

Micro-architectures of High Performance, Multi-user System Area Network Interface Cards

Boon Seong Ang
Hewlett-Packard Laboratories
1501, Page Mill Road, Palo Alto, California
boon_ang@hpl.hp.com

Derek Chiou, Larry Rudolph and Arvind
Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square, Cambridge, Massachusetts
{derek,rudolph,arvind}@lcs.mit.edu

Abstract

This paper examines two Network Interface Card micro-architectures that support low latency, high bandwidth user-level message passing in multi-user environments. The two are at different ends of a design spectrum – the Resident queues design relies completely on hardware, while the Non-resident queues design is heavily firmware driven. Through actual implementation of these designs and simulation-based micro-benchmark studies, we identify issues critical to the performance and functionality of the firmware-based approach. The firmware-based approach offers much flexibility at a moderate performance penalty, while the Resident design has superior performance for the functions it implements. This leads us to conclude that a hybrid design combining complete hardware support for common operations and a firmware implementation of less common functions achieves both high performance and flexibility.

1 Introduction

Message passing service in tightly coupled System Area Networks must guarantee full protection within a multi-user shared environment while achieving low latency and high bandwidth. This is becoming increasingly important as System Area Networks are incorporated into new I/O standards, such as Infiniband [11]. A general approach is to have each node in the network support multiple private message queues allocating a unique set to each job that requires user-level, high performance communication [9, 8, 10, 3].

In order to identify the best Network Interface Card (NIC) architecture that can achieve low cost while meeting the functional requirements and performance targets, we studied two approaches at different ends of the design spectrum. The *Resident queues* design relies on hardware

alone to support multiple message queues. When designed properly, this should achieve the highest performance, but is expected to be costly and limited to a small set of hardware functions. The *Non-resident queues design* combines a generic commercial off-the-shelf (COTS) microprocessor executing firmware with a set of basic hardware message passing capabilities resulting in a flexible design that is easily modified to present different communication functions to application code. Though it is often a cheaper solution since it employs off-the-shelf high volume parts, it has lower expected performance.

Although the qualitative trade-off is easy to surmise, the actual quantitative comparison is less obvious. How much worse is the performance of the firmware based design? What limits its performance? What kind of flexibility can the firmware based design achieve? Is the design complexity and cost of one design clearly less than those of the other? How should the embedded processor interface with the host, and with its message passing hardware? This paper sheds light on these questions by studying two specific designs, both of which are implemented in the StarT-Voyager research machine [5, 4].

This paper makes several contributions. First, the micro-architecture design embeds a microprocessor into a NIC in a novel way that is as functionally capable as any custom hardware design by employing a *slave and snooping interface* between the host system and the embedded microprocessor and a *command and completion queues interface* between the embedded microprocessor and the rest of the NIC. Second, simulation studies of actual implemented designs show that the Non-resident queues approach offers reasonably competitive performance, with degradation limited in rare cases to a factor of five, but typically much less. Two main factors limiting the firmware-based design's performance are identified: (i) the overhead of multi-threading necessary to multiplex many functions on the embedded processor, and (ii) limited firmware off-chip access band-

width and latency. While firmware provides flexibility in modifying NIC functions, this flexibility is tempered by performance considerations – complex functionality may take too many firmware instructions. Finally, the experience has convinced us that an aggressive firmware-based design may be as complicated as a full hardware based design. We believe that the best design is a hybrid combining both designs, with full hardware handling of common operations, and firmware providing rarely invoked but important functions. This is the design philosophy behind StarT-Voyager; interested readers are referred to Ang [2].

The next section describes the message passing mechanisms employed in this study, and the NIC’s point of attachment to the host. Section 3 describes the micro-architecture of our all-hardware NIC, as a contrast to the firmware based design of Section 4. The heart of this paper is in Sections 4 and 5 that describe how we embed a microprocessor into a NIC and presents and interprets our simulation results respectively.

