Making Network Interfaces Less Peripheral

Shubhendu S. Mukherjee and Mark D. Hill

Abstract

Much of a computer's value depends on how well it interacts with networks. To enhance this value, designers must improve the performance of networks delivered to users. Fortunately, the performance of networks is improving rapidly. Unfortunately, this dramatic improvement in network performance is seldom delivered to users. A key bottleneck is the host *network interface* (NI), which connects a network to a host computer. This bottleneck gets more severe as network and host performance continue to improve.

The problem with current NIs is that they were designed with an interface similar to that of a disk interface. Most current NIs require applications to use an operating system call, are placed on the I/O bus, do not allow processors to cache their registers, and force processors to interact with them with in-order and non-speculative accesses. The last two problems are partially due to the presence of "side-effects" in current NI designs.

While this kind of an interface may have been adequate in the past, we argue that future NIs should appear to their hosts more like memory than like a disk. Memory is virtualized without requiring operation system intervention (in the common case), is on the memory bus, can be cached, can be accessed out of order and speculatively, and is free of any side-effects. We discuss how to do the same for NIs, so that the dramatic improvements in network performance can be delivered to users.

1 Introduction

Much of a computer's value depends on how well it interacts with networks. To enhance this value, designers must improve the network performance delivered to users. The commonly quoted aspect of network performance is *bandwidth*. Bandwidth is the rate at which data flows through the network and computer. High bandwidth is critical for transmitting high-quality video or large files. An under-appreciated aspect of network performance is *latency*. Latency is the user-to-user delay for sending a message. Latency determines the performance of protocols that send many small messages, as can be found in network file systems, database lock managers, and fine-grain parallel computing [6, 2].

Fortunately, networks are improving rapidly. In particular, local area network (LAN) bandwidth has improved from 10-100 megabits/second to one gigabit/second or more. Aggressive LANs, such as the Myricom Myrinet or the Tandem Servernet, have moved so far that some view them as a new class of networks called a *system area network* or SAN [5]. SANs improve performance in two ways. First, aggressive links and switches provide very high bandwidth and extremely low latency. Second, reliability properties of SANs allow systems to use lean communication layers (e.g., Active Messages [12]) instead of heavy-weight and one-size-fits-all protocols (e.g., TCP/IP). Consequently, SANs help improve the performance of both network hardware (links and switches) and network software (communication protocols).

Unfortunately, improvements in network hardware and software are rarely delivered to users. A key problem is inadequate host *network interfaces* (NIs). A NI connects a network to a host computer that runs the network software. A NI includes hardware that sits typically on an I/O bus and exposes an internal interface (e.g., device registers) to a host processor. Most NIs have low-level software (usually in a device driver) inside the operating system that applications use to access the network. Figure 1a illustrates the host location of a conventional NI.

The problem with current NIs is that they were designed with an interface similar to a disk's interface. Most current NIs require applications to use an operating system call, are placed on the I/O bus, do not allow caching of device registers, and force processors to interact with them with in-order and non-speculative accesses. The last problem is

Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

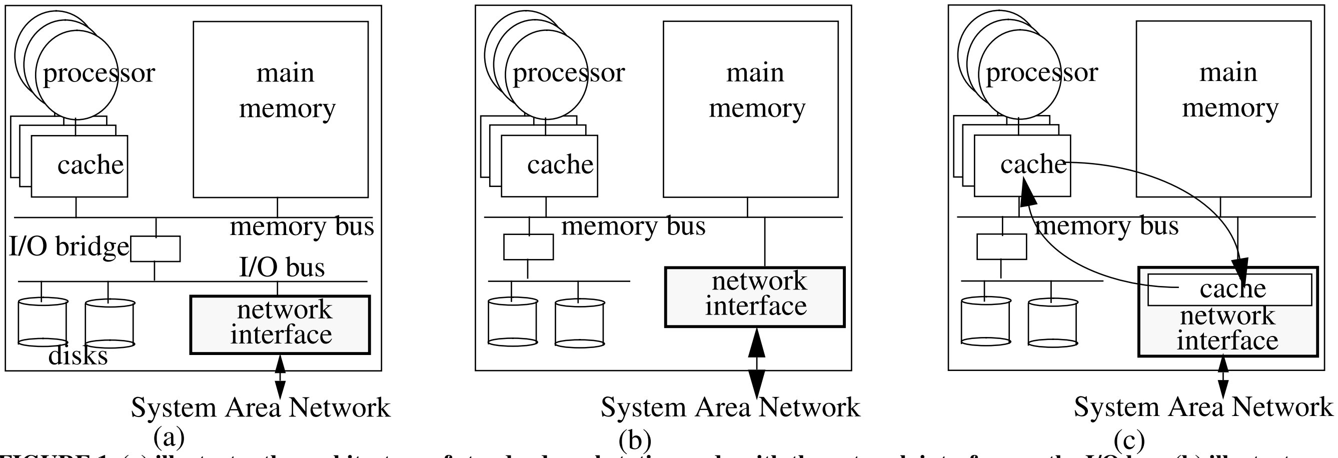

FIGURE 1. (a) illustrates the architecture of standard workstation node with the network interface on the I/O bus, (b) illustrates the same workstation node with the network interface on the memory bus, and (c) augments the network interface in (b) with a cache.

subtle and partly caused by hosts communicating with NIs using memory operations that are overloaded with side-effects (e.g., a load to a NI device register both returns a value and deletes it from the device).

Such NIs will not be adequate in the future. First, emerging SANs deliver bandwidth (10 gigabits/second or more) and latency (10s of nanoseconds) that are between two to four orders of magnitude better than that delivered by current disks. Second, new hosts will demand much higher performance than in the past because of faster processors, multimedia extensions, and/or multiple processors. If NIs do not respond they will become the bottleneck in network performance.

We argue that future NIs should appear to their hosts more like memory than like a disk. Memory is virtualized without requiring operation system intervention (in the common case), is on the memory bus, can be cached, can be accessed out of order and speculatively, and do not have side-effects. We propose to do the same for NIs, as summarized in Table 2. Traditional NIs that use direct memory access (DMA) offer some of these advantages because data DMA-ed into memory can be treated just like regular memory. Unfortunately, the DMA initiation itself often uses conventional solutions listed in Table 2.

Treating NI accesses like memory accesses is justified by the importance of network performance to future computers. Today a NI is a central piece of hardware for a computer. Therefore we believe that a NI should be treated as a "standard equipment" just like main memory or frame buffers and not as an optional and peripheral add-on.

The rest of the paper is organized as follows. The next section discusses a NI's key components. Rest of the sections discuss the opportunities for improving the performance of NI accesses, as listed in Table 2. More details on the opportunities for improvement are available in a survey paper [8].

2 Key Components of a Network Interface

This section examines network interface aspects in more detail to provide a foundation for the optimization sections that follow. A network interface (NI) in a host node is a device that allows a processor to send and receive messages from a network that connects these host nodes. The network accepts messages from a NI and delivers them to one or more NIs connected to the network. A NI consists of two parts, the internal NI and the external NI. We define the internal NI as the NI's interface to the processor, main memory, and (perhaps) disks, and external NI as the NI's interface to the network. The internal NI contains 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 data registers of the internal NI. An external NI performs network-specific functions, such as cyclic-redundancy checks, network-specific framing, etc.

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

Problems	Solutions		Discussed
	Conventional	Proposed	
Virtualize via	operating system	virtual memory hardware	Section 3
Location	I/O bus	memory bus	Section 4
Cache NI registers	not allowed	allowed	Section 5
Out-of-order and speculative access	not allowed	allowed	Section 6
Application Programming Interface (API)	has side-effects	no side-effects	Section 7
Summary: NI access similar to	disk interface access	memory access	

TABLE 2. Summary and forecast of paper

Status Registers. NI 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. NI control registers allow a user process to pass information and commands to the NI device. For example, a processor may temporarily disable NI interrupts by writing to a NI control register.

Data Registers. NI data registers contain message data sent by a processor or received by the NI from the network.

Notification mechanism. A NI informs a process of any change in NI device status through a notification mechanism. For example, the NI can interrupt the process on a change in device status, such as 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 it when free space becomes available. On receiving the new message in its data register the NI hands the message to the external NI, which injects the message into the network.

