Research Feature

Making Network Interfaces Less Peripheral

Most network interfaces behave like disk interfaces and thereby limit the effectiveness of today's high-performance networks in a variety of ways. The authors argue that—to improve performance—future NIs should appear to their hosts more like memory than like disk interfaces.

Shubhendu S. Mukherjee
Compaq Computer Corp.

Mark D. Hill
University of Wisconsin, Madison

A barrier to delivering improvements in network bandwidth and latency to users is the *network interface* (NI), which connects a network to the host computer that runs the network software. An NI includes hardware that exposes an internal interface—such as device registers—to a host processor.

A key problem with most current NIs is that their internal interface is similar to that of a disk's interface. Specifically, these NIs require applications to use an operating system (OS) call, are placed on the I/O bus, do not allow caching of device registers, and force processors to interact with them through in-order and nonspeculative accesses (Figure 1a).

The last problem is subtle and partly caused by hosts that communicate with NIs via memory operations that are overloaded with side effects. A load to an NI device register, for example, both returns a value and deletes it from the device. Because of such limitations, current NIs will not be adequate for use with newer, high-performance networks and host computers.

High-performance local area networks (LANs) such as Myricom's Myrinet or Tandem's Servernet have advanced so far that some view them as a new class of networks called *system area networks* (SANs).[1] Emerging SANs deliver bandwidths of 10 Gbps or more and latencies of tens of nanoseconds—two to four orders of magnitude better than that delivered by most current LANs. Furthermore, the high reliability of SAN hardware enables the use of leaner—and therefore higher performance—network software (communication protocols, such as Active Messages) instead of heavy-weight and one-size-fits-all protocols, such as TCP/IP.

New hosts demand much higher performance than in the past because of faster or multiple processors and because of multimedia extensions. If NIs do not adapt to these changes in networks and host computers, they will become a barrier to improving network performance.

To solve this problem, future NIs should appear to their hosts more like memory than like disk interfaces. Memory is virtualized without generally requiring OS intervention, sits on the memory bus, can be cached, can be accessed out of order and speculatively, and does not have side effects. We propose to treat NIs the same way, as summarized in Table 1. Traditional NIs that use *direct memory access* (DMA) offer some of these advantages because data placed into memory with DMA can be treated like regular memory. Unfortunately, the DMA initiation itself often uses some of the conventional solutions listed in Table 1.

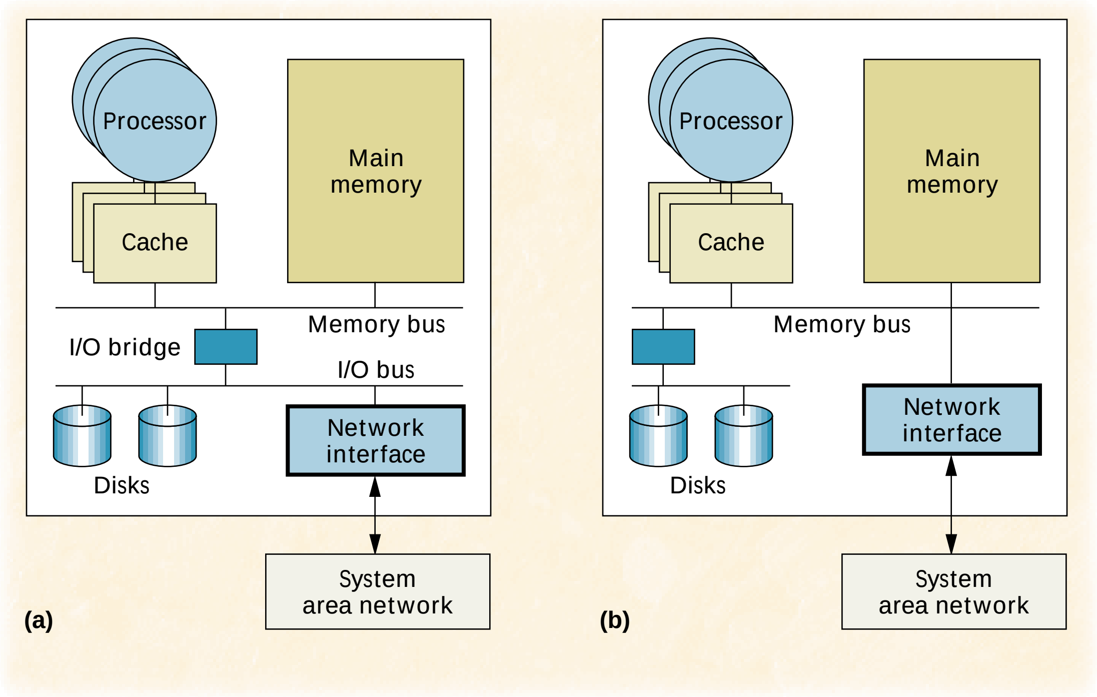
We argue that treating NI accesses like memory accesses is justified by the importance of network performance to future computers. Today an NI is a central piece of hardware. We therefore believe NIs should be treated as standard equipment, just like main memory or frame buffers, and not as an optional or peripheral add-on.

