirect_handler(int header, void (**fun)(),
                           longlong i1, longlong i2)
{
  /* call **fun with args */
  (**fun)(fun,i1,i2);
}
/*********************************************************************
 * variable-length request indirect active message: 0 to 20 args
 */
void cmam_indirect_n_handler(int header, void (**fun)(),
                             longlong i1, longlong i2)
{
  register int length=NIH_DEC_LEN(header)-1;
  register int count=length-4;
  longlong buffer[12];
  register longlong *ptr=buffer;
  register longlong *addr=
    (longlong*)(ADDR(*cmamq_rq,DEC_BY(*cmamq_rq_tail,1,CMAM_Q_MASK))+
                3);

  /* prepare argument buffer */
  *ptr++=i1;
  *ptr++=i2;
  for(;count>0;count-=4) {
    *ptr++=@addr++;
    *ptr++=@addr++;
  }
  /* call **fun with pointer and size of argument buffer */
  (**fun)(fun,&buffer,length<<2);
}
/*********************************************************************
 * variable-length reply indirect active message: 0 to 20 args
 */
void cmam_reply_indirect_n_handler(int header, void (**fun)(),
                                   longlong i1, longlong i2)
{
  register int length=NIH_DEC_LEN(header)-1;
  register int count=length-4;
  longlong buffer[12];
  register longlong *ptr=buffer;
  register longlong *addr=
    (longlong*)(ADDR(*cmamp_rq,DEC_BY(*cmamp_rq_tail,1,CMAM_Q_MASK))+
                3);

  /* prepare argument buffer */
  *ptr++=i1;
  *ptr++=i2;
  for(;count>0;count-=4) {
    *ptr++=@addr++;
    *ptr++=@addr++;
  }
  /* call **fun with pointer and size of argument buffer */
  (**fun)(fun,&buffer,length<<2);
}



/* typedef of the data transfer segment structure */
typedef struct {
        void    *abase;  /* aligned transfer base address (base&~7) */
        int     remain;  /* bytes remaining to be transferred */
        int     (*fun)();/* function to call at end of xfer */
        void    *info;   /* argument for end-of-xfer fun */
        void    *base;   /* transfer base address */
        int     pad[3];
} xfer_segment;
/* a PROTEUS pointer to the global variable */
extern  xfer_segment  *cmam_seg;          /* table of all segments */


/*********************************************************************
 * basic data transfer: 16 bytes
 */
void cmam_xfer_handler(int header,unsigned int seg_addr,
                       int i1,int i2,int i3,int i4)
{
  register xfer_segment *seg=((cmam_seg))+(seg_addr>>CMAM_SEG_SHIFT);

  register int address=(seg->abase)+(seg_addr&CMAM_OFF_MASK);

  {
    /* receive 16 bytes into specified segment */
    *(int *)(address) = i1;
    address+=4;
    *(int *)(address) = i2;
    address+=4;
    *(int *)(address) = i3;
    address+=4;
    *(int *)(address) = i4;

    /* update segment */
    if((seg->remain-=16) <= 0)
      cmam_close_segment(seg_addr>>CMAM_SEG_SHIFT);
  }
}
/*********************************************************************
 * variable length DMA data transfer: 0 to 64 bytes
 */
void cmam_mfer_handler(int header, int dummy,
                       int dest, unsigned int seg_addr)
{
  register int length=NIH_DEC_LEN(header);
  register xfer_segment *seg=((cmam_seg))+(seg_addr>>CMAM_SEG_SHIFT);

  /* update segment */
  if((seg->remain-=4*length) <= 0)
    cmam_close_segment(seg_addr>>CMAM_SEG_SHIFT);
}


/*********************************************************************
 * CMAM handler table
 *********************************************************************/
/*********************************************************************
 * low priority packet handler table
 */
void *cmam_low_htab[FUNI_NO_TYPES] = {
  cmam_handler,
  cmam_indirect_handler,
  cmam_xfer_handler,
  cmam_n_handler,
  cmam_indirect_n_handler,
  cmam_mfer_handler,
  NULL,NULL
};
/*********************************************************************
 * low priority packet handler table
 */
void *cmam_high_htab[FUNI_NO_TYPES] = {
  cmam_handler,
  cmam_indirect_handler,
  cmam_xfer_handler,
  cmam_reply_n_handler,
  cmam_reply_indirect_n_handler,
  cmam_mfer_handler,
  NULL,NULL
};
/*********************************************************************
 * packet tag assignment
 */
int cmam_tag=0;
int cmam_indirect_tag=1;
int cmam_xfer_tag=2;
int cmam_n_tag=3;
int cmam_indirect_n_tag=4;
int cmam_mfer_tag=5;


/*********************************************************************
 * network interface servicing functions -- poll and receive
 *********************************************************************/
/*********************************************************************
 * if available, receive one reply priority packet
 */
static inline void cmam_service_reply() {
  register int mmfuni=mm_funi;
  register int *tailptr=cmamp_rq_tail;
  register int *headptr=cmamp_rq_head;
  register int tail=*tailptr;
  register int head=(*headptr);

  /* check for pending reply packet */
  if (tail==head) {
    if (head=@(int*)(mmfuni-0x20000+(NIO_RPRQ_HD<<NIO_REG_OS)),
        tail==head) {
      return;
    }
    (*headptr)=head;
  }

  /* retrieve packet content from rply queue and call handler */
  {
    register longlong *addr=(longlong *)ADDR((*cmamp_rq),tail);
    register longlong dword0=@(addr++);
    register longlong dword1=@(addr++);
    register longlong dword2=@(addr);
    @(int*)(mmfuni+(NIO_RPRQ_TL<<NIO_REG_OS))=
      *tailptr=INC_BY(tail,1,CMAM_Q_MASK);

    (*(void (*)())cmam_high_htab[NIH_DEC_TYPE(*((int *)(&dword0)))])
      (dword1,dword1,dword2);
  }
}
/*********************************************************************
 * receive pending all reply priority packets
 */
static inline void cmam_service_reply_drain() {
  register int mmfuni=mm_funi;
  register int *headptr=(cmamp_rq_head);
  register int *tailptr=(cmamp_rq_tail);
  register int tail=(*tailptr);
  register int head;