When a message arrives at a NI, the external NI extracts the message from the network and hands the message to the receive interface. The receive interface writes the message to its data registers and sets a status register that indicates the presence of a new message. Flow control (e.g., return-to-sender) is typically used to ensure messages are never (or rarely) lost if the data registers are full. If the control registers have been appropriately set by the processor, the NI can send a notification to a processor in the receive host node about the arrival of this message through a processor interrupt. Finally, a processor in the receive host node reads the new message from the NI data registers.

3 Use Virtual Memory Hardware to Virtualize the Network Interface

There is a marked difference in how user processes access a disk and main memory. Both are shared physical resources virtualized across multiple user processes. Virtualizing a physical resource to a user process requires two mechanisms: protection and address translation. Protection isolates user processes from one another. Address translation allows a user process to access a physical device through virtual addresses. The operating system virtualizes a disk by requiring that all disk accesses be initiated through operating system traps. However, trapping to the operating system is usually expensive because modern microprocessors do not support them very efficiently. In contrast, main memory is virtualized through the virtual memory hardware, which is supported by all high-performance microprocessors today, and does not involve operating system intervention in the common case. Main memory is divided into physical pages and mapped to user virtual space on demand. A hardware structure called the Translation Lookaside Buffer rapidly translates user virtual page addresses to physical page addresses in main memory. Consequently, main memory accesses are much faster (less than a microsecond) compared to disk accesses (greater than 10 - 100 microseconds).

Accessing NI memory using virtual memory, and not the operating system, can therefore dramatically improve performance. The operating system 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 to it.

The NIs in the Thinking Machines' CM-5 and, more recently, the Myricom Myrinet network allow users to directly access the NI memory using this technique. We call such NIs *User-Level Network Interfaces (ULNIs)* since the NI memory can be directly accessed from user space. Compaq, Intel, and Microsoft Corporations are jointly developing such a ULNI specification called the Virtual Interface (VI) architecture [4]. The VI architecture is a logical specification that will allow a user process to bypass the operating system while sending and receiving messages from the network.

4 Place the Network Interface on the Memory Bus

In a standard workstation node (Figure 1a) disks are typically located on the peripheral I/O bus. The choice of this location is dictated primarily by the availability of a standard I/O bus interface (e.g., SBus, PCI), which enables independent vendors to manufacture NI cards to these standard specifications. Unlike I/O buses, current memory buses are usually proprietary and have non-standard interfaces. Consequently, manufacturers do not design usually disk interfaces to memory bus specifications.

Current memory buses, however, offer two significant performance advantages over I/O buses. First, memory buses offer much lower latency and higher bandwidth than I/O buses. For example, the current generation of PC memory buses are clocked at 66-75 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 bandwidth exceeding four gigabits per second. Some of the Sun Enterprise servers support an even more aggressive memory bus called the Ultragigaplane, which offers a sustained bandwidth of 20 gigabits per second. Such high bandwidth is achieved via high clocks, large widths (between 64- to 256-bits), and overlapped bus transactions.

Figure 3 shows the trends in peak link bandwidth of SANs, memory buses in personal computers, and standard I/O buses. This figure suggests that the gap between bandwidths of memory and I/O buses will continue to exist in future. In fact, I/O bus bandwidth lags behind memory bus bandwidth by at least four years. In other words, I/O buses will take another four years to achieve the peak bandwidth offered by today's memory buses. Consequently, NI cards designed to I/O buses will not be able to harness the full memory bus bandwidth. Figure 3 also shows that SAN link bandwidth is growing at a much faster rate than the bandwidth of PC memory buses. For such SANs we will need more aggressive memory buses, such as the SUN Ultragigaplane.

Second, memory buses support optimized single-writer coherence protocols, which allow processor caches to cache and share memory easily. This is because these single-writer coherence protocols provide a single and consistent image of physical memory across all processor caches. The next section argues how and why caching message data in processor and ULNI caches can help improve performance.

