A Survey of User-Level Network Interfaces for System Area Networks

Shubhendu S. Mukherjee
Mark D. Hill

Technical Report #1340

February 1997

A Survey of User-Level Network Interfaces for System Area Networks

Shubhendu S. Mukherjee and Mark D. Hill

Computer Sciences Department
University of Wisconsin-Madison
Madison, Wisconsin 53706-1685 USA
{shubu,markhill}@cs.wisc.edu

Abstract

System Area Networks (SANs), such as Myricom Myrinet and IBM Vulcan, provide latency, bandwidth, and reliability that are orders of magnitude better than traditional local area networks. SAN benefits are, however, proffered to applications only if light-weight protocols (not TCP/IP) and efficient network interfaces are used. SAN benefits 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 directly access the network interface without compromising protection by memory mapping internal interface registers into user space (e.g., the Myricom Myrinet network interface).

Future trends in processor and SAN performance improvement indicate that ULNI accesses will become a critical bottleneck—*if not the most critical bottleneck*—in systems built around SANs. Although processor accesses to ULNI registers are simply reading and writing ULNI memory, almost all ULNIs today treat them as I/O operations that can have side effects (e.g., a message send). In this paper we argue that processor accesses to ULNI memory should be treated by design as side-effect-free regular memory accesses, and not as I/O operations. Such unconventional treatment of ULNIs will allow a processor to use traditional memory access optimization techniques such as caches, and novel memory access optimizations such as out-of-order accesses, speculative loads, lock-up free caches, split-transaction memory buses, etc. to reduce and tolerate the latency to access ULNI registers.

To substantiate our claim we examine several ULNI design options, including ULNI register location, dedicated vs. non-dedicated ULNI memory, coherent caching, memory bus alternatives, data movement alternatives, application programming interface, notification alternatives, data copies, protection and address translation, and future system alternatives (e.g., SMPs, speculative processors, etc.). For each design option we show how treating ULNI register access as regular memory operations can help improve ULNI performance.

1 Introduction

The increasing demand for powerful commercial servers has opened up a new market for high-bandwidth, low-latency networks. The advent of world-wide web search engines and database queries on terabytes of data, and the possibility of network computers with "thin" clients and powerful servers, promise to stress the communication subsystem between clients and high-end servers. Parallel computers are used increasingly as high-end servers because today's uniprocessor workstations cannot match the tremendous growth in computation, memory, and I/O requirements of these applications. Two new kinds of parallel computers have evolved—Symmetric Multiprocessors (or SMPs) and Cluster of SMPs (CSMPs). SMPs extend the traditional workstation architecture with multiple processors on a single memory bus and (optionally) with more memory and disks. The difficulty of scaling today's memory buses beyond 20-30 processors has inspired the growth of a second class of parallel computers called CSMPs in which multiple SMPs are connected together through a high-bandwidth, low-latency network.

The demand for high performance communication subsystems—to connect hundreds of clients and high-end servers and to build high-end servers from clusters of SMPs—cannot be met by commodity local area networks (LANs) that have traditionally connected geographically close PCs and workstations. Today's commodity LANs offer very high latency (100-1000s of microseconds) and relatively low bandwidth (1-10 megabytes/second). The poor performance of LANs has led to the evolution of a new generation of high performance networks called System Area Networks (SANs) [4]. Examples of SANs include the Myricom Myrinet [6] and the IBM Vulcan [16] switches. SANs deliver very low latencies (less than a few microseconds) and very high bandwidth (in the 100s of megabytes/second range).

SANs not only provide very high performance, but also have the potential to deliver this high performance to end-user applications. This enables user applications to efficiently communicate between themselves with fine-grain messages. Fine-grain messages arise both in request-response protocols (e.g, Remote Procedure Calls in the Net-

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, NSF Grants MIP-9225097 and MIPS-9625558, and donations from IBM and 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.

work File System) in a client-server environment and in parallel database queries and parallel scientific applications. Request-response protocols have fine-grain communication because requests are typically small. Empirical evidence [29] suggests that even a major fraction of the response traffic in applications such as the Network File System consists of small messages that are less than 200 bytes. Fine-grain communication is an important component of parallel applications such as parallel database queries and parallel scientific applications because communicated data is often generated at a fine granularity at run-time. In modern microprocessors, for example, a single store instruction can generate only between 1 - 16 bytes. Empirical evidence [30, 11] shows that a major fraction of messages in a parallel database application and several parallel scientific applications consists of small messages.