2 Background

The work presented in this paper is based on the Network Interface Card (NIC) design that was implemented as part of the StarT-Voyager. The NIC is a hybrid containing a superset of both designs discussed in this paper. The NIC connects to the 60X bus [1], of an IBM-manufactured, AIX machine with PowerPC 604e processors, taking up one of two processor card slots in the host system (Figure 1). In other words, the NIC connects directly to the cache-coherent memory bus of the host system. NIC’s are connected through the Arctic Network [6, 7], a high performance, packet switched, Fat-Tree network. Two links, an in-link and an out-link, each of which delivers a bandwidth of 150 MBytes per second, connect to each NIC.

This study employs two common message passing mechanisms: *basic messages*, suitable for small to medium size communication, and *inter-node DMA* tailored for large transfers (on the order of kBytes). This set of message passing mechanisms is chosen for its simplicity; comparisons with more sophisticated alternatives is beyond the scope of this paper. We will, however, discuss the impact of the message passing interface design on the efficiency of firmware implementation in Section 5.2. Readers who are interested in the intricacies of message passing interface may wish to refer to Mukherjee et. al. [12], and Ang et. al. [3, 2].

Basic Message: A basic message consists of a 32-bit header and up to 88 bytes of payload. The header specifies a virtual destination queue. After a message is inserted into a transmit queue, the associated queue pointer is updated signalling the fact that the message can be launched into the

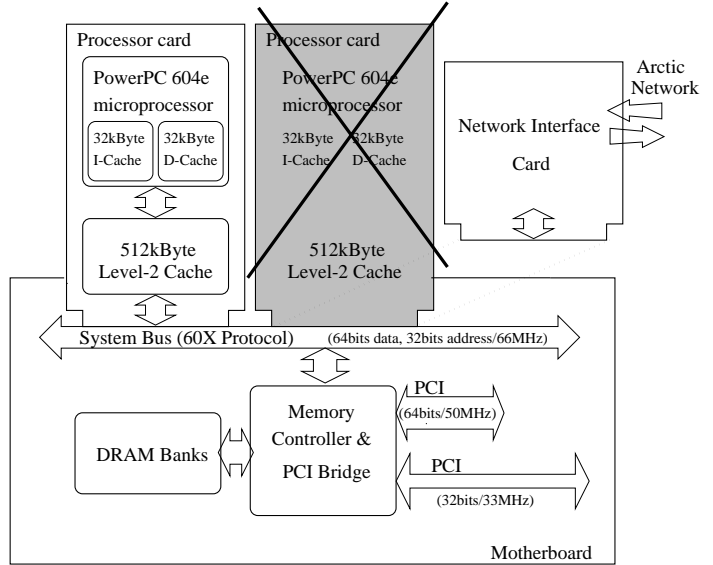


Figure 1. Our experiment system replaces one of the processor card of an SMP with a NIC.

network. Message reception works in an analogous fashion. The individual transmit and receive queues are implemented as circular buffers that can be part of user-level address space. Although the buffer space may be cachable, the pointers must be uncached. The message *content is specified by value, i.e.* the processor is responsible for assembling the message content from various parts of memory into the transmit buffer space. A pointer update immediately triggers NIC processing.

Inter-node DMA: The Inter-node DMA message *content is specified by reference*, by providing virtual source and destination node and memory addresses. The NIC is responsible for address translation and moving the data from source to destination, including marshalling the data into packets at the source, and writing it into the specified locations at the destination.

Queue Translation: Communication traffic through the message queues are multiplexed onto a shared network fabric. The NIC implements *message destination translation* as a protection mechanism, similar in concept to virtual memory address translation.