KEY COMPONENTS

An NI in a host node is essentially a device that lets a processor send and receive messages from a network that connects such nodes. The network accepts messages from an NI and delivers them to one or more NIs that are also connected to the network. An NI consists of two parts:

- The *internal* NI is the NI's interface to the processor, main memory, and disks. The internal NI contains the logic and memory that the processor uses to send and receive messages to and from the NI. For example, a processor can send a message to the network by writing messages to the internal NI's data registers.
- The *external* NI is the NI's interface to the network. An external NI performs network-specific functions, such as cyclic-redundancy checks and network-specific framing.

An internal NI consists of two parts: the *send interface* and the *receive interface*. Each interface consists of four components:



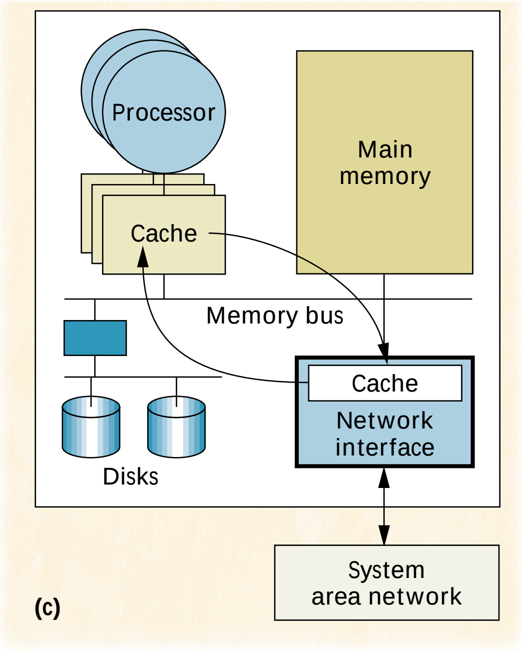


Figure 1. The architecture of a standard workstation node with the network interface (a) on the I/O bus, (b) on the memory bus, and (c) on the memory bus with a cache.

- *Status registers* contain NI status information. A receive-interface status register, for example, can indicate that a new message has arrived from the network, and a send-interface status register can indicate that the NI has successfully injected a message into the network.
- *Control registers* let a user process pass information and commands to the NI device. For example, a processor may temporarily disable NI interrupts by writing to an NI control register.
- *Data registers* contain message data sent by a processor or received by the NI from the network.
- *Notification mechanisms* help an NI inform a process of any change in NI device status. For example, the NI can interrupt the process on a change in device status, such as the arrival of a message from the network.