  /* check for pending reply packet */
  if (head=@(int*)(mmfuni-0x20000+(NIO_RPRQ_HD<<NIO_REG_OS)),
      tail!=head) {

    register int *base=(*cmamp_rq);
    *headptr=head;

    /* while reply receive queue not empty */
    do {
      /* retrieve packet content from rply queue and call handler */
      register longlong *addr=(longlong *)ADDR(base,tail);
      register longlong dword0=@(addr++);
      register longlong dword1=@(addr++);
      register longlong dword2=@(addr);

      @(int*)(mmfuni+(NIO_RPRQ_TL<<NIO_REG_OS))=
        *tailptr=INC_BY(tail,1,CMAM_Q_MASK);

      (*(void (*)())cmam_high_htab[NIH_DEC_TYPE(*((int *)(&dword0)))])
        (dword0,dword1,dword2);

      tail=*tailptr;
      head=*headptr;
    } while(tail!=head);
  }
}
/*********************************************************************
```

```
 * if available, receive one request priority packet
 */
static inline void cmam_service_request() {
  register int mmfuni=mm_funi;
  register int *tailptr=cmamq_rq_tail;
  register int *headptr=cmamq_rq_head;
  register int tail=*tailptr;
  register int head=(*headptr);

  /* check for pending request packet */
  if (tail==head) {
    if (head=@(int*)(mmfuni-0x20000+(NIO_RQRQ_HD<<NIO_REG_OS)),
        tail==head) {
      return;
    }
    *headptr=head;
  }

  /* retrieve packet content from req queue and call handler */
  {
    register longlong *addr=(longlong *)ADDR((*cmamq_rq),tail);
    register longlong dword0=@(addr++);
    register longlong dword1=@(addr++);
    register longlong dword2=@(addr);

    @(int*)(mmfuni+(NIO_RQRQ_TL<<NIO_REG_OS))=
      *tailptr=INC_BY(tail,1,CMAM_Q_MASK);

    (*(void (*)())cmam_low_htab[NIH_DEC_TYPE(*((int *)(&dword0)))])
      (dword0,dword1,dword2);
  }
}
/*********************************************************************
 * receive all pending request priority packets
 */
static inline void cmam_service_request_drain() {
  register int mmfuni=mm_funi;
  register int *headptr=(cmamq_rq_head);
  register int *tailptr=(cmamq_rq_tail);
  register int tail=(*tailptr);
  register int head;


  /* check for pending request packet */
  if (head=@(int*)(mmfuni-0x20000+(NIO_RQRQ_HD<<NIO_REG_OS)),
      tail!=head) {
    register int *base=(*cmamq_rq);
    *headptr=head;

    /* while request receive queue not empty */
    do {
      /* retrieve packet content from req queue and call handler */
      register longlong *addr=(longlong *)ADDR(base,tail);
      register longlong dword0=@(addr++);
      register longlong dword1=@(addr++);
      register longlong dword2=@(addr);

      @(int*)(mmfuni+(NIO_RQRQ_TL<<NIO_REG_OS))=
        *tailptr=INC_BY(tail,1,CMAM_Q_MASK);

      (*(void (*)())cmam_low_htab[NIH_DEC_TYPE(*((int *)(&dword0)))])
      (dword0,dword1,dword2);

      tail=*tailptr;
      head=*headptr;
    } while(tail!=head);
  }
}
/*******************************************************************
 * if available, receive one reply priority packet
 * if not, receive one request priority packet if available
 */
static inline void cmam_service() {
  register int mmfuni=mm_funi;
  register int *tailptr=cmamp_rq_tail;
  register int *headptr=cmamp_rq_head;
  register int tail=*tailptr;
  register int head=(*headptr);
  register int which=1;

  /* check for pending reply packet */
  if (tail==head) {
    if (head=@(int*)(mmfuni-0x20000+(NIO_RPRQ_HD<<NIO_REG_OS)),
        tail==head) {
      which=0;
    } else {
      (*headptr)=head;
    }
  }
  if (which) {
    /* retrieve packet content from rply queue and call handler */
    register longlong *addr=(longlong *)ADDR((*cmamp_rq),tail);
    register register longlong dword0=@(addr++);
    register register longlong dword1=@(addr++);
    register register longlong dword2=@(addr);

    @(int*)(mmfuni+(NIO_RPRQ_TL<<NIO_REG_OS))=
      *tailptr=INC_BY(tail,1,CMAM_Q_MASK);

    (*(void (*)())cmam_high_htab[NIH_DEC_TYPE(*((int *)(&dword0)))])
      (dword0,dword1,dword2);

  } else {
    tailptr=(cmamq_rq_tail);
    headptr=(cmamq_rq_head);
    tail=(*tailptr);
    head=(*headptr);

    /* check for pending request packet */
    if (tail==head) {
      if (head=@(int*)(mmfuni-0x20000+(NIO_RQRQ_HD<<NIO_REG_OS)),
          tail==head) {
        return;
      }
      (*headptr)=head;
    }
    {
      /* retrieve packet content from req queue and call handler */
      register longlong *addr=(longlong *)ADDR((*cmamq_rq),tail);
      register longlong dword0=@(addr++);
      register longlong dword1=@(addr++);
      register longlong dword2=@(addr);

      @(int*)(mmfuni+(NIO_RQRQ_TL<<NIO_REG_OS))=
        (*tailptr)=INC_BY(tail,1,CMAM_Q_MASK);

      (*(void (*)())cmam_low_htab[NIH_DEC_TYPE(*((int *)(&dword0)))])
        (dword0,dword1,dword2);
    }
  }
}
/*********************************************************************
 * receive all pending packets, reply first then requests
 */
static inline void cmam_service_drain()
```

```
{
  register int mmfuni=mm_funi;
  register int *headptr=(cmamp_rq_head);
  register int *tailptr=(cmamp_rq_tail);
  register int tail=(*tailptr);
  register int head;

