Network Interface for Message-Passing Parallel Computation on a Workstation Cluster

James C. Hoe
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
jhoe@abp.lcs.mit.edu

Abstract

As commercial microprocessors become increasingly popular in current MPP architectures, high-performance commercial workstations have also received increased attention as cost-effective building blocks for large parallel-processing systems. The Fast User-level Network (FUNet) project [10] is an attempt at constructing an inexpensive workstation-based parallel system capable of supporting efficient execution of message-passing parallel programs. Based on MIT's Arctic [1] network technology, FUNet connects stock-configured commodity workstations with a high-bandwidth packet-switched routing network. The Fast User-level Network Interface (FUNi) is the custom hardware network interface device that provides access to FUNet for both message passing and remote direct-memory-access (DMA) block transfers between parallel peer processes on FUNet-connected workstations. The FUNi hardware mechanisms allow direct low-overhead user-level accesses 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. The relaxed clock speed (25MHz max.) of SBus allows FUNi to be inexpensively implemented using FPGA parts that are synthesized from designs captured in Verilog Hardware Description Language [15]. SBus's Direct Virtual Memory Access (DVMA)[7] also assists FUNi in overcoming the performance limitations imposed by existing workstation 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.

1 Introduction

In a cluster of workstations connected by a traditional LAN, the scalability and granularity of parallel processing are heavily restricted by the cost of interprocessor communication. The Fast User-level Network (FUNet) [10] is an attempt at constructing an efficient workstation-based message-passing parallel system by augmenting a LAN-connected cluster with an additional high-performance user-level communication network suitable for parallel computation. By providing the means for low-overhead interworkstation communication, we are able to leverage the engineering effort of the workstation industry to construct a low-cost workstation-based parallel system that can rival existing MPP architectures in speedup and performance.

The benefit of these inexpensive, fully engineered, high-performance commercial workstations comes at the cost of the fixed, pre-defined hardware configuration. Thus, although FUNet is able to carry over existing MPP routing and interconnecting technology directly, the design of the Fast User-level Network Interface (FUNi) requires careful rethinking to accommodate the constraints of stock hardware. Namely, FUNi needs to deal with the lack of a specifically designed, tightly coupled interface to the microprocessor. The network interface for a workstation must be relegated to a memory-mapped device in an existing peripheral slot on the memory or even the backplane I/O bus. The long latency of accessing such a network interface device could have a strong impact on the cost of interprocessor communication and, therefore, on the efficiency of parallel execution.

This paper describes FUNi and provides the rationale for its design. Section 2 describes the message interface model which FUNi uses and explains how it helps to cope with limitations of commercial hardware. Section 3 describes features of FUNi for supporting time sharing of the processor and network resources. Section 4 presents FUNi's interface and datapath design in more specific detail. Section 5 evaluates the performance of a FUNet parallel cluster by comparing a simulated FUNet cluster with a contemporary massively parallel computer, CM-5. Section 7 discusses other related work in network interface design for workstation-based parallel processing. This paper concludes with Section 8.

2 Network Interface for Stock Workstation

Interprocessor communication incurs cost – extra processor cycles – due to communication overhead and latency. To send an interprocessor message, the processor must spend overhead cycles to compose the outgoing message and make the message available to the network. A similar overhead is also incurred on the receiving processor to receive the incoming message into computation. Communication latency can lead to extra idle processor cycles if the computation thread depends on a pending incoming message. To achieve fine-grain parallelization where communication and synchronization are frequent, the effects of both overhead and latency must be minimized.

2.1 Design Objective: Minimizing Communication Overhead

Communication latency is less dependent on the network implementation. For example in a common scenario where one processor requests for a remote fetch, the latency of the fetch is a function of how soon the data actually becomes available on the remote processor and how fast the remote processor processes the request. The round-trip transit latency of the request and reply messages on the network is negligible by comparison. The software component of communication latency not only tends to be large but is also less predictable. Fortunately, split-phased transactions [2] can allow us to tolerate the effects of communication latency by overlapping the communication with useful computations. Instead of stalling for the requested data, a processor could perform other independent computations and continue the thread requesting the data only when the fetch has been satisfied. Thus, communication latency, a software problem with a software solution, is the less important factor in the network interface design.

Communication overhead, on the other hand, is heavily dependent on the network and the network interface design. Communication overhead steals real processor cycles from useful computations. If the communication overhead is not kept minimal, processor utilization will degrade in fine-grain parallel programs because the overhead cycles overwhelm the relatively short computation threads. Unfortunately, there is no simple way to mask the effects of communication overhead. Thus, the key design goal of FUNi is to minimize – within the design space allowed by existing workstation architectures – the processor overhead of sending and receiving interworkstation messages, even at the cost of increased latency.