3 Resident Queues NIC

NIC processing is *event driven*: on the host side, it is triggered by a bus operation and on the network side, it

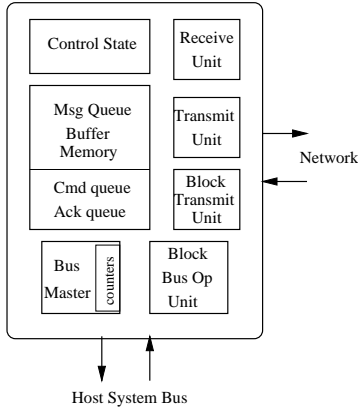


Figure 2. The Resident Queues NIC.

is triggered by a packet arrival or link-level flow-control event (Figure 2). In the *Resident Queues* design, message queues are allocated in fast SRAM on the NIC itself. The state associated with each queue is separated into two parts: (i) buffer space, implemented with dual-ported synchronous SRAMs and (ii) control state, embedded with the transmit and receive functional units. The exact location and size of each queue’s buffer space is programmable by setting appropriate base and bound registers in the queue’s control state. To support a number of queues with minimal duplication of hardware, the message queue control state is aggregated into “files”, similar to register files. Control logic that choreographs the launch and arrival of messages to and from the network are structured as functional units shared between the queues.

The user code accesses the message queue buffer space directly. Software can access message queue control state through two windows. One window, for OS use, has access to the full state including configuration information and a second window, for user-level code, provides limited access to only producer and consumer pointers.

Inter-node DMA is implemented with a *Block Bus Operations unit* and a *Block Transmit unit* at the sender NIC, and a *Remote Command Queue* and a *counting service* in the *Bus Master unit* at the receiver NIC. Each DMA packet includes both data and bus commands. Execution of bus commands by the Bus Master unit at the destination NIC writes data to appropriate host main memory locations. The counting service tracks the number of packets that have arrived; when all packets of a transfer have arrived, an acknowledgement is inserted into the *Acknowledgement queue*. The host processor sets up the Block Bus Operations unit, the Block Transmit unit, and initializes the DMA Channel Counters with commands issued to the *Local Command queues*. All these commands include acknowledgement options.

Splitting the DMA support into modular functional units

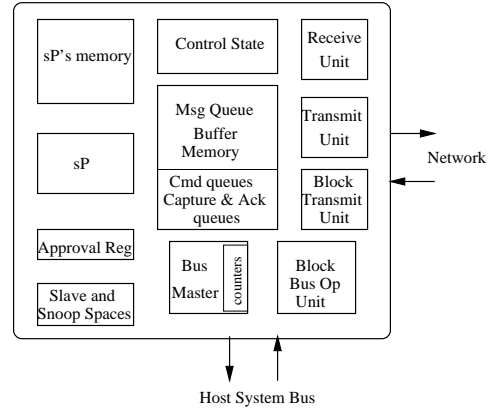


Figure 3. The Non-resident Queues NIC.

improves flexibility. For instance, multi-cast DMA only needs to invoke Block Bus Operations unit once, followed by multiple invocations of the Block Transmit unit. (Operation chaining between these two units ensures full pipelining between them, thus avoiding any latency penalty in the uni-cast case.) Similarly, the receiver can set up the counting service such that it is only informed when data from multiple source nodes have arrived.

The actual implementation modelled in this study uses dual-ported SRAMs, Xilinx 4k family FPGA’s, and Chip-Express CX2001 LPGA (Laser Programmable Gate Array). Except for minor modifications, this is a subset of the StarT-Voyager NES, which has been implemented. Readers interested in details of the StarT-Voyager NES implementation are referred to Ang [2].

4 Non-resident Queues NIC

Two major NIC components implement non-resident queues: (i) a Service Processor (sP) and (ii) custom NIC functional units which we refer to as NIC Core. Since the host employs PowerPC 604e processors, the PowerPC 604 processor was chosen as the sP so that the design deals with only one bus protocol. The custom NIC functional units mediate between the sP, the host memory bus, and the connection to the network (Figures 3). The message queues reside in host main memory, with the NIC providing only transient buffer space.