  /* check for pending reply packet */
  if (head=@(int*)(mmfuni-0x20000+(NIO_RPRQ_HD<<NIO_REG_OS)),
      tail!=head) {
    register int *base=(*cmamp_rq);

    *headptr=head;

   /* while reply receive queue not empty */
   do {
     /* retrieve packet content from rply queue and call handler */
     register longlong *addr=(longlong *)ADDR((*cmamp_rq),tail);
      register longlong dword0=@(addr++);
      register longlong dword1=@(addr++);
      register longlong dword2=@(addr);

      @(int*)(mmfuni+(NIO_RPRQ_TL<<NIO_REG_OS))=
        (*tailptr)=INC_BY(tail,1,CMAM_Q_MASK);

      (*(void (*)())cmam_high_htab[NIH_DEC_TYPE(*((int *)(&dword0)))])
        (dword0,dword1,dword2);

      tail=(*tailptr);
      head=(*headptr);
    } while(tail!=head);
  }

  tailptr=(cmamq_rq_tail);
  headptr=(cmamq_rq_head);
  tail=(*tailptr);

  /* check for pending request packet */
  if (head=@(int*)(mmfuni-0x20000+(NIO_RQRQ_HD<<NIO_REG_OS)),
      tail!=head) {
    register int *base=(*cmamq_rq);

    *headptr=head;

  /* while request receive queue not empty */
  do {
    /* retrieve packet content from req queue and call handler */
      register longlong *addr=(longlong *)ADDR((*cmamq_rq),tail);
      register longlong dword0=@(addr++);
      register longlong dword1=@(addr++);
      register longlong dword2=@(addr);

      @(int*)(mmfuni+(NIO_RQRQ_TL<<NIO_REG_OS))=
        (*tailptr)=INC_BY(tail,1,CMAM_Q_MASK);

    (*(void (*)())cmam_low_htab[NIH_DEC_TYPE(*((int *)(&dword0)))])
      (dword0,dword1,dword2);

      tail=(*tailptr);
      head=(*headptr);
    } while(tail!=head);
  }
}


/*********************************************************************
 * CMAM sending primitives adapted for FUNi
 *********************************************************************/
/*********************************************************************
 * void cmam_4(int node, void (*fun)(),
 *             int i1, int i2, int i3, int i4)
 * sends a low priority active message
 */
void cmam_4(longlong i0, longlong i1, longlong i2)
{
  register int mmfuni=mm_funi;
  register int temp;
  register int head=(*cmamq_sq_head);
  register int tail=(*cmamq_sq_tail);

  /* check for space in request send queue */
  temp=INC_BY(head,1,CMAM_Q_MASK);
  if (tail==temp) {
    while((*cmamq_sq_tail)=tail=
             @(int *)(mmfuni+(NIO_RQSQ_TL<<NIO_REG_OS)),
          tail==temp) {
      cmam_service();
      head=(*cmamq_sq_head);
      temp=INC_BY(head,1,CMAM_Q_MASK);
      tail=(*cmamq_sq_tail);
      if (tail!=temp) {
        break;
      }
    }
  }

  {
    register longlong *addr=(longlong *)ADDR((*cmamq_sq),head);
    /* formulate header */
    (*(int *)(&i0))=(((*(int *)(&i0))&NIH_PROC_MASK)|
                     (cmam_tag<<NIH_TYPE_OS)|
                     (5<<NIH_LEN_OS));
    /* enqueue packet content */
    @(addr++)=(longlong)i0;
    @(addr++)=(longlong)i1;
    @addr=(longlong)i2;
  }

  /* update FUNi register */
  @(int*)(mmfuni+(NIO_RQSQ_HD<<NIO_REG_OS))=
    (*cmamq_sq_head)=temp;

  cmam_service_drain();
}
/*********************************************************************
 * void cmam_indirect_4(int node, void (**fun)(),
 *                      int i1, int i2, int i3, int i4)
 * sends a low priority indirect active message
 */
void cmam_indirect_4(longlong i0, longlong i1, longlong i2) {
  register int mmfuni=mm_funi;
  register int head=(*cmamq_sq_head);
  register int tail=(*cmamq_sq_tail);
  register temp;

 /* check for space in request send queue */
  temp=INC_BY(head,1,CMAM_Q_MASK);
  if (tail==temp) {
    while((*cmamq_sq_tail)=tail=
            @(int *)(mmfuni+(NIO_RQSQ_TL<<NIO_REG_OS)),
          tail==temp) {
      cmam_service();
      head=(*cmamq_sq_head);
      temp=INC_BY(head,1,CMAM_Q_MASK);
      tail=(*cmamq_sq_tail);
      if (tail!=temp) {
        break;
      }
    }
  }

 {
    /* formulate header */
    register longlong *addr=(longlong *)ADDR((*cmamq_sq),head);
    (*(int *)(&i0))=(((*(int *)(&i0))&NIH_PROC_MASK)|
                     (cmam_indirect_tag<<NIH_TYPE_OS)|
                     (5<<NIH_LEN_OS));
    /* enqueue packet content */
    @(addr++)=(longlong)i0;
    @(addr++)=(longlong)i1;
    @addr=(longlong)i2;
  }

  /* update FUNi register */
  @(int*)(mmfuni+(NIO_RQSQ_HD<<NIO_REG_OS))=(*cmamq_sq_head)=temp;

  cmam_service_drain();
}
/*********************************************************************
 * void cmam_xfer_4(int node, int seg_addr,
 *                        int i1, int i2, int i3, int i4)
 * sends a low priority data transfer message
 */
void cmam_xfer_4(longlong i0, longlong i1, longlong i2) {
  register int mmfuni=mm_funi;
  register int head=(*cmamq_sq_head);
  register int tail=(*cmamq_sq_tail);
  register temp;

 /* check for space in request send queue */
  temp=INC_BY(head,1,CMAM_Q_MASK);
  if (tail==temp) {
    while((*cmamq_sq_tail)=tail=
             @(int *)(mmfuni+(NIO_RQSQ_TL<<NIO_REG_OS)),
          tail==temp) {
      cmam_service();
      head=(*cmamq_sq_head);
      temp=INC_BY(head,1,CMAM_Q_MASK);
      tail=(*cmamq_sq_tail);
      if (tail!=temp) {
        break;
      }
    }
  }