2.2 Shortcoming of Memory-Mapped Network Interface

As mentioned previously, a network interface for stock workstations can only communicate with the processor through a bus. A straightforward message-passing interface could be implemented as memory-mapped registers such as in CM-5 [14], or as a packet-sized array of memory-mapped registers as suggested by Joerg and Henry [9] (Figure 1). These interfaces are passive devices that only respond to the processor's direct manipulation through memory-mapped operations. A user program composes an outbound packet by writing the content of the packet, with its header, to the registers through memory-mapped writes. The content of the register or the register array is formatted as a packet and enqueued into the outbound network buffer. Conversely, user processes can receive and read an inbound packet in the receive registers by memory-mapped reads. Without specific provision to speed up the memory-mapped access path in stock workstations, the long latency – typically on the order of tens of cycles per access – to reach the network interface quickly adds up to tremendous overhead cycles when sending or receiving a message. This effect is especially noticeable in the contemporary RISC microprocessor design whose memory system is optimized for cached memory accesses.

2.3 Active Network Interface Device

As an active device with DMA capability, FUNi logically extends the register array and the network buffers into the user's virtual memory space. Figure 2 illustrates this idea. In the memory-mapped designs described in the previous section, a user enqueues an outbound packet, through memory-mapped writes, into the hardware network buffer directly, and the network interface locates pending outbound packets by dequeuing them from the hardware buffer. A similar effect can be achieved with software enforced circular queues allocated in the user's virtual memory space and jointly maintained by the user program and FUNi. Instead of enqueuing into the hardware buffer through memory-mapped writes, the user process would enqueue the outbound packet into the head of a circular queue in user's virtual memory. FUNi would then retrieve the pending outbound packets from the tail of the circular queue by DMA. A similar transformation can also be made for the receive registers and inbound network buffer.

Moving the messaging interface into the memory system has the benefit of decoupling the processor overhead of communication from the bandwidth and latency of accessing a memory-mapped network interface device. Long latency memory-mapped accesses by the processor are replaced by normal memory accesses that are supported by caching techniques. The user process can enqueue and dequeue outbound and inbound packets at its own rate independent of the bandwidth that is available to FUNi. In addition to reducing messaging overhead, extending the network buffers into the user memory space also 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 amount of FUNi hardware states that needs to be context switched is fixed. The logical size of the buffers can be arbitrarily enlarged in the paged virtual memory. This provides the buffer size necessary for the program execution to tolerate the possibly irregular network traffic on our highly distributed parallel system in which fine-grained coordination of peer processes is difficult.

3 Hardware Support for Time Sharing

FUNet is designed to allow multiple parallel applications to time-share the network and processor resources while providing each application the illusion of a private and reliable network. The primary concern with network security is the privacy of communication which is maintained through automatic tagging of network packets. Another concern in sharing is gross performance degradation, or even deadlock, of the network by a single participant. This issue is addressed in FUNet by the Acknowledgment/Retry End-to-End Flow Control Protocol carried out by FUNi.

3.1 Network Privacy and Authenticity

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. To avoid the high overhead cost of system calls, protection and authentication of user-level communications are enforced by the FUNi hardware.

For each parallel application on FUNet, the operating systems on all participating nodes of FUNet collectively assign a unique Group Identifier (GID). When the process of a parallel application is switched in on a workstation, the operating system makes the corresponding GID available to FUNi. During the time-slice of the process, every outbound packet is automatically tagged with the GID. When the packet arrives at its destination, the receiving FUNi compares the GID tag of the inbound packet against the local