The performance advantages of memory buses suggest that ULNIs should be placed on memory buses, just like main memory (Figure 1b). The main problem with memory buses is that they do not export a standard interface. However, the advent of ULNIs as standard equipment, like memory or frame buffers, emphasizes the need for memory bus designers to export a standard interface to either systems designers internal to a company or third-party vendors manufacturing independent ULNI devices. Companies such as Intel, IBM, and Sun Microsystems that manufacture both microprocessors and network-centric computers can allow system designers to design ULNIs to their internal memory bus. Intel's MPP supercomputer called Teraflop, for example, attaches the ULNI device directly on the PentiumPro memory bus. For independent vendors finding a standard interface on the memory bus may imply coordinating with microprocessor companies to acquire their memory bus specification. Corollary Inc. has taken such an approach, albeit in a different context, to glue together two four-processor PentiumPro systems into a eight-processor PentiumPro SMP node [10]. Alternatively, manufacturers of proprietary memory buses could provide special *bridges* to other open standard interfaces, such as the PCI.

The bridge needed converts proprietary memory bus signals to other standard signals. A standard bridge might connect to a standard I/O bus (e.g., PCI). A standard bridge supports many standard devices 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 one demanding I/O device without a physical I/O bus. This bridge can fake the I/O bus signals to offer higher performance (e.g., no arbitration time) to standard devices. SGI's Power Challenge, for example, uses this type of bridge (which they call a "personality interface") to convert between SGI's proprietary I/O bus

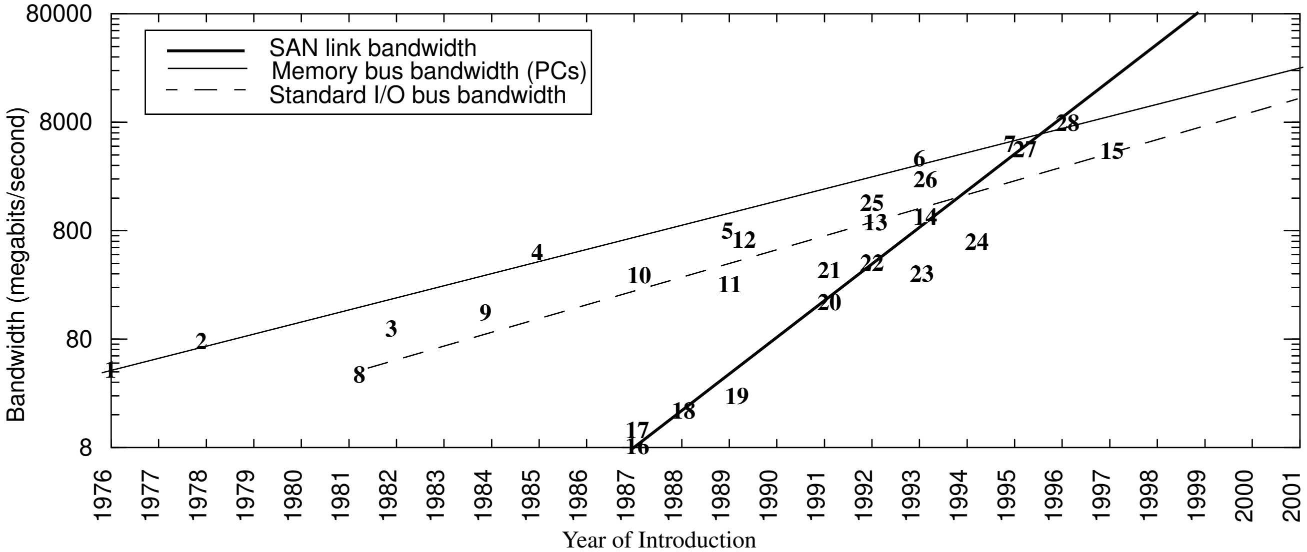

FIGURE 3. This figure shows the trends in peak link bandwidth of SANs and peak bandwidth of memory buses in personal computers and standard I/O buses. The peak SAN link bandwidth is growing at 100% per year, while the peak memory and I/O bus bandwidths are growing at roughly 30% per year. Legend for memory bus: (1) 8085 (2) 8086 (3) 80286 (4) 80386 (5) 80486 (6) Pentium (7) PenitumPro. Legend for I/O bus: (8) IBM PC (9) ISA bus (10) MCA (11) EISA bus (12) 32-bit/20-MHz Sbus (13) VESA (14) 32-bit/33-MHz PCI (15) 64-bit/66-MHz PCI. Legend for System Area Networks: (16) TMC CM-2 (17) nCube (18) Intel iPSC/2 (19) Maspar (20) TMC 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.

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 like frame buffers then this option provides an attractive way to obtain performance comparable to a memory-bus ULNI without some of the cost.