To send a message to the network, a processor first reads the send-interface status register to ensure there is enough space in the send interface's data registers. If there is enough space, the processor writes a new message to the data registers. If there is not enough space, the processor can either poll the NI periodically or have the NI notify the processor when space becomes available. When the NI's internal interface gets the message, it hands the message to the external NI, which injects the message into the network.

When a message from the network arrives at an NI, the external NI extracts the message from the network and hands it to the receive interface. The receive interface writes the message to its data registers and sets a status

Table 1. Summary of conventional and proposed solutions.

Problem	Conventional solution	Proposed solution
Virtualized by	Operating system	Virtual memory hardware
Location	I/O bus	Memory bus
Cache NI registers	Not allowed	Allowed
Out-of-order and speculative access	Not allowed	Allowed
Application Programming Interface (API)	Has side effects	Has no side effects

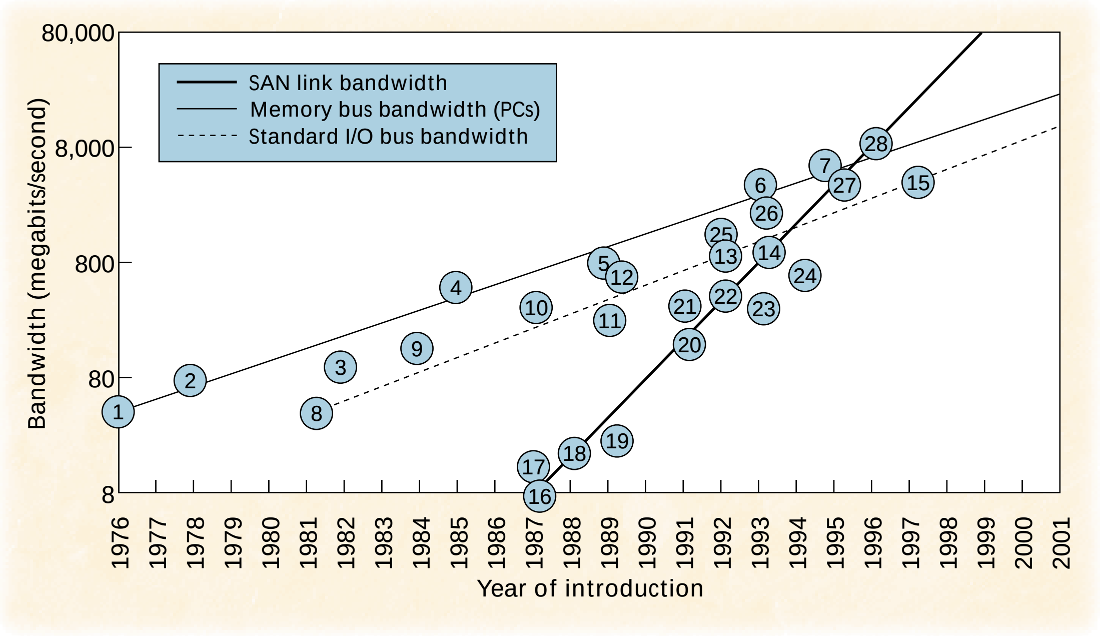


Figure 2. In PCs and standard I/O buses, the peak SAN link bandwidth is growing 100 percent per year, while the peak memory and I/O bandwidths are growing roughly 30 percent per year. Memory buses in the figure: (1) Intel 8085, (2) Intel 8086, (3) Intel 80286, (4) Intel 80386, (5) Intel 80486, (6) Intel Pentium, (7) Intel Pentium Pro. I/O buses: (8) IBM PC, (9) ISA bus, (10) MCA, (11) EISA bus, (12) Sun 32-bit/20-MHz Sbus, (13) VESA, (14) 32-bit/33-MHz PCI, (15) 64-bit/66-MHz PCI. SANs: (16) TMC CM-2, (17) nCube/ten, (18) Intel iPSC/2, (19) MasPar MP-1216, (20) Thinking Machines CM-5, (21) Intel Delta, (22) Meiko CS-2, (23) IBM SP-2, (24) Myricom Myrinet, (25) Intel Paragon, (26) Cray T3D, (27) Cray T3E, (28) SGI/Craylink.

register that indicates the presence of a new message.