  {
    register longlong *addr=(longlong *)ADDR((*cmamq_sq),head);
    /* formulate header */
    (*(int *)(&i0))=(((*(int *)(&i0))&NIH_PROC_MASK)|
                     (cmam_xfer_tag<<NIH_TYPE_OS)|
                     (5<<NIH_LEN_OS));
    /* enqueue packet content */
    @(addr++)=(longlong)i0;
    @(addr++)=(longlong)i1;
    @addr=(longlong)i2;
  }

  /* update FUNi register */
  @(int*)(mmfuni+(NIO_RQSQ_HD<<NIO_REG_OS))=
    (*cmamq_sq_head)=temp;

  cmam_service_drain();
}
/*********************************************************************
 * void cmam_reply_4(int node, void (*fun)(),
 *                   int i1, int i2, int i3, int i4)
 * sends a high priority active message
 */
void cmam_reply_4(longlong i0, longlong i1, longlong i2)
{
  register int mmfuni=mm_funi;

  register int head=(*cmamp_sq_head);
  register int tail=(*cmamp_sq_tail);
  register temp;

  /* check for space in reply send queue */
  temp=INC_BY(head,1,CMAM_Q_MASK);
  if (tail==temp) {
    while((*cmamp_sq_tail)=tail=
            @(int *)(mmfuni+(NIO_RPSQ_TL<<NIO_REG_OS)),
          tail==temp) {
      cmam_service_reply();
      head=(*cmamp_sq_head);
      temp=INC_BY(head,1,CMAM_Q_MASK);
      tail=(*cmamp_sq_tail);
      if (tail!=temp) {
        break;
      }
    }
  }

  {
    register longlong *addr=(longlong *)ADDR((*cmamp_sq),head);
    /* formulate header */
    (*(int *)(&i0))=(((*(int *)(&i0))&NIH_PROC_MASK)|
                     (cmam_tag<<NIH_TYPE_OS)|
                     (5<<NIH_LEN_OS));
    /* enqueue packet content */
    @(addr++)=(longlong)i0;
    @(addr++)=(longlong)i1;
    @addr=(longlong)i2;
  }

  /* update FUNi register */
  @(int*)(mmfuni+(NIO_RPSQ_HD<<NIO_REG_OS))=
    (*cmamp_sq_head)=temp;

  cmam_service_reply_drain();
}
/*****************************************************************
 * void cmam_reply_indirect_4(int node, void (**fun)(),
 *                     int i1, int i2, int i3, int i4)
 * sends a high priority indirect active message
 */
void cmam_reply_indirect_4(longlong i0, longlong i1, longlong i2) {
  register int mmfuni=mm_funi;

  register int head=(*cmamp_sq_head);
  register int tail=(*cmamp_sq_tail);
  register temp;

  /* check for space in reply send queue */
  temp=INC_BY(head,1,CMAM_Q_MASK);
  if (tail==temp) {
    while((*cmamp_sq_tail)=tail=
            @(int *)(mmfuni+(NIO_RPSQ_TL<<NIO_REG_OS)),
          tail==temp) {
      cmam_service_reply();
      head=(*cmamp_sq_head);
      temp=INC_BY(head,1,CMAM_Q_MASK);
      tail=(*cmamp_sq_tail);
      if (tail!=temp) {
        break;
      }
    }
  }

  {
    /* formulate header */
    register longlong *addr=(longlong *)ADDR((*cmamp_sq),head);
    (*(int *)(&i0))=(((*(int *)(&i0))&NIH_PROC_MASK)|
                     (cmam_indirect_tag<<NIH_TYPE_OS)|
                     (5<<NIH_LEN_OS));
    /* enqueue packet content */
    @(addr++)=(longlong)i0;
    @(addr++)=(longlong)i1;
    @addr=(longlong)i2;
  }

  /* update FUNi register */
  @(int*)(mmfuni+(NIO_RPSQ_HD<<NIO_REG_OS))=
    (*cmamp_sq_head)=temp;

  cmam_service_reply_drain();
}
/*********************************************************************
 * void cmam_reply_xfer_4(int node, int seg_addr,
 *                        int i1, int i2, int i3, int i4)
 * sends a high priority data transfer message
 */
void cmam_reply_xfer_4(longlong i0, longlong i1, longlong i2) {
  register int mmfuni=mm_funi;

  register int head=(*cmamp_sq_head);
  register int tail=(*cmamp_sq_tail);
  register temp;

  /* check for space in reply send queue */
  temp=INC_BY(head,1,CMAM_Q_MASK);
  if (tail==temp) {
    while((*cmamp_sq_tail)=tail=
            @(int *)(mmfuni+(NIO_RPSQ_TL<<NIO_REG_OS)),
          tail==temp) {
      cmam_service_reply();
      head=(*cmamp_sq_head);
      temp=INC_BY(head,1,CMAM_Q_MASK);
      tail=(*cmamp_sq_tail);
      if (tail!=temp) {
        break;
      }
    }
  }

  {
    register longlong *addr=(longlong *)ADDR((*cmamp_sq),head);
    /* formulate header */
   (*(int *)(&i0))=(((*(int *)(&i0))&NIH_PROC_MASK)|
                    (cmam_xfer_tag<<NIH_TYPE_OS)|
                    (5<<NIH_LEN_OS));

    /* enqueue packet content */
  @(addr++)=(longlong)i0;
    @(addr++)=(longlong)i1;
    @addr=(longlong)i2;
  }