4.1 Interface between sP and the Host System

With the help of custom functional units, the sP can directly participate in application processor (aP) bus transactions through two special address spaces: the *sP Serviced Space* and the *Snooped Space*. The sP Serviced Space is a physical address region on the system bus that is handled by

the sP: sP determines the data for bus reads, the destination for bus writes, and when each bus transaction is allowed to complete. This address space is thus active. The sP can also initiate bus operation on the host system bus through local command queue requests, thereby permitting such operations as accessing host main memory.

A *transaction capture* mechanism informs the sP about a bus transaction by inserting the address and control information as a 64-bit entry into a *Capture & Ack* queue, which is polled by the sP. Completion of a captured bus transaction can be “suspended” through an *Approval Register* mechanism, which is also the means for the sP to provide the data for bus-read actions.

The Snooped space is similar to the sP Serviced space, except it is a portion of the host main memory for which the NIC Core provides cache-line granularity access-permission check, similar to the permission check in Typhoon [14, 15, 17] or S-COMA[16]. This capability enables the sP to implement fancier message passing interfaces, such as CNI [12], that can only be implemented efficiently if cache-line ownership acquisition is used to trigger processing.

4.2 Interface between sP and NIC Custom Functional Units

The interface between the sP and the NIC Core is critical to performance, with sP occupancy and overall latency of firmware implemented functions varying significantly between designs. In place of traditional status and command register interface, we adopt a *command and completion queues* interface.

I/O devices are commonly designed with status and command registers memory mapped by the microprocessor: the microprocessor writes command registers to issue requests and reads status registers to check results and progress, or it responds to an interrupt raised by the device. Such an interface is unsuitable for our design because: (i) each command register typically supports only one request at a time; (ii) dependencies between operations to different registers cannot be enforced; (iii) polling individual status registers to determine progress is slow; and (iv) interrupts are expensive.

Our design avoids these deficiencies by using an interface composed of (i) two command queues, (ii) a completion queue and (iii) a *OnePoll* mechanism that provides the ability to simultaneously poll from several queues.

Most commands issued by the sP are not single cycle and often must be issued in a specified order. Rather than spending most of its time in send-acknowledge protocols the sP issues, all at once, an entire sequence of commands needed for processing a new request. It does this by sending messages to the command queues. Each command has

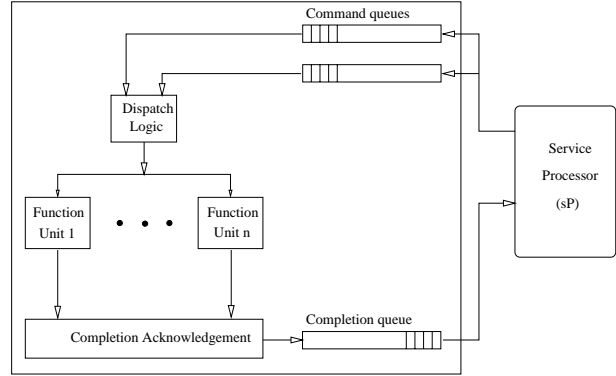


Figure 4. A command and completion queues interface.

the option of completion notification, making it possible for only the last command to issue a notification of completion by sending a message to the completion queue. Since it is expensive for the sP to poll the Core the completion queue is treated like another message queue, and can be one of the queues named in a *OnePoll*. This design also facilitates pipelining: not only can the sP issue more commands while earlier ones are being executed, the NIC Core can be designed to exploit parallelism between these commands. Our choice is a compromise between design simplicity and parallelism exploitation (see Ang [2] for discussion of these choices).

4.3 Firmware Message Passing Implementation

Firmware implements Basic messages with queue buffers mapped to the host main memory, queue status state mapped to NIC SRAM and queue control mapped to the sP Serviced Space. Mapping control addresses to sP Serviced Space enables event driven processing, with captured transactions triggering sP processing.

A small number of hardware queues in the Core are used as proxy transmit and receive queues. Firmware multiplexes and demultiplexes the messages in user-level private queues between these shared queues. For out-going traffic, firmware also implements destination address translation to impose access control. Much of the work done by sP firmware is resource allocation and deallocation, and coordination of very low-level hardware operation.