Flow control (such as a return-to-sender function) ensures messages are rarely lost if the data registers are full. If the processor has appropriately set the control registers, the NI can use a processor interrupt to send a notification to a processor in the receive host node about the message's arrival. A processor in the receive host node then reads the new message from the NI data registers.

VIRTUALIZING THE NI

Virtualizing a physical resource (to a user process) requires two mechanisms: *protection* and *address translation*. Protection isolates user processes from one another. Address translation lets a user process access a physical device through virtual addresses. The OS virtualizes a disk by requiring that all disk accesses be initiated through OS trap commands. However, trapping to the OS is usually expensive because modern microprocessors do not support traps efficiently.

In contrast, main memory gets virtualized through the virtual memory hardware (which is supported by all high-performance microprocessors today), generally without OS intervention. Main memory is divided into physical pages and mapped to user virtual space on demand. A hardware structure called the *translation lookaside buffer* translates user virtual page addresses to physical page addresses in main memory. Consequently, main memory accesses are much faster (less than a microsecond) than disk accesses (typically more than 10 to 100 microseconds).

Accessing NI memory using virtual memory can thus dramatically improve performance. The OS simply maps NI memory pages directly into user space. The virtual memory hardware translates these memory-mapped virtual addresses to appropriate physical addresses in the NI memory and ensures protected access.

For example, the NIs in Thinking Machines' CM-5 and, more recently, Myricom's Myrinet network let users access the NI memory directly using this technique. We call such NIs *user-level network interfaces* (or ULNIs), since the NI memory can be directly accessed from user space. Compaq, Intel, and Microsoft are jointly developing a ULNI specification called the *virtual interface* architecture,[2] which will allow a user process to bypass the OS while sending and receiving messages to and from the network.

THE MEMORY BUS

In a standard workstation node (as shown in Figure 1a), disks are typically located on the peripheral I/O bus. This location is dictated primarily by the availability of a standard I/O bus interface (like SBus or PCI). Unlike I/O buses, current memory buses are usually proprietary and have nonstandard interfaces. Consequently, manufacturers do not usually design disk interfaces to memory bus specifications.

Current memory buses, however, offer significant performance advantages: lower latency, higher bandwidth, and support for optimized single-writer coherence protocols.

Lower latency and higher bandwidth

Memory buses offer much lower latency and higher bandwidth than I/O buses. For example, the current generation of PC memory buses run at 66 to 75 MHz, which is more than two times faster than the current generation of 33-MHz PCI buses. Additionally, all I/O bus accesses typically traverse the memory bus and the I/O bridge, which connects proprietary memory buses to standard I/O buses.

Current memory buses offer peak bandwidths of more than 4 Gbps. Some of the Sun Enterprise servers support an even more aggressive memory bus called the Ultragigaplane, which offers a sustained band-