To allow user applications to communicate efficiently with fine-grain messages over SANs, the latency to access the SAN interface must be reduced significantly. This latency consists of three parts—the software protocol latency, the latency through the operating system, and the hardware latency to access (e.g., read data from) the SAN interface. LANs use very heavy-weight software protocols such as TCP/IP, which make the conservative assumption that LANs, like the internet, are highly unreliable. For SANs, such overly conservative protocols can be replaced with lean communication layers such as Active Messages [54, 24]. Typically, LAN interfaces deliver messages to user applications via the operating system, which can be very expensive. For SANs, the latency through the operating system can be eliminated by providing applications with direct user-level access to the network interface hardware. For example, the Thinking Machines' CM-5 allows protected, user-level access to the network interface hardware by memory-mapping the interface registers into user space. We call such interfaces *User-Level Network Interfaces (or ULNIs)*. Finally, the latency to access a SAN interface can be reduced with novel techniques such as User-Level DMA [5] or Coherent Network Interfaces [36].

Technology and application trends, however, indicate that the latency to access the SAN interface will become a critical bottleneck—*if not the most critical bottleneck*—in systems built around SANs. A network interface has two sides—an internal interface that talks to microprocessors, main memory, and (perhaps) disks within a host node and an external interface that talks to the network that connects different host nodes. Pressure on both the internal and external interfaces of a network interface is increasing at a tremendous pace.

The pressure on the internal interface is increasing because of two reasons. First, the rate at which applications are generating messages and, therefore, accessing the network interface is increasing rapidly. This is because processor performance is improving exponentially; the steady drop in feature sizes and the introduction of microarchitectural techniques such as out-of-order and speculative execution are projected to improve processor performance by a factor of 80 in the next ten years [23]. Second, multiple processors in an SMP node are now sharing the same network interface to send and receive messages from a SAN. This is increasing the pressure on the internal network interface. This has been made possible by the advent of coherent memory buses; multiple processors can now be easily connected together to the memory bus to form an SMP node.

The pressure at the external interface is a result of the tremendous improvement in network performance. The advent of optical fiber networks have provided low cost, very high bandwidth networks. Optical fiber network bandwidth has improved by more than a factor of 20 in the last 10 years [13] and is projected to improve by another factor of 20 in the next decade if this trend continues. End-to-end network latency is primarily determined by the network switch latencies, which has also dropped steadily over the last decade. Switches with latencies less than one microsecond are already available today. Switch latencies will continue to drop in future with the advent of low-cost CMOS processes, novel switch architectures [3], and high-speed optical switches [38].

Designing high performance internal network interfaces for fine-grain communication that arises in message-passing applications is singularly more challenging than designing the external network interface. Since network interfaces and network switches typically come from the same manufacturer, the manufacturer is free to change and optimize the external interface between the network interface and the network switch. However, to allow reuse of network interfaces across machines or across different generations of the same machine, the internal network interface must conform to a standard specification (e.g., workstation I/O bus specification). As illustrated by the DRAM market [41], squeezing performance from standard specifications across different generations of machines and technologies can be a very hard problem. In this paper we focus on design alternatives for high performance internal network interfaces for fine-grain communication.

There are two fundamental obstacles to improving the latency to access the internal network interface. First, although accesses to internal network interface registers is primarily reading and writing memory, they have tradi-

tionally been treated like processor-initiated disk I/O operations. For example, accesses to internal network interface registers are either routed through the operating system (similar to a file I/O operation) or performed through uncached loads and stores to device registers memory-mapped to user I/O space. Additionally, the network interface device has been relegated to the I/O bus, where I/O devices such as disks typically reside.

Second, accesses to internal network interface registers often have *side effects*; they can trigger special actions such as popping of a hardware fifo buffer or a message send that a processor cannot detect. These side effects, which are a result of the tight coupling between device memory access and device commands, prevent the new generation of microprocessors (such as PA-RISC, Intel Pentium Pro, MIPS R10000) from independently optimizing network interface register access performance through out-of-order accesses and speculative loads.