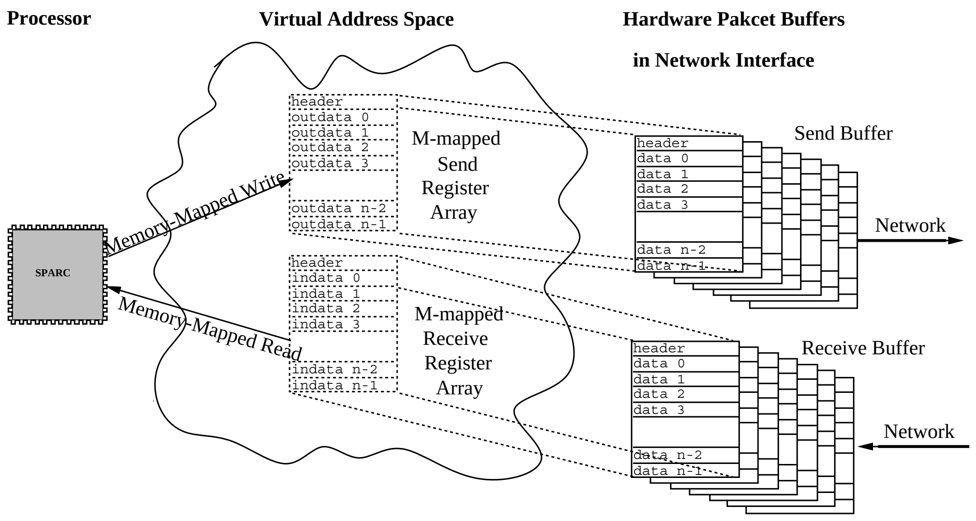

Figure 1: A Message Interface based on Memory-Mapped Register Arrays

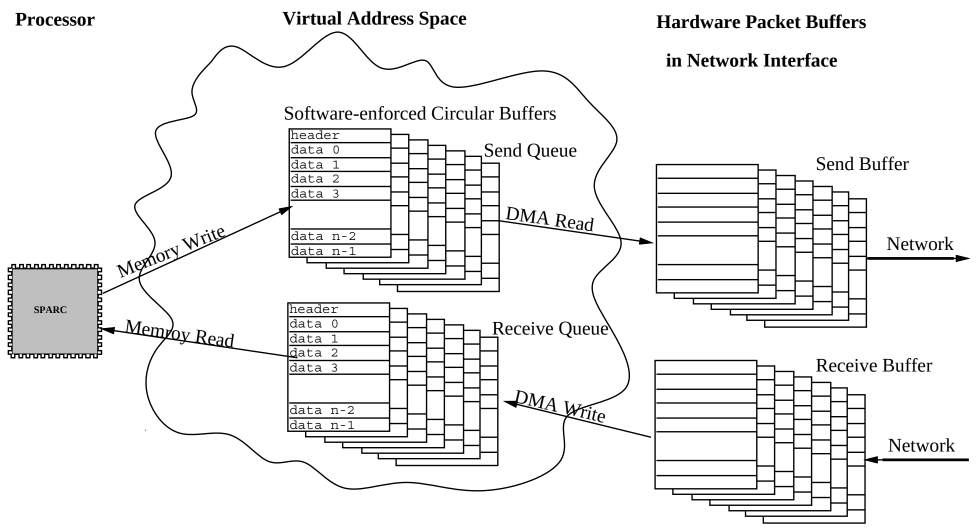

Figure 2: A Message Interface based on Software Queues in User's Virtual Memory

GID. The inbound packet is delivered to the executing process only if a match is made. In the case of a mismatched GID, indicating the correct receiving process is not presently executing, the inbound packet is not delivered. Under the flow control protocol of FUNet, 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 cooperating peer processes of the same application whom all share the same application GID.

3.2 Acknowledgment/Retry End-to-End Flow Control

An Acknowledgment/Retry End-to-End Flow Control Protocol, a simplification of Selective Repeat Protocol [13], is carried out by the FUNi hardware transparently from user programs. When FUNi absorbs an inbound packet from the network, it needs to return an acknowledgment to the originating FUNi. If FUNi accepts the packet, a positive acknowledgment is sent back to acknowledge the acceptance. If FUNi cannot accept the packet for any reason, a negative acknowledgment needs to be returned to the originating FUNi to request a retry. The originating FUNi is responsible for buffering each of its outbound packets until the packet is accepted and positively acknowledged.

With the ability to reject incoming packets, FUNi at each node can continuously absorb packets from the network even with only finite storage. Because network packets are continuously absorbed, the network will neither deadlock nor block regardless of the individual behavior at each node. FUNi can never be indirectly blocked from communication by other misbehaving communication pairs. A FUNi can only be denied from sending if its own receivers are not accepting the inbound packets, thus causing the sending FUNi to run out of buffering resources for outbound packets that are pending delivery. Thus, this mechanism also serves as an automatic rate control for throttling network activities of over-active sending processes, preventing them from swamping other processors with messages.

4 FUNi

FUNi will be implemented as peripheral cards for SBus [7]. The SBus card is chosen as the target FUNi implementation primarily for the SBus's DVMA (Direct Virtual Memory Accesses) feature that is crucial to FUNi's programming interface. The ease of implementation is also a major consideration that stood in favor of the SBus. The SBus compatibility also allows FUNi to work directly in a wide range of SBus-equipped commercial workstation platforms. The implementation of custom logic on the FUNi card abandons the traditional schematic capturing process. Instead, designs will be entered in Verilog Hardware Description Language and compiled into the appropriate netlists by Synopsys HDL Compiler. The current plan calls for implementation using the Xilinx 4000 Family of Field Programmable Gate Arrays (FPGA) [17]. The reprogrammability of the FPGA firmware will allow rapid revisioning of the FUNi hardware during the hardware development and future studies. This section starts by describing the FUNi messaging interface and then describes the hardware datapath that implements this interface.

4.1 FUNi Messaging Interface

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 payload length of message-passing packets can vary from 0 to 21 32-bit words. (Memory-to-memory data transfers can occur in burst sizes varying from 0 to 20 words.) Aside from allowing 2^{14} user-defined packet types, the network interface also supports two hardware-enforced 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.

4.1.1 Interface Registers and Packet Queues

User- and system-level processes control the operation of FUNi by reading and writing to FUNi's internal control registers through memory-mapped accesses. However, the sending and receiving interface is based on a set of software-enforced circular packet queues jointly maintained by the user program and FUNi. Two sets of send and receive queue pairs are provided, one for each packet priority. The queues are logically divided into 32-word packet slots. A set of memory-mapped registers, containing the head index, tail index and base address, is associated with each circular queue. Figure 3 depicts FUNi's message sending and receiving interface in a user's virtual memory space. There is an additional register that specifies the size of the circular queues.