Another possibility is standardizing the interface between the internal and external NIs. Microprocessor vendors can provide the internal interface that communicates with the processor and third-party vendors can provide the external interface that communicates with the network. This relieves third-party vendors from details of a particular memory bus's coherence protocol and allows microprocessor vendors to deliver the network's performance to a user process via its own optimized internal interface.

5 Cache Network Interface Registers in Processor and Network Interface Caches

This section discusses why conventional NI registers are marked uncacheable and what are the advantages of caching ULNI registers in processor caches and ULNIs.

5.1 Why conventional NI registers are marked uncacheable?

Disk interface memory is typically not cached in processor caches. Instead device memory is usually marked uncacheable for three reasons. First, processor accesses to ULNI device memory often have side-effects (Figure 4), unlike processor accesses to regular cacheable memory. For example, a processor's store to regular cacheable memory does not have side-effects, such as sending a message into the network. In contrast, a processor's store to ULNI memory may have such a side-effect. Because of such side-effects, NIs often require loads and stores to ULNI memory to appear strictly in order. In current microprocessors the simplest way to ensure this is to mark these loads and stores uncacheable.

Second, the ULNI memory behaves more like a processor cache than main memory. This is because it can generate new data (e.g., on message reception) just like a processor. In contrast, main memory is a passive device that can only return data that has been stored to it. Consequently, if a processor is allowed to cache ULNI memory locations (e.g., message buffers), the ULNI must have the ability to invalidate these memory locations when a new message arrives. Unfortunately, most NIs reside on I/O buses, which usually do not support invalidation signals.

(a)	(b)	(c)
store X to A	store X to A	store X to A
load Y from B	store Y to A	store Y to B → NI sends message to network

FIGURE 4. This figure shows three kinds of side-effects that exist in NI designs. The instructions shown in this figure are uncached loads and stores to ULNI registers memory-mapped to virtual addresses A and B. (a) shows that the store-load pair must be strictly in order for some NIs to work correctly (e.g., Princeton UDMA initiation), even though the instructions appear unrelated to the processor. (b) shows two consecutive stores to the same address must occur in order (e.g., TMC CM-5 NI). (c) extends (b) to show that the second store (in the general case, the "n"th store) can trigger an action in the NI, such as sending a message into the network. (a) and (b) have the side-effects that a previous store determines implicitly the next uncached load or store a NI expects. (c) has the side-effect of sending a message on a store.

Third, caching ULNI registers in processor caches requires support for ULNI register reuse [7]. 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 because a processor's cache could lose the cache block read from the ULNI due to a cache replacement. Consequently, ULNI registers read via a cache block requires a handshake between the processor and ULNI to allow explicit reuse of the cache block.

The first and third problems—presence of side-effects in ULNI memory accesses and support ULNI register reuse—can be eliminated by designing the application programming interface carefully (discussed later). The second problem—keeping ULNI device memory and processor caches coherent—can be solved by placing the ULNI device on the memory bus. This would allow a ULNI to directly observe and participate in the system's coherence protocol, and thereby generate invalidation signals when necessary.

5.2 Advantages of caching ULNI registers in processor caches

Caching ULNI registers in processor caches offers two advantages [7]. First, caching status or control registers in processor caches helps remove unnecessary memory bus traffic. For example, if a processor were polling on an uncached status register, every processor poll would go across the memory bus to the ULNI device. Unsuccessful polls—that is, those that do not find messages in the NI—waste precious memory bus bandwidth, which could be used by other processors in an SMP node. Instead, if the status register is cached all unsuccessful polls will hit in the processor's cache. When a message arrives finally and the ULNI status changes, the ULNI device invalidates the cached status register in the processor's cache. On its next poll attempt, the processor will incur a cache miss, which can be satisfied directly by the ULNI.

Second, uncached accesses provide very low bandwidth compared to cache block accesses because they transfer only a few bytes of data (e.g., 1-16 bytes). In contrast, cache blocks are typically much larger (e.g., 32-128 bytes) and are capable of exploiting the full transfer bandwidth of today's memory buses.