  /* update FUNi register */
  @(int*)(mmfuni+(NIO_RPSQ_HD<<NIO_REG_OS))=
    (*cmamp_sq_head)=temp;

  cmam_service_reply_drain();
}


/*********************************************************************
 * FUNet extensions to CMAM sending primitives
 *********************************************************************/
/*********************************************************************
 * void cmam_n(int node, void(*fun)(),
 *             longlong *arg, int nbyte)
 * sends a low priority active message of length 0 to 20
 */
void cmam_n(longlong i0, longlong *buf, int nbyte)
{
  register int mmfuni=mm_funi;
  register int temp;
  register int head=(*cmamq_sq_head);
  register int tail=(*cmamq_sq_tail);
  /* check for space in request send queue */
  temp=INC_BY(head,1,CMAM_Q_MASK);
  if (tail==temp) {
    while((*cmamq_sq_tail)=tail=
            @(int *)(mmfuni+(NIO_RQSQ_TL<<NIO_REG_OS)),
          tail==temp) {
      cmam_service();
      head=(*cmamq_sq_head);
      temp=INC_BY(head,1,CMAM_Q_MASK);
      tail=(*cmamq_sq_tail);
      if (tail!=temp) {
        break;
      }
    }
  }

  {
    register int n=nbyte>>2;
    register longlong *addr=(longlong *)ADDR((*cmamq_sq),head);
    register longlong *arg=buf;

    /* formulate header */
    (*(int *)(&i0))=(((*(int *)(&i0))&NIH_PROC_MASK)|
                     (cmam_n_tag<<NIH_TYPE_OS)|
                     (((n&0x1f)+1)<<NIH_LEN_OS));

    /* enqueue packet content */
    @(addr++)=(longlong)i0;
    for(;n>0;n-=4) {
      @(addr++)=*(arg);
      arg++;
      @(addr++)=*(arg);
      arg++;
    }
  }

  /* update FUNi register */
  @(int*)(mmfuni+(NIO_RQSQ_HD<<NIO_REG_OS))=
    (*cmamq_sq_head)=temp;

  cmam_service_drain();
}
/*********************************************************************
 * void cmam_indirect_n(int node, void(**fun)(),
 *                      longlong *arg, int nbyte)
 * sends a low priority indirect active message of length 0 to 20
 */
void cmam_indirect_n(longlong i0, longlong *buf, int nbyte)
{
  register int mmfuni=mm_funi;
  register int temp;
  register int head=(*cmamq_sq_head);
  register int tail=(*cmamq_sq_tail);

  /* check for space in request send queue */
  temp=INC_BY(head,1,CMAM_Q_MASK);
  if (tail==temp) {
    while((*cmamq_sq_tail)=tail=
            @(int *)(mmfuni+(NIO_RQSQ_TL<<NIO_REG_OS)),
          tail==temp) {
      cmam_service();
      head=(*cmamq_sq_head);
      temp=INC_BY(head,1,CMAM_Q_MASK);
      tail=(*cmamq_sq_tail);
      if (tail!=temp) {
        break;
      }
    }
  }

  {
    register int n=nbyte>>2;
    register longlong *addr=(longlong *)ADDR((*cmamq_sq),head);
    register longlong *arg=buf;

    /* formulate header */
    (*(int *)(&i0))=(((*(int *)(&i0))&NIH_PROC_MASK)|
                     (cmam_indirect_n_tag<<NIH_TYPE_OS)|
                     (((n&0x1f)+1)<<NIH_LEN_OS));

    /* enqueue packet content */
    @(addr++)=(longlong)i0;
    for(;n>0;n-=4) {
      @(addr++)=*(arg);
      arg++;
      @(addr++)=*(arg);
      arg++;
    }
  }
  /* update FUNi register */
  @(int*)(mmfuni+(NIO_RQSQ_HD<<NIO_REG_OS))=
    (*cmamq_sq_head)=temp;

  cmam_service_drain();
}
/*********************************************************************
 * void cmam_mfer_n(int node, unsigned int seg_addr,
 *                  void *source, void *dest, int nbyte) {
 * sends a low priority DMA transfer message of size 0 to 16
 */
void cmam_mfer_n(int node, unsigned int seg_addr,
                 void *source, void *dest, int nbyte) {
  register int mmfuni=mm_funi;
  register int head=(*cmamq_sq_head);
  register int tail=(*cmamq_sq_tail);
  register temp;

  /* check for space in request send queue */
  temp=INC_BY(head,1,CMAM_Q_MASK);
  if (tail==temp) {
    while((*cmamq_sq_tail)=tail=
            @(int *)(mmfuni+(NIO_RQSQ_TL<<NIO_REG_OS)),
          tail==temp) {
      cmam_service();
      head=(*cmamq_sq_head);
      temp=INC_BY(head,1,CMAM_Q_MASK);
      tail=(*cmamq_sq_tail);
      if (tail!=temp) {
        break;
      }
    }
  }

  {
    register longlong *addr=(longlong *)ADDR((*cmamq_sq),head);
    longlong data;

    /* formulate header */
    (*(int *)(&data))=((node&NIH_PROC_MASK)|
                       (cmam_mfer_tag<<NIH_TYPE_OS)|
                       (((nbyte>>2)&0x1f)<<NIH_LEN_OS)|
                       NIH_MODE_MASK);
    (*((int *)(&data)+1))=source;
    /* enqueue remote DMA request */
    @(addr++)=(longlong)data;
    (*(int *)(&data))=dest;
    (*((int *)(&data)+1))=seg_addr;
    @(addr)=(longlong)data;
  }

  /* update FUNi register */
  @(int*)(mmfuni+(NIO_RQSQ_HD<<NIO_REG_OS))=
    (*cmamq_sq_head)=temp;

  cmam_service_drain();
}
/*******************************************************************
 * void cmam_reply_n(int node, void(*fun)(),
 *                   longlong *arg, int nbyte)*/
 * sends a high priority active message of length 0 to 20
 */
void cmam_reply_n(longlong i0, longlong *buf, int nbyte)
{
  register int mmfuni=mm_funi;
  register int temp;
  register int head=(*cmamp_sq_head);
  register int tail=(*cmamp_sq_tail);

  /* check for space in reply send queue */
  temp=INC_BY(head,1,CMAM_Q_MASK);
  if (tail==temp) {
    while((*cmamp_sq_tail)=tail=
             @(int *)(mmfuni+(NIO_RPSQ_TL<<NIO_REG_OS)),
          tail==temp) {
      cmam_service_reply();
      head=(*cmamp_sq_head);
      temp=INC_BY(head,1,CMAM_Q_MASK);
      tail=(*cmamp_sq_tail);
      if (tail!=temp) {
        break;
      }
    }
  }