For each queue, the user software controls one end of the queue, and FUNi controls the other end. 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. The two parties rely on the memory-mapped head and tail index 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 another register to pass the tail index to the producer so the producer knows which slots are freed. The circular queues rely on the standard convention of head and tail indices. The head index points to the next free slot for enqueuing a new packet. The tail index points to 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.

Before interfacing with FUNi, a user program first needs to allocate memory space for sending and receiving queues and then initialize FUNi with the starting addresses and the size of the queues. After initialization, it is possible for the software to switch to a new or larger packet queue by updating the appropriate control registers. Once initialized, FUNi will be able to deliver inbound network packets to the receive queues and retrieve outbound network packets from the send queues for transmission.

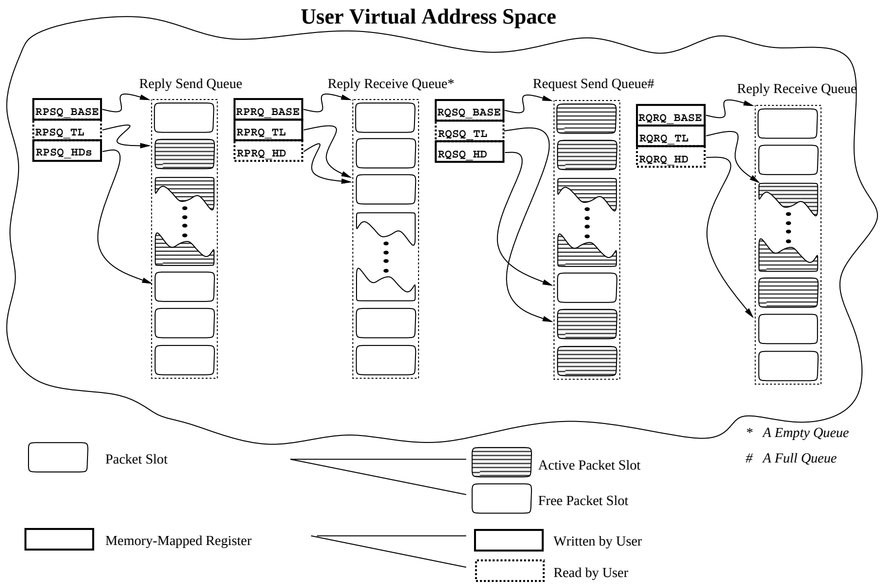

Figure 3: FUNi Message Interface

4.1.2 Receiving Operations

When a packet arrives from the network, the network interface will use DVMA to append the packet to the one of the two receive queues according to the packet's priority. To enqueue a packet, the network interface first compares the content of the tail and the head index registers to prevent overflow. Then the content of packet is stored sequentially into the tail slot of the queue. Once the packet is completely stored, the network interface will increment the head index register. The network interface is allowed to perform DVMA upon packet arrival until the queue for that particular packet type becomes full. At that point the network interface must reject further incoming packets of that type.

The user process can detect the presence of an unreceived incoming packet by comparing the head and tail indices of the receive queue. When an unreceived packet is found, the user program first extracts the packet length from the packet header located at the first word of the packet slot pointed to by the tail index. Then the correct number of words from the packet can be read from the successive addresses following the header word. After the packet's content is received, the program needs to release the slots occupied by the packet by incrementing the content of the tail index register.

4.1.3 Sending Operations

For sending packets, FUNi and the user process exchange roles as producer and consumer. FUNi uses the contents of the head and tail index registers to determine and locate pending outbound packets in the user send queues. FUNi will attempt to retrieve and transmit the pending packets from the user send queues whenever possible. Thus to send a packet, the program composes the packet in the next available slot at the head of the appropriate packet queue. A packet header, containing the logical destination address, packet length, packet type, is written to the first word of that packet slot. The packet payload is stored to the successive addresses following the packet header. After composing, the user process increments the content of the head index register to make the current packet slot visible to FUNi for transmission.

4.1.4 Message-Passing plus Memory-to-Memory

With a passive network interface, performing a block memory transfer requires the sending process to explicitly copy, in verbatim, each byte of transfer from the source to the interface. Similarly, the receiving process must later explicitly copy, in verbatim, each byte from the interface to the destination location. The active FUNi device is extended with a remote DMA transfer feature to eliminate the data movement overhead on both the sending and the receiving nodes. The user program only needs to enqueue a header that specifies the location and length (2 to 20 words) of the transfer block. FUNi will compose the transfer packet directly from its source. Similarly, FUNi can use DMA to write the data from inbound transfer packets directly to their destination location. This DMA-style remote block transfer will significantly reduce the overhead cost of bulk data transfer.

4.2 FUNi Datapath

The FUNi datapath can be divided into seven principal blocks: SBus Interface, FUNi Core Module, Route Table RAM's, Undelivered Packet Cache, Synchronization FIFO Group, Router Interface, and Differential Transceivers. Figure 4 diagrams the high-level datapath of FUNi.