In this paper we argue that to improve accesses to internal network interface registers in high performance systems, we must treat accesses to these registers as regular memory accesses, not as I/O operations, and decouple these accesses from commands that trigger special actions in the network interface. Additionally, the device commands themselves, whenever possible, should be treated as memory operations (e.g., a device command to send a message can be accomplished simply by incrementing a pointer). Such treatment allows a processor to use traditional memory access optimization techniques such as caches that exploit temporal and spatial locality and novel memory access optimizations such as out-of-order accesses, speculative loads, lock-up free caches, split-transaction memory buses, etc. to reduce and tolerate accesses to network interface registers. Since the gap between processor and DRAM access performance is increasing [23], microprocessors will continue to invent novel techniques to bridge this gap. Treating network I/O as memory, and not as I/O, will allow internal network interface accesses to take advantage of such future innovations. Table 1 examines several myths about traditional network interface designs; most of these myths are a result of treating network I/O as I/O, and not as regular memory.

To substantiate our claim we examine the evolution of SANs (Section 2) and define the different components of *User-Level Networks Interfaces (or ULNIs)* for SANs (Section 3). Then we examine several factors—location (Section 4), dedicated vs. non-dedicated memory (Section 5), coherent caching (Section 6), memory bus alternatives (Section 7), data movement alternatives (Section 8), application programming interface (Section 9), notification alternatives (Section 10), data copies (Section 11), protection and address translation (Section 12), and future system alternatives (Section 13)—that influence the design of such network interfaces. For each factor we show how treating network I/O as memory can help improve the internal network interface performance.

2 Evolution of System Area Networks (SANs)

SANs fall somewhere in between standard LANs that offer good scalability and reusability across different computer systems and custom memory buses that offer low latency, high bandwidth, and high reliability. In this section we discuss how SANs have evolved to combine the benefits of memory buses and LANs.

The demand for high performance communication subsystems, which are used to connect client and powerful commercial servers and to build high-end servers from clusters of SMPs, cannot be met by commodity LANs. Shared media LANs such as Ethernet or FDDI offer very high latency (100 - 1000s of microseconds) and relatively low bandwidth (1-10 megabytes/second). The recent transition to 100 megabits/second Ethernet or switched LANs such as switched-Ethernet or ATM alleviates this situation only partially. These recent networks increase the bandwidth only somewhat and their latencies continue to be in the 100s of microseconds range.

The poor performance of LANs is aggravated by the huge latency to access a LAN interface from within a commodity workstation node. This latency is a result of three factors. First, legacy protocols such as TCP/IP and UDP/IP used to communicate over LANs make the conservative assumption that LANs, like the internet, are highly unreliable and therefore can drop, corrupt, replay, expose, forge, and arbitrarily delay network messages. These assumptions result in complex software protocol stacks that incur huge overheads at both the sending and receiving host nodes. For example, Clark, et al.'s [10] analysis of an optimized TCP/IP implementation shows that in the common case the TCP/IP protocol stack executes roughly 700 instructions just to process the TCP/IP protocol headers. Second, all message sends and receives are routed through the operating system, which can be very expensive. Anderson, et al., [2] found that jumping in and out of operating systems takes 100s of instructions even on modern microprocessors. Worse, operating system performance is improving at a much slower rate compared to the improvement in microprocessor performance [40]. Third, the LAN interface resides on the I/O bus because, unlike memory buses, I/O buses are standardized, which allows third party vendors to design the interface to a stan-