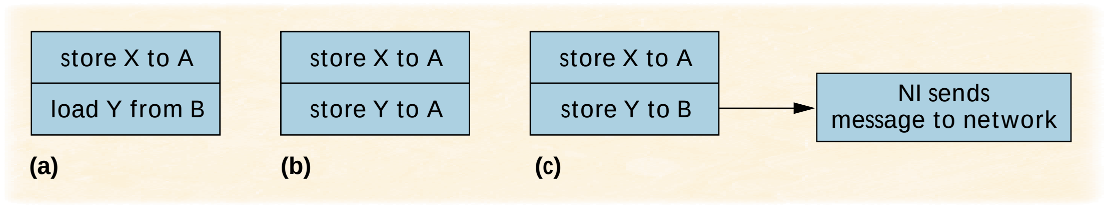


Figure 3. Processor accesses to ULNI device memory have three kinds of side effects. In this figure, the instructions are uncached loads and stores to ULNI registers that are memory-mapped to virtual addresses A and B. In (a), the store-load pair must be in order for some network interfaces (such as in Princeton Shrimp's Ultra DMA technique) to work correctly, even though the instructions appear unrelated to the processor. In (b), two consecutive stores to the same address must occur in order (such as in Thinking Machines' CM-5 NIs). Thus, in (a) and (b), a previous store implicitly determines the next uncached load or store that a network interface expects. In (c), the second store triggers an action in the network interface, such as the sending of a message into the network.

width of 20 Gbps. Such buses achieve high bandwidth via high clock speeds, large widths (between 64 and 256 bits), and overlapped bus transactions.

Figure 2 compares the trends in peak link bandwidth of SANs, memory buses in PCs, and standard I/O buses. This figure suggests that there will continue to be a gap between the bandwidths of memory and I/O buses. In fact, I/O bus bandwidth lags behind memory bus bandwidth by at least four years. Consequently, NI cards designed for I/O buses will not harness full memory-bus bandwidth.

Optimized coherence protocols

Memory buses support optimized single-writer coherence protocols. These protocols allow processor caches to cache and share memory easily because they provide a single and consistent image of physical memory across all processor caches.

The performance advantages of memory buses suggest that ULNIs should be placed on memory buses, just like main memory, as shown in Figure 1b. The main problem with memory buses is that designers do not currently export a standard interface. The advent of ULNIs as standard equipment, like memory or frame buffers, emphasizes the need for designers to do so.

Companies such as Intel, IBM, and Sun Microsystems, which manufacture both microprocessors and network-centric computers, could allow system designers to design ULNIs to their internal memory bus. Intel's MPP Teraflop supercomputer, for example, attaches a ULNI device directly to the Pentium Pro memory bus. Independent vendors may have to acquire memory bus specifications from microprocessor companies to develop a standard interface for the memory bus. Corollary, for example, has taken such an approach, albeit in a different context, to glue together two four-processor Pentium systems into an eight-processor Pentium Pro SMP node.[3] Alternatively, manufacturers of proprietary memory buses could provide special bridges to other open-standard interfaces, such as PCI.

Bridging the bus

A standard bridge might connect to a standard I/O bus but may not provide the performance or coherence access needed by ULNIs. A more aggressive bridge could convert directly to a standard I/O bus connector that supports a single I/O device without a physical I/O bus. This bridge could fake the I/O bus signals to offer higher performance with no arbitration time.

Silicon Graphics' Power Challenge, for example, uses this type of bridge (which the company calls a *personality interface*) to convert between SGI's proprietary I/O bus and a standard SCSI device. Similarly, Intel's Accelerated Graphics Port is a standard bridge that offers graphics accelerators a dedicated high-bandwidth path to main memory. An even more aggressive bridge can convert to a device-specific interface that is proprietary but less demanding and more stable between product generations than a memory bus. If network connections become standard equipment, this option would provide a way to obtain performance comparable to that of a memory bus ULNI at a lower cost.

Another possibility is standardizing the interface between the internal and external NIs. Microprocessor vendors could provide the internal interface that communicates with the processor. Third-party vendors could provide the external interface that communicates with the network. This technique would keep third-party vendors from having to deal with a particular memory bus' coherence protocol and let microprocessor vendors deliver the network's performance to a user process via its own optimized internal interface.

CACHING NI REGISTERS

Conventional NI registers are marked uncacheable. ULNI registers, on the other hand, can be cached in processor caches and in ULNIs themselves, which offers several advantages.

Uncached access to conventional NI registers

Unlike processor accesses to regular cacheable memory, processor accesses to ULNI device memory often have side effects (shown in Figure 3), such as sending a message into the network. Because of such side effects, NIs often require loads and stores to ULNI memory to appear to be in order. In current microprocessors, the simplest way to ensure this is to mark these loads and stores uncacheable.