4.2.1 SBus Interface

The FUNi SBus card will be both a master and a slave device on the SBus. The FUNi card behaves as a slave device in response to memory-mapped accesses from the CPU. FUNi assumes the role of a master device to perform DVMA to access the user's packet queues. An off-the-shelf L64853A SBus DVMA Controller [11] from LSI Logic will provide both interfaces on the FUNi card.

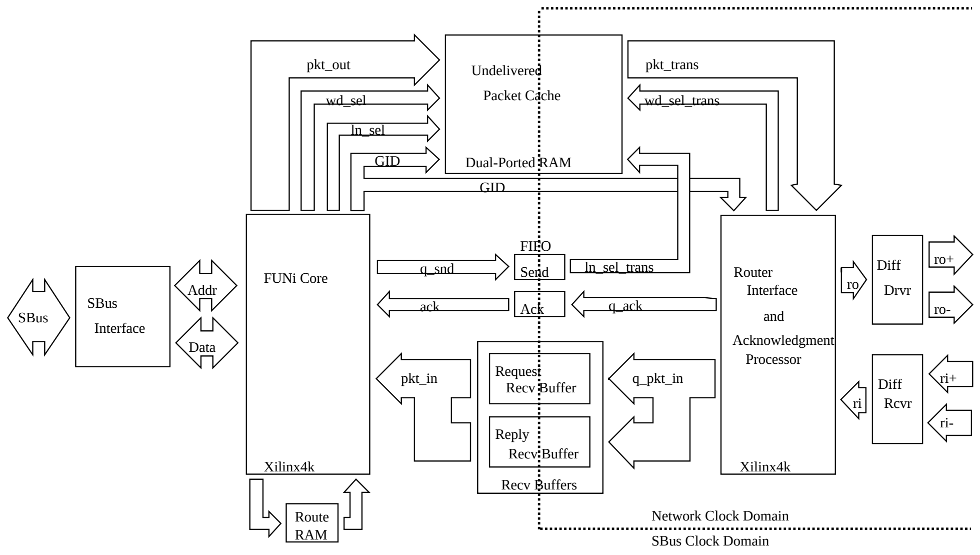

Figure 4: FUNi SBus Card Datapath

4.2.2 FUNi Core Module

The FUNi Core Module contains three finite-state-machine controllers that coordinate packet movements through FUNi. This module will be partitioned, according to the subunit boundary, for implementation as a 3-FPGA chip set. The Retrieval Unit is responsible for retrieving the pending outbound packets from the user's two send queues in memory into the Undelivered Packet Cache, and the Dispatch Unit is responsible for scheduling sends and retries of the outbound packets. The Dispatch Unit is also responsible for processing acknowledgments and maintaining the bookkeeping information for the Acknowledgment/Retry Flow Control Protocol. The Delivery Unit is responsible for delivering inbound packets to the user's receive queues in memory. The FUNi Core Module also contains sixteen 16-bit memory-mapped FUNi status and control registers. These registers are used to specify information about the software packet queues and control the behavior of FUNi.

4.2.3 Undelivered Packet Cache

The multi-context Undelivered Packet Cache, implemented with dual-ported RAM, buffers 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 this cache. The usage of the cache is managed by the Dispatch Unit. The cache is multi-contexted so cached packets do not need to be flushed between context switches.

4.2.4 Route Table RAM's

In the FUNi programming model, remote workstations are named by abstracted integral node ID's. Before a packet is transmitted, the integral destination ID in the packet header needs to be converted into the corresponding route bits that the network routers understand. The content of the route table is loadable by the operating system. This gives the operating system the ability to individually determine, for each workstation, who and where its peer workstations are. Thus, it is possible to partition a cluster into non-interfering sub-clusters for more flexible usage. Individual node can also be remapped or excluded for fault tolerance or load balancing.

4.2.5 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 clocking 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 speeds to connect to the same network.

The Send FIFO and Acknowledgment FIFO are two small FIFO's of 4 and 5 bits wide respectively. The depth of the FIFO's is bounded by the number of Undelivered Packet Cache lines available for each context. The Dispatch Unit schedules a packet for transmission by enqueuing a request into the Send FIFO. The Router Interface Module forwards returned acknowledgments to the Dispatch Unit through the five-bit wide Acknowledgment FIFO.

When the Router Interface Module receives a data packet, the packet is first enqueued into one of the two hardware receive buffers according to the packet's priority. Two buffers are required to buffer reply and request packets separately because higher priority reply packets must not be blocked by request packets. The depth of the buffers is not important since the user receive queues in memory provide 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 hardware buffers do need to have some buffering capacity to handle a momentary pile up of inbound packets. The Router Interface Module will reject subsequent inbound packets when these buffers are full.

4.2.6 Router Interface and Differential Transceivers

The Router Interface Module has three tasks. First, the Transmitter and Input Port Buffers of the interface implement the necessary handshake with Arctic to transmit and receive packets on the network. Second, the Network Packet Preprocessor in the interface participates in the Acknowledgment/Retry Flow Control Protocol by deciding whether a network packet can be accepted and returning the appropriate acknowledgment packet. Lastly, the Transmit Scheduler coordinates the sharing of the transmitter between the data traffic from the Undelivered Packet Cache and the acknowledgment traffic from the Network Packet Preprocessor.