Myths	Reality
All network accesses must be routed through the operating system.	The operating system is required only to guarantee protected access to the network interface. The network interface and the operating system can be designed to avoid the operating system in the critical path of message sends and receives (Section 12).
Network interfaces must be optimized to handle large messages.	Network interfaces must also pay attention to small messages, which are very important for request-response protocols in client-server applications and parallel database queries, and in parallel scientific computations (Section 1).
The network interface must reside on the I/O bus.	Traditionally, network interfaces have resided on I/O buses because these buses are typically standardized; however, network interfaces can reside on memory buses if memory bus designers provide *personality interfaces* (Section 4).
Processor register file is the ideal place to map the network interface registers.	Although processor registers provide very fast access, they neither offer a standard interface (Section 4) nor provide plentiful buffering (Section 5). The former can drive up the cost, while the latter can degrade performance.
The network interface requires very few message buffers.	Network interfaces can require large amounts of buffer space (Section 5).
Accesses to the network interface must be uncached.	Network interfaces can be designed to allow both processors and network interfaces to cache network interface registers (Section 6).
Caching network interface registers cannot improve performance because network I/O does not have any locality.	Network interface register accesses have both temporal and spatial locality; hence, caching them can help improve performance. Additionally, memory buses are optimized for cache block transfers; hence, unlike uncached accesses, cached accesses to the network interface allows us to use the full transfer bandwidth of memory buses (Section 6).
Processes must use the underlying data movement primitives (e.g., uncached loads or stores, DMA) as the application programming interface (API) for the network interface.	The user API can be decoupled from the underlying data movement primitives; such decoupling allows independent optimizations of the user API and data movement primitives (Section 9).
Data copies must be avoided at all cost.	Carefully making multiple copies of data can actually improve performance (Section 11).
Network Interfaces cannot handle virtual addresses.	Network interfaces can be designed to directly deposit data into the user virtual space (Section 12).
Combining network interface access with network interface commands improves performance.	Combining network interface access with commands can actually hurt performance by preventing out-of-order and speculative accesses to the network interface registers (Section 13.2.1).
Processors cannot perform out-of-order and speculative accesses to network I/O space (i.e., the network interface registers).	Network interfaces can be designed to allow out-of-order and speculative accesses to network I/O space (Section 13.2.1).

TABLE 1. Some Myths About Network Interfaces

dard specification. Because of this restriction all accesses to a LAN interface must traverse the memory bus and the I/O bridge, which connects the memory and I/O buses together. Additionally, I/O buses are typically much slower (roughly a factor of four or more) compared to memory buses, which further slows down accesses to the LAN interface.

Memory bus interconnection technology is in striking contrast with LANs. Memory buses deliver extremely low latencies (10s of nanoseconds) and very high bandwidth (100 - 2500 megabytes/second). Memory buses can be accessed from processors in a few processor cycles because their high reliability and highly trusted environment avoid software intervention, their direct accessibility (e.g., through cache misses) avoids operating system intervention, and their proximity to the processor (only a few processor cycles away) avoids the overhead of I/O bridge and I/O bus traversals. But, unlike LANs, often memory buses are customized and have non-standard interfaces, and are hard to extend to hundreds of hosts spread across a room.

Ideally, we want a network that combines the best of memory buses and LANs. We want a network that combines the performance and reliability of a memory bus so that we can avoid running TCP/IP, but we also want the scalability and standardized interfaces of LANs, so that they can be reused across several generations of machines and/or can be manufactured by third party vendors. Thus, four goals have given rise to a new generation of networks called System Area Networks (SANs). These goals are:

- Performance (low latency and high bandwidth),
- Reliability,
- Scalability, and
- Reusability.

Gordon Bell [4] coined the name System Area Networks, but did not define their characteristics. In our view some MPP networks such as the TMC CM-5 network or the Meiko CS2 network can be classified as SANs. More recent examples of SANs are the Myricom Myrinet switch [6], the IBM Vulcan switch used in SP2 [16], the Spider switch used in the SGI/Cray Origin machine [19], the Cray T3E network [45], Nowatzyk, et al.'s S-Connect [37], Dolphin SCI switch [49], the Fujitsu AP-Net [47] and the Cray Gigaring [44]. Tandem's ServerNet [26] can also be classified as a SAN. However, the Tandem ServerNet is unique because it replaces the memory bus, the I/O bus, and the LAN with a single interconnection network called the ServerNet.

All these SAN switches deliver latencies less than a few microseconds and link bandwidth exceeding 80 megabytes/second. They are highly reliable---they do not drop network messages, provide CRC checks for error detection, and are expected to operate in a closed, secure, and trusted environment such as a business office or a machine room. Hence, errors are considered extremely rare; if the system does detect a SAN error (e.g., CRC error) it can either return the error status to the end-user application or crash the user application itself. These switches can be composed to build configurations that connect hundreds of host nodes, which makes them highly scalable (like switched LANs). Finally, they provide internal network interfaces that can be reused across different machines or across different generations of the same machine.