5.3 Advantages of treating ULNI memory as a cache

Like processor caches ULNI caches can cache ULNI registers. Instead of allocating ULNI registers in ULNI memory the registers can be allocated in the user's virtual space and backed-up by main memory. Like processor caches ULNI caches can cache simply the portion of main memory that contains the ULNI registers. Such ULNI caches help improve performance in at least two ways. First, 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 become critical for latency-bound, request-response protocols.

Second, the ULNI cache may overflow when bursts of messages arrive at a ULNI. However, ULNI cache replacements to main memory will buffer these messages automatically without any processor intervention [7]. Contrast this with the more conventional and lower performance solution in which processors must copy the data explicitly from memory-mapped ULNI registers to the user's virtual space.

6 Allow Out-of-Order and Speculative Accesses to Network Interface Memory

To tolerate the latency of main memory access, processors use two techniques: out-of-order (OOO) access and speculative execution. OOO accesses allow loads and stores to bypass earlier loads or stores. Consecutively, a processor need not stall because of a cache miss on a particular load. Speculative execution is more aggressive than OOO accesses in tolerating memory access latency. Processors speculate on control dependence (e.g., branch prediction), data dependence, data addresses, and data values, and perform computations based on these speculated values. If the speculation is successful, idle processor resources can be used effectively and memory access latencies can be tolerated. However, if the speculation is incorrect, then all previous computation based on speculatively loaded values must be squashed and any process-specific state must be rolled back to the point at which the speculation started. In the context of messaging, we want processors to read from and write messages to ULNI memory speculatively, just like regular memory.

Processors do not usually perform OOO and speculative accesses to ULNI memory for three reasons. First, many I/O buses do not adequately support multiple outstanding transactions, which forces processor accesses to NIs to be serialized on the I/O bus. Second, as discussed in the last section, the presence of side-effects in NIs often force NI accesses to be performed in order preventing OOO accesses. Further, current NIs usually do not provide any mechanism to rollback any side-effects if the processor's speculation is incorrect, which prevents speculative loads to NI memory. Third, the most microprocessors today disallow OOO and speculative accesses on uncached loads or stores, which is the predominant way in which NIs are accessed.

The first problem—the absence of support for multiple outstanding transactions on common I/O buses—can be solved by interfacing the ULNI device to the memory bus, which usually supports multiple outstanding transactions. The second problem—the presence of side-effects in ULNI memory accesses—can be eliminated by designing the application programming interface to the ULNI carefully (see next section). Finally, the third problem—absence of OOO and speculative access to uncached I/O space—can be solved by caching ULNI registers in processor caches on which processors can speculate and not allowing a processor's speculatively stored state to be reflected outside the processor. Both mechanisms are supported efficiently by modern microprocessors today.

7 Use Memory-Based Queues as Application Programming Interface

Typically a user process accesses a peripheral I/O device via the operating system or uses the underlying data movement primitive as its Application Programming Interface (API) to the I/O device. For example, user APIs based on program-controlled I/O uses 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 UDMA [1] mechanism uses DMA transfers as the user API to the ULNI device. We argue that instead of exposing the underlying data movement primitive as the user API, ULNIs should structure the ULNI date registers as *memory-based* queues [3, 11, 7]. Such memory-based queues can be classified neither as program-controlled I/O nor 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 enqueues the message at the tail of the send queue either by explicitly writing the message into the send queue memory or by inserting a virtual pointer to the message into the send queue. The ULNI dequeues messages from the head by reading the send queue memory and, if necessary, translating the virtual pointer to the message to its physical memory address and subsequently reading the message from the user virtual space. For the receive queue, the ULNI similarly enqueues messages at the tail of the receive queue and the processor dequeues messages from the head. Device commands for such APIs are no longer explicit DMA-initiation requests; instead, ULNI device commands are simple memory operations, such as incrementing or decrementing queue head or tail pointers. For example, when a processor enqueues a message to the send queue and increments the tail pointer, the ULNI interprets this as a device command to send a message to the network. If the tail pointer is uncached, then the ULNI treats the increment as a signalling store; if the tail pointer is cached, the ULNI must poll on the tail pointer for new messages.

There are four advantages to treating ULNI API as memory-based queues. First, unlike uncached accesses or UDMA, memory-based queues decouple a processor and a ULNI. This enables both the processor and ULNI to send and receive multiple messages to and from the queues without blocking.