5 FUNet Cluster Performance Evaluation

This section 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.

5.1 The FUNi Simulator

The simulator is based on the PROTEUS [4] simulation engine that allows rapid development of event-driven simulators of parallel architectures. The PROTEUS simulation engine is a collection of C source files for an abstracted core system of a simulator. The FUNet simulator is created by incorporating a custom simulation of FUNi and FUNet into the PROTEUS simulation engine.

5.1.1 Processing Nodes: 40 MHz SPARCstation2

The cycle-counting data in our version of the PROTEUS engine is derived from the SPARCstation2. For our simulation, we assign 40 MHz as the clock rate of our simulated processing nodes. PROTEUS uses an optimistic model for the instruction execution on a single-issue SPARC microprocessor that is fully pipelined. 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 can be occupied by another instruction. PROTEUS 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, details about memory-mapped I/O and bus transactions are incorporated into our simulation. A memory-mapped read latency is approximately 28 CPU cycles (derived from 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. Loads and stores to the user's send and receive queues are treated as cache misses and their latency and effect on bus contention are accounted for.

5.1.2 Programming Environment

User applications are coded in a superset of the C programming language. The FUNet cluster maintains a MIMD message-passing programming model. When the simulator starts at time zero, a process is created on 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. In our effort to assess the effectiveness of the FUNi design, we ignore the effect of time sharing. The benchmark programs are executed alone without interference from other applications.

5.1.3 Physical Network: Hypercube Arctic Hub

The FUNet cluster simulator incorporates a custom network simulation for a hypercube direct-routing network based on Arctic. The operation of the Arctic network is accurately depicted in the simulation. The network is simulated at the estimated network clock rate of 25 MHz. An hypothetical 8-by-8 Arctic router is simulated with three buffers at each input section, with one reserved for high priority packets. The flow-through latency of the simulated Arctic is six 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.

5.2 FUNi

FUNi hardware events are accurately accounted for in terms of latency and resource utilization. The simulator supports the full programming interface defined in [10]. User processes access FUNi's internal control registers through simulated memory-mapped reads and writes. 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 cycles to acquire the bus), and a transfer bandwidth of one 32-bit word per two cycles is used in the simulation.

6 Benchmark and Analysis

Two benchmark programs based on University of California at Berkeley's version of the Connection Machine Active Message (CMAM) communication library [16] were executed on a 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 directly. 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 that 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. A description of the benchmarks is presented below, followed by the results of the experiments.

6.1 CMAM Primitives Benchmark

The first benchmark is used to quantify the performance of FUNi. Instead of measuring idealistic raw throughput

Active Message Passing Primitives										
	send tp		send ovhd		recv tp		recv ovhd		round-trip	
	usec	cyc	usec	cyc	usec	cyc	usec	cyc	usec	cyc
FUNiAM_4	5.5	219.4	2.8	113.2	8.7	348.2	5.5	221.3	29.0	1160.8
FUNiAM_reply_4	5.4	215.8	2.5	98.2	8.4	336.1	5.0	201.3		
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		
FUNiAM_xfer_4	2.9	218.5	5.1	126.6	1.8	365.0	2.7	235.6		
FUNiAM_reply_xfer_4	3.0	215.8	5.7	111.7	1.9	345.2	3.0	215.6		
FUNiAM_mfer_n	10.1	63.6	25.7	24.9	7.5	85.5	14.4	44.3		
FUNiAM_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		
FUNiAM_*	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		
FUNiAM_get_*	13.8	553.7	30.1	1205.0	20.0	797.8	36.6	1457.6		
FUNiAM_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						

Table 1: Performance Comparison between CMAM and FUNet Active Message Library Primitives

by sending and receiving meaningless messages, we measure the performance of FUNi when coupled with the CMAM library. The benchmark suite that is included in the CMAM library distribution has been adapted for FUNet's ported version of the CMAM library. The adapted CMAM primitive benchmark suite is executed in a 32-node FUNet cluster simulation. For reference, a similar suite is also executed on 32 nodes of CM-5. A subset of the result is shown in Table 1.

The first section of the table presents the results of the active message primitives in the CMAM library. For each primitive, five parameters are measured. The send and receive throughput time measures the total time required for a node to send or receive an active message to other nodes. The send and receive overhead measures the execution time of the corresponding primitives on the CPU. The final column of this section shows the round-trip time of a request-priority active message and a returning reply-priority message. The next section of the table presents the results from the data transfer primitives. The last section measures the performance of a high-level communication library that implements a shared-memory coherence protocol in software.

In general, the results reveal that, as expected, FUNi performs much worse in terms of bandwidth and latency when compared to CM-5. However, FUNi's communication overhead in processor cycles is within a factor of two to four of CM-5. *(Note: FUNi's version of the CMAM primitive is coded in C and compiled by GCC with optimization off due to peculiarity of the PROTEUS simulator. Overhead can be further reduced by hand crafting the primitives.)* Furthermore, when coupled with the larger packet size and the DMA feature of FUNi, the extended CMAM primitives perform competitively with their CM-5 counterpart in all respects.