Furthermore, ULNI memory behaves more like a processor cache than like main memory. This is because it can generate new data—on message reception, for example—just like a processor generates new data on a store. In contrast, main memory is a passive device that can only return data stored to it. If a processor is allowed to cache ULNI memory locations (like message buffers), the ULNI must have the ability to invalidate these memory locations when a new message arrives. However, most NIs reside on I/O buses, which usually do not support invalidation signals.

Finally, caching ULNI registers in processor caches requires support for ULNI register reuse.[4] Conventional ULNIs do not have to remember the value of a ULNI register once a processor reads it because processor loads are atomic. Unfortunately, a processor's loads to words in a cache block are not atomic

To harness SAN benefits effectively, we must design NIs more like memory than like disk interfaces.

because—in the case of a cache replacement—a processor's cache can lose the cache block from the ULNI before the processor has a chance to read all the words in the cache block. The ULNI registers require a handshake between the processor and ULNI to allow explicit reuse of the cache block.

Designing a ULNI's API carefully can eliminate side effects in ULNI memory accesses and the need to support ULNI register reuse. The ULNI device memory and processor caches can be kept coherent by placing the ULNI device on the memory bus, which allows a ULNI to observe and participate in the system's coherence protocol and thereby generate invalidation signals when necessary.

Caching NI registers in processor caches

Caching status or control registers in processor caches helps remove unnecessary memory bus traffic. If a processor were polling an uncached status register, every processor poll would go across the memory bus to the ULNI device. Unsuccessful polls—those that do not find messages in the NI—waste memory bus bandwidth that could be used by other processors in an SMP node.

If the status register were cached, all unsuccessful polls would hit in the processor's cache. If a message were to arrive and change the ULNI status, the ULNI device would invalidate the cached status register in the processor's cache. On its next poll attempt, the processor would incur a cache miss, which could be satisfied directly by the ULNI.

Furthermore, uncached accesses provide very low bandwidth compared to cache block accesses because they transfer only a few bytes of data—from 1 to 16 bytes. In contrast, cache blocks are typically much larger—from 32 to 128 bytes—and can exploit the full transfer bandwidth of today's memory buses.

Treat ULNI memory as a cache

Like processor caches, ULNI caches can cache ULNI registers. Instead of allocating ULNI registers in ULNI memory, however, the registers can be allocated in the user's virtual space and can be backed up by main memory. Like processor caches, ULNI caches can simply cache the portion of main memory that contains the ULNI registers.

Such ULNI caches help improve performance. Processor cache misses for ULNI registers can be intercepted and satisfied directly by the ULNI cache through a cache-to-cache transfer. Contrast this with data transfer via DMA, in which messages reach the processor cache in two steps (and consequently two memory bus crossings): from ULNI device to main memory and from main memory to the processor cache. This increase in latency may cause serious problems for latency-bound request-response protocols.

Also, the ULNI cache may overflow when bursts of messages arrive at a ULNI. ULNI cache replacements to main memory will buffer these messages automatically without processor intervention.[4] Contrast this with the more conventional and lower performance solutions in which processors must copy the data explicitly from memory-mapped ULNI registers to the user's virtual space.

OUT-OF-ORDER AND SPECULATIVE ACCESSES

To tolerate the latency of main memory access, processors use two techniques: *out-of-order access* and *speculative execution*. Out-of-order accesses allow loads and stores to bypass earlier loads or stores. A processor need not stall because of a cache miss on a particular load. Speculative execution tolerates memory access latency better than out-of-order accesses.

Processors speculate on control dependence (like branch prediction), data dependence, data addresses, and data values, and then perform computations based on these speculated values. If the speculation is correct, idle processor resources can be used effectively and memory access latencies can be tolerated. However, if the speculation is incorrect, all previous computation based on speculatively loaded values must be quashed and any process-specific state must be rolled back to the point at which the speculation started.