5 Simulation Studies

This section describes simulation evaluation results obtained with micro-benchmarks. The results fall into two categories. The first category compares the latency and band-

width of the two designs when handling individual messages. The second category evaluates the performance of the off-the-shelf sP, providing both absolute performance numbers for representative communication operations and an understanding of the factors limiting its performance.

5.1 Simulation Environment

StarT-sim is an execution driven simulator that allows run-time determined memory access or message passing latency times to influence execution path of any non-deterministic programs. Written in C, StarT-sim runs in a single process and has two major components: (i) a processor core simulator (AugRS6k), and (ii) a memory system, NIC and network simulator (Csim). AugRS6k¹, operates in an event driven fashion and is capable of modelling a multi-processor system. As long as a simulated processor is executing instructions that do not depend on external events, the simulator advances its “local” time and continues executing more instructions. Simulation of a processor only suspends at a memory access instruction and resumes when global time reaches its local time.

Processor simulation is achieved by editing the assembly code of the simulated program with an extra custom compilation pass. Memory access instructions are replaced with calls to special routines which enforce causal order by suspending and resuming simulation as appropriate. These routines also make calls to model caches, the NIC, and the network. Code is also added at basic block exit points to advance local processor time.

Csim is a detailed RTL (Register Transfer Level) style model of the processor cache, memory bus, NIC’s, and Arctic network and is written in C. Simulation in Csim proceeds in a cycle-by-cycle, time synchronous fashion. Modelling is very detailed, accounting for contentions and delays in the NIC’s and the memory system. The NIC portion of this model is derived from the Verilog RTL description used to synthesize the custom hardware in StarT-Voyager NES.

StarT-sim models two separate clock domains. The processor core, load/store unit, and cache are in the processor clock domain while the memory bus, the NES, and the network operate in the bus clock domain. The performance numbers reported assume a processor clock frequency of 140 MHz, and a bus clock frequency of 35 MHz to match those of our hardware proto-type.

StarT-sim also models address translations, including both page and block address translation mechanisms found in the PowerPC architecture. Our simulation environment provides a skeletal set of virtual memory management sys-

¹The processor simulator was based on the Augmint[13] simulator which models x86 processors. Initial effort to port it to model the PowerPC architecture was done at IBM Research. We made extensive modifications to complete the port.

TxQ Resident?	RxQ Resident?	Bandwidth (MByte/s)
Yes	Yes	53.15
Yes	No	25.75
No	Yes	17.80
No	No	17.40

Table 1. Bandwidth achieved with Basic Message. The four cases employ different combinations of Resident and Non-resident queues at the sender and receiver.

tem software routines to allocate virtual and physical address ranges, track virtual to physical address mappings, and handle page faults.

5.2 Latency and Bandwidth Comparison

We present bandwidth and latency numbers for the four combinations resulting from the cross-product of the sender and the receiver using either Resident or Non-resident queues. Table 1 presents the Basic message bandwidth numbers. The highest bandwidth is achieved with Resident queues and drops by a third when both sender and receiver use Non-resident message queues. The table also shows that the bottleneck is at the sender’s end, since close to half the Resident queues bandwidth is attained if the sender uses a Resident queue while the receiver uses a Non-resident queue.

Figure 5 reports one-way message latency for various payload sizes. Non-resident queues incur longer latencies, with smaller messages suffering the worst deterioration because several overhead components are fixed regardless of message size. In the worst case of a one word (4 Byte) Basic message sent between Non-resident queues, latency is almost five times longer than that between Resident queues. The figure also shows that Non-resident message transmission causes greater latency deterioration than Non-resident message receive.