  {
    register int n=nbyte>>2;
    register longlong *addr=(longlong *)ADDR((*cmamp_sq),head);
    register longlong *arg=buf;
    /* formulate header */
    (*(int *)(&i0))=(((*(int *)(&i0))&NIH_PROC_MASK)|
                    (cmam_n_tag<<NIH_TYPE_OS)|
                    (((n&0x1f)+1)<<NIH_LEN_OS));

    /* enqueue packet content */
    @(addr++)=(longlong)i0;
    for(;n>0;n-=4) {
      @(addr++)=*(arg);
      arg++;
      @(addr++)=*(arg);
      arg++;
    }
  }

  /* update FUNi register */
  @(int*)(mmfuni+(NIO_RPSQ_HD<<NIO_REG_OS))=
    (*cmamp_sq_head)=temp;

  cmam_service_reply_drain();
}
/**********************************************************************
 * void cmam_indirect_reply_n(int node, void(**fun)(),
 *                            longlong *arg, int nbyte)
 * sends a high priority active message of length 0 to 20
 */
void cmam_indirect_reply_n(longlong i0, longlong *buf, int nbyte)
{
  register int mmfuni=mm_funi;
  register int temp;
  register int head=(*cmamp_sq_head);
  register int tail=(*cmamp_sq_tail);

  /* check for space in reply send queue */
  temp=INC_BY(head,1,CMAM_Q_MASK);
  if (tail==temp) {
    while((*cmamp_sq_tail)=tail=
            @(int *)(mmfuni+(NIO_RPSQ_TL<<NIO_REG_OS)),
          tail==temp) {
      cmam_service_reply();
      head=(*cmamp_sq_head);
      temp=INC_BY(head,1,CMAM_Q_MASK);
      tail=(*cmamp_sq_tail);
      if (tail!=temp) {
        break;
      }
    }
  }

  {
    register int n=nbyte>>2;
    register longlong *addr=(longlong *)ADDR((*cmamp_sq),head);
    register longlong *arg=buf;

    /* formulate header */
    (*(int *)(&i0))=(((*(int *)(&i0))&NIH_PROC_MASK)|
                     (cmam_indirect_n_tag<<NIH_TYPE_OS)|
                     (((n&0x1f)+1)<<NIH_LEN_OS));

    /* enqueue packet content */
    @(addr++)=(longlong)i0;
    for(;n>0;n-=4) {
      @(addr++)=*(arg);
      arg++;
      @(addr++)=*(arg);
      arg++;
    }
  }

  /* update FUNi register */
  @(int*)(mmfuni+(NIO_RPSQ_HD<<NIO_REG_OS))=
    (*cmamp_sq_head)=temp;

  cmam_service_reply_drain();
}
/*******************************************************************
 * void cmam_reply_mfer_n(int node, unsigned int seg_addr,
 *                    void *source, void *dest, int nbyte) {
 * sends a high priority DMA transfer message of size 0 to 16
 */
void cmam_reply_mfer_n(int node, unsigned int seg_addr,
                       void *source, void *dest, int nbyte) {
  register int mmfuni=mm_funi;
  register int head=(*cmamp_sq_head);
  register int tail=(*cmamp_sq_tail);
  register temp;

  /* check for space in reply send queue */
  temp=INC_BY(head,1,CMAM_Q_MASK);
  if (tail==temp) {
    while((*cmamp_sq_tail)=tail=
            @(int *)(mmfuni+(NIO_RPSQ_TL<<NIO_REG_OS)),
          tail==temp) {
      cmam_service_reply();
      head=(*cmamp_sq_head);
      temp=INC_BY(head,1,CMAM_Q_MASK);
      tail=(*cmamp_sq_tail);
      if (tail!=temp) {
        break;
      }
    }
  }

  {
    register longlong *addr=(longlong *)ADDR((*cmamp_sq),head);
    longlong data;

    /* formulate header */
    (*(int *)(&data))=((node&NIH_PROC_MASK)|
                       (cmam_mfer_tag<<NIH_TYPE_OS)|
                       (((nbyte>>2)&0x1f)<<NIH_LEN_OS)|
                       NIH_MODE_MASK);
    (*((int *)(&data)+1))=source;

    /* enqueue remote DMA request */
    @(addr++)=(longlong)data;
    (*(int *)(&data))=dest;
    (*((int *)(&data)+1))=seg_addr;
    @(addr)=(longlong)data;
  }

  /* update FUNi register */
  @(int*)(mmfuni+(NIO_RPSQ_HD<<NIO_REG_OS))=
    (*cmamp_sq_head)=temp;

  cmam_service_reply_drain();
}
```

Appendix B

Context Switching FUNi

```
/*******************************************************************
 * A simple-minded instantiation of the context switching operation
 *
 *******************************************************************/

/*******************************************************************
 * we will assume the following struct for holding the FUNi hardware
 * stats has been added to a parallel process's context block
 *******************************************************************/

tyepdef struct {
  /* register states */
  int GID;
  int TICKET;
  int CNTL;

  int QMASK;

  int RPSQ_BASE;
  int RPSQ_TL;
  int RPSQ_HD;

  int RQSQ_BASE;
  int RQSQ_TL;
  int RQSQ_HD;

  int RPRQ_BASE;
  int RPRQ_HD;
  int RPRQ_TL;
  int RQRQ_BASE;
  int RQRQ_HD;
  int RQRQ_TL;

  /* special drain receive buffers*/
  int RP_Q[SIZE_OF_INTERNAL_BUFFER];
  int RP_Q_HD;
  int RQ_Q[SIZE_OF_INTERNAL_BUFFER];
  int RQ_Q_HD;
} FUNicontex;

/********************************************************************
 * declaration for pointers to the exiting and the next processes'
 * FUNi context block.
 ********************************************************************/
FUNicontex *exit_ctx;
FUNicontex *next_ctx;

\begin{singlespacing}
\begin{verbatim}

/********************************************************************
 * begin context switching FUNi
 ********************************************************************/