Processors do not usually perform out-of-order and speculative accesses to ULNI memory for three reasons:

- Many I/O buses do not adequately support multiple outstanding transactions, so processor accesses to NIs must be serialized on the I/O bus. This problem can be solved by interfacing the ULNI device with the memory bus, which usually supports multiple outstanding transactions.
- To avoid side effects, NI accesses are not generally performed out of order. Speculative loads to NI memory are also generally avoided because current NIs usually do not roll back side effects caused by incorrect processor speculation. Designing the ULNI's API carefully can eliminate this problem.
- Most microprocessors today disallow out-of-order and speculative accesses on uncached loads and stores, which are the predominant ways to access NIs. This problem can be solved by caching (and thereby speculating on) ULNI registers in processor caches and by not allowing a processor's speculatively stored state to be reflected outside the processor. Modern microprocessors support both mechanisms efficiently.

MEMORY-BASED QUEUES

A user process typically accesses a peripheral I/O device via the OS or it uses the underlying data movement primitive as the API to the I/O device. For example, user APIs based on program-controlled I/O use uncached loads and stores (the data movement primitives) to memory-mapped device registers as the user API to the I/O device. Similarly, Princeton's User-Level DMA[5] mechanism uses DMA transfers as the user API to the ULNI device. Instead of exposing the underlying data movement primitive as the user API, ULNIs could more effectively structure the ULNI date registers as memory-based queues.[2,4,6,7] Such memory-based queues cannot be classified as program- controlled I/O or as DMA.

Memory-based queues consist of two parts: a *send queue* and a *receive queue*. Each queue is allocated in virtual memory and managed as a circular buffer with head and tail pointers. To send a message, the processor queues the message at the tail of the send queue either by explicitly writing the message into the send queue's memory or by inserting a virtual pointer to the message in the send queue. The ULNI dequeues messages from the head by reading the send queue memory and, if necessary, translating the virtual pointer to its physical memory address and reading the message from the user virtual space.

For the receive queue, the ULNI similarly queues messages at the tail of the receive queue and the processor dequeues messages from the head. Device commands for such APIs are thus no longer explicit DMA-initiation requests. They are simple memory operations, such as incrementing or decrementing queue head or tail pointers. For example, when a processor queues a message to the send queue and increments the tail pointer, the ULNI interprets this action as a command to send a message to the network. If the tail pointer is uncached, the ULNI treats the increment as a signaling store. If the tail pointer is cached, the ULNI must poll the tail pointer for new messages.

There are four advantages to treating the ULNI API as a memory-based queue:

- Unlike uncached accesses or UDMA, memory-based queues decouple a processor and a ULNI. This decoupling enables the processor and the ULNI to send and receive multiple messages to and from the queues without blocking.
- Memory-based queues avoid side effects by treating ULNI queue accesses simply as side-effect-free regular-memory accesses. ULNI commands for such queues are primarily incrementing or decrementing queue pointers. This lets processors cache ULNI queues, perform out-of-order accesses on queue memory, and speculatively send and receive messages to and from these queues.
- Since memory-based queues are allocated like regular memory and managed as circular buffers, the reuse handshake is simple: A comparison of the head and tail pointers reveals whether a queue location can be reused.
- Memory-based queues simplify the problem of multiprogramming a ULNI for SMPs because the queues give each process protected, simultaneous access to the ULNI without invoking the OS while sending and receiving messages. In contrast, machines like the Thinking Machines' CM-5 allow only one user process to access the ULNI at one time and must context-switch the entire ULNI when it context-switches a user process.

There are at least three additional opportunities for improving the performance of processor-to-NI interactions.[8,9] Using virtual memory—instead of small and dedicated NI memory—to buffer network messages can provide megabytes of buffer space for network messages. Moving data between a processor and an NI in cache block units allows data to be read directly from the NI (like program-controlled I/O) as well as to be transferred in blocks (like DMA). Finally, using cache invalidations as notification signals—instead of heavyweight interrupts—lets an NI inform a processor of its status changes rapidly.