The Non-resident Queues performance, although not as good as anticipated, is still respectable. The one-way latency of 12 μ s is not unlike those reported for Myrinet. Myrinet performance varies depending on the host system used and the interface firmware running on the Lanai, its custom designed embedded processor. Numbers reported by various researchers indicate one-way latency as low as 5.5 μ s. However, well known user-level interfaces, such as AM-II and VIA incur one-way latency numbers of 10.5 μ s and 30 μ s respectively and achieve bandwidths of 31 MBytes/s and 90 Mbytes/s respectively [18]. Although the Non-resident bandwidth of around 18 MByte/s may look low, this is achieved with the sP performing send and re-

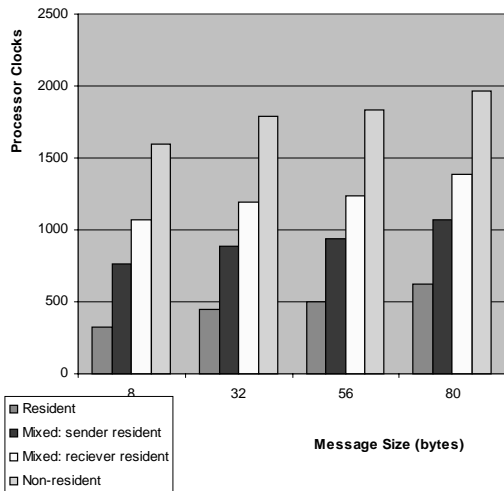


Figure 5. One-way latency of various sized Basic Message, using different combinations of Resident and Non-resident queues.

ceive of messages with only 80 byte payload; larger message sizes are addressed in the next subsection.

An insight gained from this evaluation is the importance of handling the most common communication operations completely in hardware. The relatively long latency of many well known message passing Network Interface Units, such as Myrinet and SP-2, is due in large part to using firmware to handle *every* message send and receive operation. It is common to attribute their long latency to their location on the I/O bus. While that is partly responsible, firmware is by far the biggest culprit.

Another insight is that the low-level designs of the message passing mechanism can affect the efficiency of firmware implementation. In particular, the Basic message design forces multi-phase firmware processing of an operation as full information for complete processing of an operation is not immediately available when processing is triggered. In the case of Basic message transmit, the destination address must be marshalled before further processing can occur. This adds significant overhead from saving and restoring state between the phases.

5.3 Performance Limits of the sP

The NIC Core provides the sP with a large, flexible set of functions, so that the sP is functionally capable of emulating almost any communication abstraction. What is less apparent is the real sP performance and the factors that limit this performance.

Our results show that performance is constrained by either context switching or off-chip access. When the sP implements fine-grained functions, the context switch overhead – event poll, dispatch and book-keeping – dominates, especially for multi-phase operations. On the other hand, when each sP invocation orchestrates a large amount of data communication, the cost of sP off-chip access dominates. Our one-poll mechanism does reduce the number of off-chip accesses, but we regret our lack of hardware support for allocating and deallocating NIC Core hardware resource.

Two sets of experiments comprise this subsection. The first is the Non-resident Basic message performance presented earlier. We revisit the performance results in Section 5.3.1, dissecting it to quantify the cost of generic sP functions. The second set of experiments examines several block transfer implementations. It differ qualitatively from the first set in that a fair amount of communication is involved each time the sP is invoked. Because the sP overhead from dispatch and resource allocation is amortized, these experiments reveal a different set of performance constraints, as reported in Section 5.3.2.

5.3.1 sP Handling of Micro-operations

Simulating Non-resident Basic message benchmarks give the cost for each invocation of the sP when it emulated Basic message queues, see Table 2. These numbers are consistent with the bandwidth numbers in Table 1 which shows that at the bottleneck point, one can calculate² that each Non-resident Basic packet takes 660 processor clock cycles for transmit and 457 processor clock cycles for receive. Together, they also show that sP occupancy constrains the transmit bandwidth, but not the receive bandwidth.