6.2 Matrix Multiply

This particular version of matrix-multiply is taken from von Eiken et al. [16]. The example is well suited for a FUNet cluster because the algorithm pipelines each remote fetch with computations on previously fetched data. The overlapping of communication delay with useful computations hides the effect of FUNi's relatively high communication latency. 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 its relatively low bandwidth and high latency. The second experiment demonstrates the scalability of the FUNet system. For each experiment, three runs are made. One run is made on a CM-5 using UC Berkeley's version of CMAM library. Next, a run is made on the FUNet cluster using an identical version of matrix multiply as the one used for CM-5. Finally, another run is made on the FUNet cluster, this time allowing the use of the FUNet extensions to the CMAM library for improved performance. By comparison, we are able to demonstrate that 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 successful contemporary MPP system.

6.2.1 Latency Hiding and Overhead Amortization

This experiment was performed by von Eiken, et al. for CMAM on a 128-node CM-5 [16]. The experiment is scaled down for execution on a 32-node FUNet simulator. In the different trials, the dimensions of the matrices are varied to control the ratio of computation versus communication while maintaining the total number of floating point operations. Figure 5 plots the results of the experiment. The Y-axis represents the percentage of CPU utilization in each run, and the X-axis marks increased ratio of computation versus

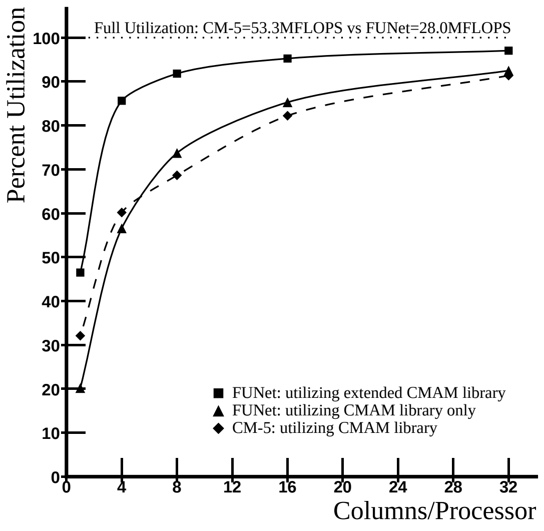

Figure 5: Utilization vs. Columns per Processor for Matrix-Multiply

communication. Comparing the curves, we see that FUNet exhibits normalized behavior similar to CM-5. In all cases, processor utilization quickly approaches optimal.

6.2.2 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. Figures 6, 7, and 8 plot the result from multiplying two square-matrices of 64-by-64, 128-by-128 and 256-by-256, respectively. In Figure 6 for square matrix multiply of dimension 64-by-64, all three curves break from linear speedup because the problem size is simply too small for the computation to amortize the communication overhead on larger systems. In Figure 7 and Figure 8 for larger square matrix multiplies, we begin to see an improvement in the linearity of speedup in all cases as problem size is increased.

7 Related Work

PVM [3] and Linda [8] are examples of software systems that enable parallel processing on a cluster of workstations using existing LAN facilities. Interworkstation communication is accomplished through interface routines implemented over existing Unix interprocessor communication and networking facilities. These systems have the advantages of not requiring additional hardware. However, the scalability and granularity of parallel processing are restricted by the overhead of communication.

The high communication overhead has been attributed to the poor implementations of transport and network protocols [6]. Afterburner [5] combines hardware and software techniques to improve the performance of TCP/IP communications. The Afterburner project investigated several software techniques for reducing redundant data movement in the TCP/IP protocol and implemented an accompanying network-independent network interface card for HP series 700 workstations. For packets in the KByte range, they have shown significant improvement over traditional imple-