{
  /* begin context switch mode */
  exit_ctx->FUNictx.CNTL=
    *(int *)(mm_FUNi_base+(NIO_CNTL<<NIO_REG_OS));

  *(int *)(mm_FUNi_base+(NIO_CNTL<<NIO_REG_OS))=
    (NIC_CTXM_MASK | NIC_CRQM_MASK);

  while (!(*(int *)(mm_FUNi_base+(NIO_CNTL<<NIO_REG_OS)) &
           NIC_CSQR_MASK));

  /* save send Queue Registers */
  {
    exit_ctx->FUNictx.QMASK=
      *(int *)(mm_FUNi_base+(NIO_QMASK<<NIO_REG_OS));

    exit_ctx->FUNictx.RPSQ_BASE=
      *(int *)(mm_FUNi_base+(NIO_RPSQ_BASE<<NIO_REG_OS));
    exit_ctx->FUNictx.RPSQ_TL=
    *(int *)(mm_FUNi_base+(NIO_RPSQ_TL<<NIO_REG_OS));
  exit_ctx->FUNictx.RPSQ_HD=
    *(int *)(mm_FUNi_base+(NIO_RPSQ_HD<<NIO_REG_OS));

  exit_ctx->FUNictx.RQSQ_BASE=
    *(int *)(mm_FUNi_base+(NIO_RQSQ_BASE<<NIO_REG_OS));
  exit_ctx->FUNictx.RQSQ_TL=
    *(int *)(mm_FUNi_base+(NIO_RQSQ_TL<<NIO_REG_OS));
  exit_ctx->FUNictx.RQSQ_HD=
    *(int *)(mm_FUNi_base+(NIO_RQSQ_HD<<NIO_REG_OS));

  while (!(*(int *)(mm_FUNi_base+(NIO_CNTL<<NIO_REG_OS)) &
           NIC_CTXR1_MASK));
  exit_ctx->FUNictx.TICKET=
    *(int *)(mm_FUNi_base+(NIO_TICKET<<NIO_REG_OS));
}

while (!(*(int *)(mm_FUNi_base+(NIO_CNTL<<NIO_REG_OS)) &
         NIC_CRQR_MASK));

/* save receive Queue Registers */
{
  exit_ctx->FUNictx.RPRQ_BASE=
    *(int **)(mm_FUNi_base+(NIO_RPRQ_BASE<<NIO_REG_OS));
  exit_ctx->FUNictx.RPRQ_HD=
    *(int *)(mm_FUNi_base+(NIO_RPRQ_HD<<NIO_REG_OS));
  exit_ctx->FUNictx.RPRQ_TL=
    *(int *)(mm_FUNi_base+(NIO_RPRQ_TL<<NIO_REG_OS));

  exit_ctx->FUNictx.RQRQ_BASE=
    *(int **)(mm_FUNi_base+(NIO_RQRQ_BASE<<NIO_REG_OS));
  exit_ctx->FUNictx.RQRQ_HD=
    *(int *)(mm_FUNi_base+(NIO_RQRQ_HD<<NIO_REG_OS));
  exit_ctx->FUNictx.RQRQ_TL=
    *(int *)(mm_FUNi_base+(NIO_RQRQ_TL<<NIO_REG_OS));
}

/* swap in the drain queues to draining hardware rcv buffers */
{
  *(int *)(mm_FUNi_base+(NIO_QMASK<<NIO_REG_OS))=
    SIZE_OF_INTERNAL_BUFFER-1;
  *(int **)(mm_FUNi_base+(NIO_RPRQ_BASE<<NIO_REG_OS))=
    &exit_ctx->FUNictx.RP_Q[0];
  *(int **)(mm_FUNi_base+(NIO_RQRQ_BASE<<NIO_REG_OS))=
    &exit_ctx->FUNictx.RQ_Q[0];
    *(int *)(mm_FUNi_base+(NIO_CNTL<<NIO_REG_OS))=
      (NIC_CTXM_MASK | NIC_DRAIN_MASK);

    /* wait to drain */
    while (!(*(int *)(mm_FUNi_base+(NIO_CNTL<<NIO_REG_OS)) &
             NIC_CTXR2_MASK));

    exit_ctx->FUNictx.RP_Q_HD=
      *(int *)(mm_FUNi_base+(NIO_RPRQ_HD<<NIO_REG_OS));
    exit_ctx->FUNictx.RQ_Q_HD=
      *(int *)(mm_FUNi_base+(NIO_RQRQ_HD<<NIO_REG_OS));
}
/* FUNi context saving completed */
/* no more DVMA to user memory of last context */
    .
    .
    .
    other context switching tasks
    .
    .
    .

/* FUNi context restoring begin */

/* restore packets drained before exit to self */
{
    *(int *)(mm_FUNi_base+(NIO_TICKET<<NIO_REG_OS))=0x0;

    *(int **)(mm_FUNi_base+(NIO_RPSQ_BASE<<NIO_REG_OS))=
      &next_ctx->FUNictx.RP_Q[0];
    *(int *)(mm_FUNi_base+(NIO_RPSQ_HD<<NIO_REG_OS))=
      next_ctx->FUNictx.RP_Q_HD;

    *(int **)(mm_FUNi_base+(NIO_RQSQ_BASE<<NIO_REG_OS))=
      &next_ctx->FUNictx.RP_Q[0];
    *(int *)(mm_FUNi_base+(NIO_RQSQ_HD<<NIO_REG_OS))=
      next_ctx->FUNictx.RP_Q_HD;

    *(int *)(mm_FUNi_base+(NIO_GID<<NIO_REG_OS))=RESERVED_GID;
    *(int *)(mm_FUNi_base+(NIO_CNTL<<NIO_REG_OS))=NIC_CRQM_MASK;

    while (*(int *)(mm_FUNi_base+(NIO_RPSQ_TL<<NIO_REG_OS)) !=
           next_ctx->FUNictx.RP_Q_HD);
    while (*(int *)(mm_FUNi_base+(NIO_RQSQ_TL<<NIO_REG_OS)) !=
        next_ctx->FUNictx.RQ_Q_HD);

  while (*(int *)(mm_FUNi_base+(NIO_TICKET<<NIO_REG_OS)) !=
        0xff);