Why does the sP take several hundred processor clocks to process each of these events? First, all our code is written in C and then compiled with GCC. Manual examination of the code suggests that careful assembly coding should improve performance by at least 20-30%. Second, the sP code is written to handle a large number of events. Polling and then dispatching to these events, and saving and restoring the state of suspended, multi-phase processing all contribute to the cost.

As an illustration, consider the timing for sP code fragments taken from the Non-resident Basic message implementation. The sP’s top-level dispatch code using a C switch-case statement requires 13 processor clocks. When

²Each emulated Basic Message packet carries 84 bytes of data. When sP emulates message transmit, it achieves 17.8 MBytes/s, *i.e.* it processes 212 thousand packets every second. Since the sP operates at 140 MHz, this works out to be one packet every 660 cycles. Similarly, when the sP emulates message receive, it achieves 25.75 MBytes/s, *i.e.* it processes 306.5 thousand packets every second. This works out to be one packet every 457 cycles.

Component	sP occupancy (proc clks/packet)
Tx emulation, Phase 1 (Marshal data)	346
Tx emulation, Phase 2 (Destination trans and launch)	281
Total	627
Rx emulation, Phase 1 (Demultiplex data)	209
Rx emulation, Phase 2 (Free buffer and update queue state)	112
sP free receive buffer	59
Total	380

Table 2. Breakdown of sP occupancy when it implements Non-resident Basic message.

Transfer Method	Bandwidth (MBytes/s)
NIC Hardware DMA	84.40
sP sends and receives	62.41
sP sends, NIC hardware receives	70.85

Table 3. (4 kByte) Block transfer bandwidth.

this is followed by dequeuing the state of a suspended operation from a FIFO queue, and then a second dispatch, an additional 35 processor clocks is incurred.

Hardware resource management, such as allocation and deallocation of space in the local command queues, also incurs overhead. With each task taking several tens of cycles, these dispatches, lookups and resource management very quickly add up to a large number of sP processor cycles.

5.3.2 sP Handling of Macro-operations

We measured the bandwidth achieved for 4 kBytes block transfer using three transfer methods, two of which involves the sP, see Table 3. The first method uses the NIC hardware Inter-node DMA capability.

In the second method, the sP is involved at both the sending and receiving ends. The sP packetizes and sends data by issuing aP bus operations to its local command queue to read data into the NIC. These are followed by commands to ship the data across the network. The sP takes advantage of the local command queue’s FIFO guarantee to avoid a second phase processing of the transmit packets. On the receive end, the receive queue is set up so that the data part of the packet goes into one SRAM, while the header part goes into another SRAM. The sP examines only the latter, and then issues aP bus operation commands to its local

Operation	bus transactions	
	num & type	bus clocks
Poll for events	2 short	8
3 local commands	3 short	12
1 cache-line write-miss	1 long	5
1 cache-line flush	1 long	5
Poll for command ack (aggregated)	2 short/2	4
Total		34

Table 4. Off-chip firmware accesses when sP sends a block of data.

command queue to move the data into the appropriate host DRAM locations. Processing at the receiver sP has a second phase to de-allocate receive queue data buffer space. This can be aggregated, and the reported numbers are from code that aggregates the buffer free action of two packet into one sP invocation.

In the third benchmark, the sP is only involved at the sender end. On the receiving side, the packets are processed by the NIC’s remote command queue, the same hardware used in NIC hardware DMA. The purpose of this example is to characterize sP packetize and send performance. When both sending and receiving are done by sP firmware, the receive process is the likely performance bottleneck because it has two phases. The third transfer method achieves higher performance than the second one, Table 3, confirming this suspicion.

The numbers in Table 3 can be explained by the conjecture that the limiting factor on sP performance is the number of times it makes off-chip accesses. For example, under the second transfer method, the 62.41 MBytes/s bandwidth implies that the bottleneck processes one packet every 143.5 processor clock (36 bus clocks). As shown in Table 4, when the sP sends block data, off-chip access occupies the sP bus for 34 bus clocks per packet.