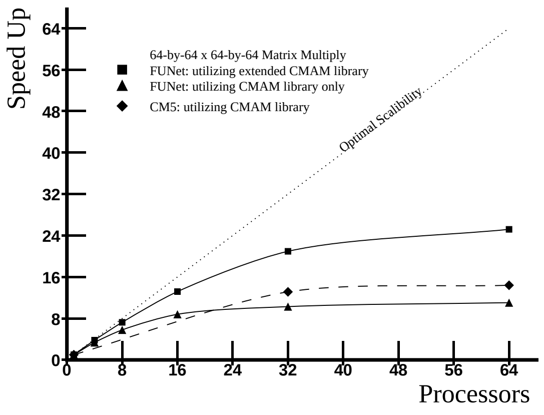

Figure 6: Processor Scalability: Multiplying 64-by-64 Matrices

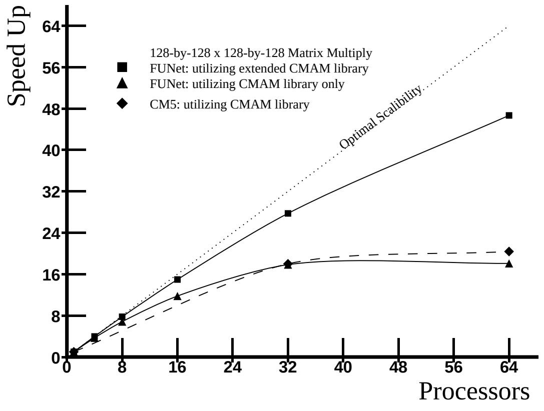

Figure 7: Processor Scalability: Multiplying 128-by-128 Matrices

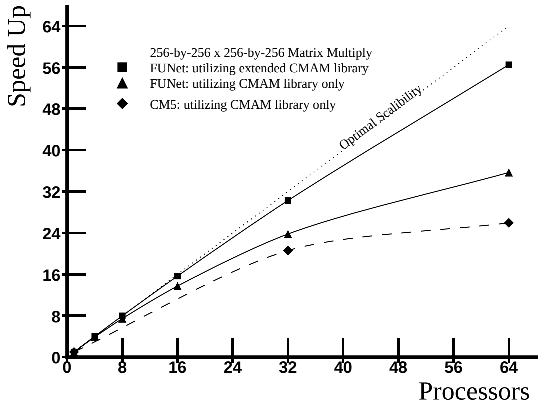

Figure 8: Processor Scalability: Multiplying 256-by-256 Matrices

mentations.

Nevertheless, these communication facilities involving operating system calls and heavyweight protocols fail to address the needs of fine-grain parallel processing. The interprocessor communication in fine-grain parallel processing occurs in frequent and small-size messages. The communication overhead must be further minimized by giving the user processes direct control of the network interface. These low-overhead user-level network interface designs can be found in many contemporary MPP architectures [9, 14]. However, these designs typically involve the support of custom system or CPU design.

In most contemporary workstation designs, the RISC microprocessors are optimized for cached accesses while the bus architectures are optimized for blocked transfers. The network interface design must take these constraints into account and use the available features to its advantage. FUNi uses the DVMA feature of the SBus to replace the costly memory-mapped accesses. The SHRIMP multicomputer project [12] specifies another user-level network interface that involves the memory system to reduce the communication overhead in a constrained environment. The SHRIMP network interface is designed for a network of Pentium PC's with Xpress Bus and EISA bus. Communication between any two PC's is accomplished by mapping the virtual memory of one PC into the other. The network interface on the source PC snoops the bus for writes into the mapped area of memory and automatically formats an outbound packet to the target PC associated with that memory location. The target network interface delivers the message to the corresponding mapped area of memory with DMA.

8 Conclusion

In an attempt to design a network interface that would retrofit commercial workstation hardware, our design space was limited. By relying on split-phased transactions to tolerate network latency, we focused on minimizing communication overhead in the FUNi design, even at the cost of increased latency. The result is a network interface based on software circular queues and an active network interface device with the ability to directly access these queues in the virtual memory. Based on the preliminary simulation, by keeping the overhead low, a FUNi-equipped FUNet cluster is able to successfully execute a relatively fine-grained parallel program with good performance and scalability, despite the moderately long communication latency.

We believe we have produced a satisfactory design that supports efficient and scalable fine-grained message-passing on a cluster of workstations and other platforms where long network access latency needs to be tolerated. However, 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. Future generations of microprocessors and workstations need to facilitate parallel processing by incorporating a tightly coupled network interface as an integral part of their design.

9 Acknowledgments

This research was supervised by Professor Greg M. Papadopoulos and was supported in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and by an Office of Naval Research Graduate Fellowship.

References

[1] A. Boughton, et al. *Arctic User's Manual.* Computation Structures Group, Laboratory of Computer Science, Massachusetts Institute of Technology, 1993.

[2] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Transaction on Programming Languages and Systems*, 11(4):598–632, October 1989.

[3] A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. A user's guide to PVM parallel virtual machine. Technical Report TM-11826, Oak Ridge National Laboratory, July 1991.

[4] E. A. Brewer. Aspects of a parallel-architecture simulator. Technical Report MIT/LCS/TR-527, Laboratory of Computer Science, Massachusetts Institute of Technology, January 1992.

[5] Chris Dalton, el al. Afterburner. *IEEE Network*, July 1993.

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

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

[8] D. Gelernter. Parallel programming in Linda. In *Proceedings of International Conference on Parallel Processing*, August 1985.

[9] D. S. Henry and C. F. Joerg. A tightly-coupled processor-network interface. In *Proceedings of ASPLOS*, October 1992.

[10] J. C. Hoe. Effective parallel computation on workstation cluster with user-level communication network. Master's thesis, Massachusetts Institute of Technology, February 1994.

[11] LSI Logic. *L64853A SBus DMA Controller Technical Manual*, 1991.

[12] M. A. Blumrich, et al. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of 21st ISCA*, April 1994.

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

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

[15] D. E. Thomas and P. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[16] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of 19th ISCA*, May 1992.

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