  *(int *)(mm_FUNi_base+(NIO_CNTL<<NIO_REG_OS))=
    NIC_CTXM_MASK | NIC_CRQM_MASK;
}

/* restore registers for next context*/
{

  *(int *)(mm_FUNi_base+(NIO_TICKET<<NIO_REG_OS))=
    next_ctx->FUNictx.TICKET;
  *(int *)(mm_FUNi_base+(NIO_QMASK<<NIO_REG_OS))=
    next_ctx->FUNictx.QMAKSK;

  *(int **)(mm_FUNi_base+(NIO_RPSQ_BASE<<NIO_REG_OS))=
    next_ctx->FUNictx.RPSQ_BASE;
  *(int *)(mm_FUNi_base+(NIO_RPSQ_TL<<NIO_REG_OS))=
    next_ctx->FUNictx.RPSQ_TL;
  *(int *)(mm_FUNi_base+(NIO_RPSQ_HD<<NIO_REG_OS))=
    next_ctx->FUNictx.RPSQ_HD;

  *(int **)(mm_FUNi_base+(NIO_RQSQ_BASE<<NIO_REG_OS))=
    next_ctx->FUNictx.RPSQ_BASE;
  *(int *)(mm_FUNi_base+(NIO_RQSQ_TL<<NIO_REG_OS))=
    next_ctx->FUNictx.RPSQ_TL;
  *(int *)(mm_FUNi_base+(NIO_RQSQ_HD<<NIO_REG_OS))=
    next_ctx->FUNictx.RPSQ_HD;

  *(int **)(mm_FUNi_base+(NIO_RPRQ_BASE<<NIO_REG_OS))=
    next_ctx->FUNictx.RPRQ_BASE;
  *(int *)(mm_FUNi_base+(NIO_RPRQ_HD<<NIO_REG_OS))=
    next_ctx->FUNictx.RPRQ_HD;
  *(int *)(mm_FUNi_base+(NIO_RPRQ_TL<<NIO_REG_OS))=
    next_ctx->FUNictx.RPRQ_TL;

  *(int **)(mm_FUNi_base+(NIO_RQRQ_BASE<<NIO_REG_OS))=
    next_ctx->FUNictx.RPRQ_BASE;
  *(int *)(mm_FUNi_base+(NIO_RQRQ_HD<<NIO_REG_OS))=
    next_ctx->FUNictx.RPRQ_HD;
  *(int *)(mm_FUNi_base+(NIO_RQRQ_TL<<NIO_REG_OS))=
  next_ctx->FUNictx.RPRQ_TL;
}
```

```
  /* context switch complete, restoring the cntl register */
  *(int *)(mm_FUNi_base+(NIO_GID<<NIO_REG_OS))=
    next_ctx->FUNictx.GID;
  *(int *)(mm_FUNi_base+(NIO_CNTL<<NIO_REG_OS))=
    next_ctx->FUNictx.CNTL;
}
```

Appendix C

Matrix Multiply Loop

```
/*******************************************************************
 * Matrix-multiply inner loop
 *******************************************************************/


/*******************************************************************
 * global variable declaration
 * NOTE: To emulate the correct behavior on a real FUNet cluster, the
 *       global variables have been extended into array to give each
 *       node in the simulation a private copy of the global variable.
 * local global variable is referenced by X[processor_]
 *******************************************************************/
/* pointers to the local columns of matrix A, B and C*/
double *A[NO_OF_PROCESSORS],*B[NO_OF_PROCESSORS],*C[NO_OF_PROCESSORS];


/*******************************************************************
 * int end_transfer()
 * termination function to be called when transfer to an opened
 * segment is complete
 *******************************************************************/
int end_transfer(void *info, void *base) {
  (*(int*)info)++;
  return(0);
}


/*******************************************************************
```

```
 * void get_handler()
 * transfer "byte_count" bytes starting at (double*A)+offset
 *   to "seg_addr" of node "who"
 *
 * called as an handler in active message to retrieve columns of A
 *******************************************************************/
void get_handler(int seg_addr, int offset, int who, int byte_count) {
  cmam_reply_xfer(who,seg_addr,A[processor_]+offset,byte_count);
}


/*******************************************************************
 *matrix multiply inner loop
 * A[N][R] x B[R][M] = C[N][M]
 *******************************************************************/
void matrix(int N,int R,int M) {
  double *b=B[processor_];
  double *c=C[processor_];

  int i, j, k;

  int j0, dj, nj;

  int Rp=R/cmam_partition_size;

  double v0[N], v1[N];

  double *v=v0, *nv=v1, *tv;

  int flag=0;

  int seg_addr;

  j0=cmam_self_address*Rp;

  /* open a segment for transfer */
  while((seg_addr= cmam_open_segment(nv,N*sizeof(double),
                                   end_transfer,&flag))==-1) {
    cmam_poll();
  }

  /* fetch first column of A from self */
  cmam_4(cmam_self_address,
         get_handler,
         seg_addr,
        0*N,
        cmam_self_address,
        N*sizeof(double));

  /* main loop: for all columns in A*/
  for(dj=0;dj<R;dj++) {

    j=(j0+dj)%R;
    nj=(j0+dj+1)%R;

    /* wait for the fetch request to be satisfied */
    cmam_wait(&flag,1);

    tv=v;
    v=nv;
    nv=tv;

    /* if not done, proceed with requesting the next column of A */
    if (nj!=j0) {
      while((seg_addr= cmam_open_segment(nv,N*sizeof(double),
                                         end_transfer,&flag))==-1) {
        cmam_poll();
      }
      cmam_4(nj/Rp,
             get_handler,
             seg_addr,
             N*(nj%Rp),
             cmam_self_address,
             N*sizeof(double));
    }

    /*
     * update C based on the last fetched column of A while fetching
     *   the next column of A
     */
    for(k=0; k<M/cmam_partition_size; k++) {
      for(i=0;i<N;i++) {
        c[k*N+i]=c[k*N+i]+v[i]*b[k*R+j];
      }
      cmam_poll();
    }
  }
}
```