Second, 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 allows processors to cache ULNI queues, perform OOO accesses on queue memory, and speculatively send and receive messages to and from these queues.

Third, 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 or not.

Fourth, memory-based queues simplify the problem of multiprogramming a ULNI for SMPs. This is because memory-based queues provide each process protected, yet simultaneous, access to the ULNI without invoking the operating system while sending and receiving messages. In contrast, machines, such as 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.

8 Additional Opportunities

There are at least three additional opportunities for improving the performance of processor-NI interactions. Details about these opportunities can be found in Mukherjee and Hill [8]. These opportunities are using virtual memory to buffer network messages, moving data between a processor and a NI in cache block units, and notifying a processor of NI events via cache invalidations. Virtual memory is typically huge in today's computers and, therefore, can provide such large amounts of buffering. High-performance ULNI devices can demand large amounts—that is, tens of megabytes—of buffer memory to accommodate a wide variety of protocols, support large degree of multiprogramming, handle large bursts of messages, and guarantee reliable delivery of messages via flow control. Using cache blocks to read and write NI memory allows a processor to directly read data out of a NI, like program-controlled I/O, and transfer data in blocks, like DMA. Finally, using cache invalidations as notification signals, instead of heavyweight interrupts, allows a NI to rapidly inform a processor of its status changes.

9 Conclusions

A new generation of networks called System Area Networks (SANs) has evolved to satisfy the increasing demand for high-bandwidth, low-latency networks. The benefits of SANs are realized in applications only if light-weight protocols (not TCP/IP) and efficient network interfaces are used. The benefits of SANs are squandered, for example, if applications must invoke the operating system to send and receive messages. In contrast, User-level Network Interfaces (ULNIs) allow host applications to access the network interface directly without compromising protection by memory mapping internal interface registers into user space.

The exponential improvement in microprocessors' and SANs' performance and the advent of SMPs indicate that processor accesses to ULNIs will become a critical bottleneck for computer systems built with SANs. Processor accesses to ULNI registers is simply reading and writing ULNI memory. Nevertheless, most ULNIs treat such accesses as disk interface accesses that can have side-effects (e.g., a message send). Such treatment disallows current ULNIs to take advantage of memory access optimization techniques, such as traditional caches, out-of-order accesses, and speculation. Instead, we argued that ULNI memory accesses should be treated as regular side-effect-free memory accesses. Memory is virtualized without requiring operation system intervention (in the common case), is on the memory bus, can be cached, can be accessed out of order and speculatively, and is free of any side-effects. We discussed how to do the same for NIs, so that the dramatic improvements in network performance can be delivered to users.

Acknowledgments

We would like to thank Guri Sohi for inspiring this paper. Rebecca Hoffman, Rich Martin, Larry Peterson, Avinash Sodani, and the anonymous reviewers provided valuable comments on earlier drafts of this paper.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, CCR-9101035, MIP-9225097, and MIPS-9625558, and donations from Sun Microsystems. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

References

[1] Matthias A. Blumrich, Cesary Dubnicki, Edward W. Felten, and Kai Li. Protected User-level DMA for the SHRIMP Network Interface. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[2] R. Cypher, A. Ho, S. Konstatinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2–13, 1993.

[3] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *SIGCOMM '94*, pages 2–13, August 1994.

[4] Dave Dunning and Greg Regnier. The Virtual Interface Architecture. In *Hot Interconnects V*, pages 47–58, 1997.

[5] Robert W. Horst. TNet: A Reliable System Area Network. *IEEE Micro*, 15(1):37–45, February 1994.

[6] Kimberly A. Keeton, Thomas E. Anderson, and David A. Patterson. LogP Quantified: The Case for Low-Overhead Local Area Networks. In *Hot Interconnects III*, 1995.

[7] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[8] Shubhendu S. Mukherjee and Mark D. Hill. A Survey of User-Level Network Interfaces for System Area Networks. Technical Report 1340, Computer Sciences Department, University of Wisconsin–Madison, February 1997.

[9] Ioannis Schoinas and Mark D. Hill. Address Translation in Network Interfaces. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA)*, February 1998.

[10] Pete Vogt. Profusion: A Buffered, Cache Coherent Crossbar Switch. In *Hot Interconnects V*, pages 87–96, 1997.

[11] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 40–53, December 1995.

[12] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.