6 Conclusions

Our investigation shows that with innovative design, a firmware driven NIC built with off-the-shelf microprocessor can deliver fairly competitive performance. Nevertheless, such a design is unlikely to deliver the same level of performance as a hardware based design. In particular, latency of a hardware based design is difficult to match.

A pure hardware based design suffers from inflexibility, however. Furthermore, supporting more than a small set of functions quickly increases the hardware design complexity. In contrast, firmware handles greater complexity gracefully, and allows easy modifications.

We believe that the best design approach is a combination of both, with full hardware support for the most frequently invoked functions, backed up by firmware implementation of less common ones. This is the design approach we took in StarT-Voyager [2]. A specific example is implementing a large number of message queues with a *message queue caching* approach – hardware fully implements a small number of message queues, with the rest emulated by firmware. The hardware queues operate as a cache for “hot” queues, with the switch between hardware and firmware emulated queue transparent to host software.

Acknowledgements: This paper describes reserach done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. We would like to acknowledge Daniel L. Rosenband for his excellent ASIC implementations and Mike Ehrlich for his management of the StarT-Voyager NES hardware effort. Funding for this work is provided 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 Ft Huachuca contract DABT63-95-C-0150.

References

- [1] M. S. Allen, M. Alexander, C. Wright, and J. Chang. Designing the PowerPC 60X Bus. *IEEE Micro*, 14(5):42 – 51, Sep/Oct 1994.
- [2] B. S. Ang. *Design and Implementation of a Multi-purpose Cluster System Network Interface Unit*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, Feb. 1999.
- [3] B. S. Ang, D. Chiou, L. Rudolph, and Arvind. Message Passing Support on StarT-Voyager. In *Proceedings of the 5th International Conference on High Performance Computing, Chennai (Madras), India*, Dec. 1998.
- [4] B. S. Ang, D. Chiou, L. Rudolph, and Arvind. StarT-Voyager: A Flexible Platform for Exploring Scalable SMP Issues. In *Proceedings of Supercomputing'98, Orlando, Florida*, pages 228–237, Nov. 1998.
- [5] B. S. Ang, D. Chiou, L. Rudolph, and Arvind. The StarT-Voyager Parallel System. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, Paris, France*, pages 185–194, Oct. 1998.
- [6] G. A. Boughton. Arctic Routing Chip. In *Proceedings of Hot Interconnects II, Stanford, CA*, pages 164 – 173, Aug. 1994.
- [7] G. A. Boughton. Arctic Switch Fabric. In *Proceedings of the 1997 Parallel Computing, Routing, and Communication Workshop, Atlanta, GA*, June 1997.
- [8] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilks. An Implementation of the Hamlyn Sender-managed Interface Architecture. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation, Seattle, WA.*, pages 245–259, Oct. 1996.
- [9] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilks. Hamlyn: a High-performance Network Interface with Sender-based Memory Management. Technical Report HPL-95-86, Computer Systems Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA., July 1995.
- [10] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66 – 76, Mar/Apr 1998.
- [11] D. Garcia, J. Cowan, G. Still, C. Madison, M. Bradley, and K. Potter. Future I/O. In *Proceedings of Ho Interconnects 7, Stanford, CA*, Aug. 1999.
- [12] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 247–258, May 1996.
- [13] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint Multiprocessor Simulation, Toolkit for Intel x86 Architectures. In *Proceedings of 1996 International Conference on Computer Design*, Oct. 1996.
- [14] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture, Chicago, IL*, pages 325 – 336, Apr. 1994.
- [15] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.
- [16] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An Argument for Simple COMA. In *Proceedings of the First International Symposium on High-Performance Computer Architecture, Raleigh, NC*, pages 276–285, Jan. 1995.
- [17] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297 – 307, Oct. 1994.
- [18] T. von Eicken and W. Vogels. Evolution of the Virtual Interface Architecture. *IEEE Computer*, 31(11):61 – 68, Nov. 1998.