SANs have evolved to satisfy the increasing demand for high-bandwidth, low-latency networks. But the benefits can be realized only if we use lightweight protocols and efficient NIs.

SANs have evolved to satisfy the increasing demand for high-bandwidth, low-latency networks. But the benefits can be realized only if we use lightweight protocols and efficient NIs. We squander SANs' benefits, for example, if applications must invoke the OS to send and receive messages from the NI.

To harness SAN benefits effectively, NIs must be designed more like memory than like disk interfaces. This includes accessing the NI directly from user space through virtual memory hardware, placing the NI on the memory bus, caching NI registers, accessing the NI out of order and speculatively, and designing a side effect-free API. ❖

Acknowledgments

We thank Guri Sohi for inspiring this article. Jim Burnette, Rebecca Hoffman, Rich Martin, Larry Peterson, Avinash Sodani, and the anonymous reviewers provided valuable comments on earlier drafts. This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525, ARPA order #B550, CCR-9101035, MIP-9225097, MIPS-9625558, and donations from Sun Microsystems.

References

1. R.W. Horst, "TNet: A Reliable System Area Network," *IEEE Micro*, Feb. 1994, pp. 37-45.

2. D. Dunning and G. Regnier, “The Virtual Interface Architecture,” *Proc. Hot Interconnects V*, ACM Press, New York, Aug. 1997, pp. 47-58.
3. P. Vogt, “Profusion: A Buffered, Cache Coherent Crossbar Switch,” *Proc. Hot Interconnects V*, ACM Press, New York, Aug. 1997, pp. 87-96.
4. S.S. Mukherjee et al., “Coherent Network Interfaces for Fine-Grain Communication,” *Proc. 23rd Int’l Symp. Computer Architecture*, ACM Press, New York, 1996, pp. 247-258.
5. M.A. Blumrich et al., “Protected User-level DMA for the Shrimp Network Interface,” *Proc. Second IEEE Symp. High-Performance Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 154-165.
6. P. Druschel, L.L. Peterson, and B.S. Davie, “Experiences with a High-Speed Network Adaptor: A Software Perspective,” *Proc. SIGCOMM 94*, ACM Press, New York, 1994, pp. 2-13.
7. T. von Eicken et al., “U-Net: A User-Level Network Interface for Parallel and Distributed Computing,” *Proc. 15th ACM Symp. OS Principles*, ACM Press, New York, 1995, pp. 40-53.
8. S.S. Mukherjee and M.D. Hill, *A Survey of User-Level Network Interfaces for System Area Networks*, Tech. Report 1340, Computer Sciences Department, University of Wisconsin, Madison, 1997.
9. S.S. Mukherjee, *Design and Evaluation of Network Interfaces for System Area Networks*, PhD dissertation, University of Wisconsin, Madison, 1998.

Shubhendu S. Mukherjee is a senior hardware engineer on the Alpha Architecture team at Compaq Computer Corp. His current interests include microarchitectures for high-performance parallel computers. Mukherjee received a BTech from the Indian Institute of Technology, Kanpur, and an MS and a PhD from the University of Wisconsin, Madison. Mukherjee performed this work as a PhD candidate. Contact him at shubu@muhthr.hlo.dec.com.

Mark D. Hill is a professor and Romnes fellow in the Computer Sciences Department and the Electrical and Computer Engineering Department at the University of Wisconsin, Madison. He also codirects the Wisconsin Wind Tunnel parallel-computing project. His current research interests include memory systems of shared-memory multiprocessors and high-performance uniprocessors. Hill received a BSE from the University of Michigan, Ann Arbor, and an MS and a PhD in computer engineering from the University of California, Berkeley. Contact him at markhill@ cs.wisc.edu.