3 Components of a User-Level Network Interface

A network interface in a host node is a device that allows a processor to send and receive messages to and from a network that connects these host nodes. The network accepts messages from a network interface and delivers them to one or more network interfaces connected to the network. A network interface consists of two parts: the internal network interface and the external network interface. We define the internal network interface as the network interface's interface to the processor, main memory, and (perhaps) disks, and external network interface as the network interface's interface to the network (Figure 1). The internal network interface contains logic and memory that the processor uses to send and receive messages to and from the network interface. For example, an internal network interface may contain data registers that the processor writes messages to. An external network interface performs network-specific functions such as CRC calculations, network-specific framing, etc.

We define *User-Level Network Interface* (ULNIs) as the internal network interface that non-privileged user code can directly access to send and receive messages without any operating system intervention, at least in the common case. Traditionally, the operating system had taken responsibility to send and receive messages from user pro-

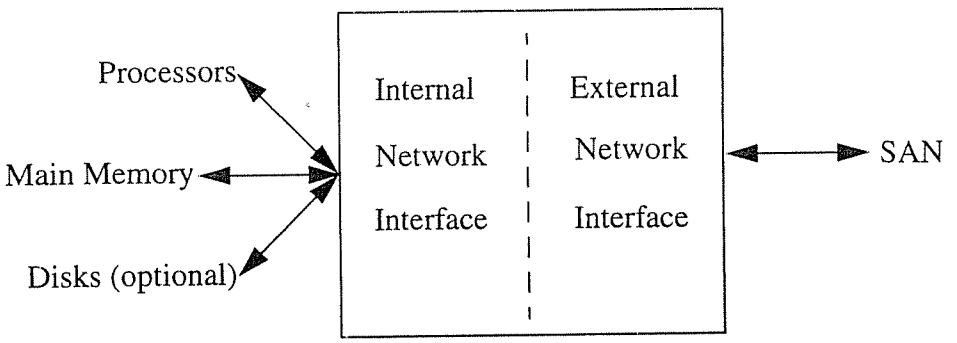

FIGURE 1. Internal and External Network Interfaces

cesses. For example, Unix users can send and receive messages through the socket interface. However, SANs such as the Myricom Myrinet allows direct, but protected, user access to the network interface by allowing users to memory-map the internal interface memory into the user's address space. The Myrinet host interface is, therefore, a ULNI. In this paper we limit ourselves to design alternatives for ULNIs for SANs.

A ULNI consists of two parts: the send interface or the send ULNI and the receive interface or the receive ULNI. Each interface consists of four components: status registers, control registers, data registers, and an optional notification mechanism. *Here we use the term registers just as an architectural specification*; the registers themselves may be implemented with DRAMs, for example. We will examine alternative implementations of these registers and notification mechanism later. In this section we discuss the function of each of the components.

Status Registers. ULNI status registers contain NI device status information. For example, a receive ULNI status register can indicate that a new message has arrived from the network and a send ULNI status register can indicate that the network interface has successfully injected a message into the network.

Control Registers. ULNI control registers allows a user process to pass information and commands to the NI device. For example, a processor may want NI interrupts disabled in a critical section. It can do so by writing to a control register in the ULNI.

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

Notification mechanism. A ULNI notification mechanism is a mechanism through which the ULNI informs the user process of any change in NI device status. For example, the ULNI can interrupt the user process on a change in device status such as arrival of a message from the network. Such explicit notification may be unnecessary if a user process monitors changes in the ULNI status registers. Hence the notification mechanism is optional.

To send a message to the network, a processor first reads the send ULNI status register to ensure there is enough space in the send ULNI 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 spin on the send ULNI status register, do something else and come back later to check the status register again, or totally avoid the ULNI until it is notified through an interrupt that the data registers are free and it can send another message. On receiving the new message in its data register, the ULNI hands the message to the external network interface, which injects the message into the network.

When a message arrives at the receiving external network interface, it extracts the message from the network and hands the message to the receive ULNI. The receive ULNI writes the message to its data registers, assuming there is enough space[1], and sets a status register that indicates to the processor that a message has arrived in the receive ULNI. Additionally, if the control registers have been appropriately set by the processor, the ULNI 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 ULNI data registers.

1. We can lose a network message if there is not enough space in the receive ULNI data registers. To avoid losing messages, the external network interface can use some form of flow control (e.g., return-to-sender [17]).

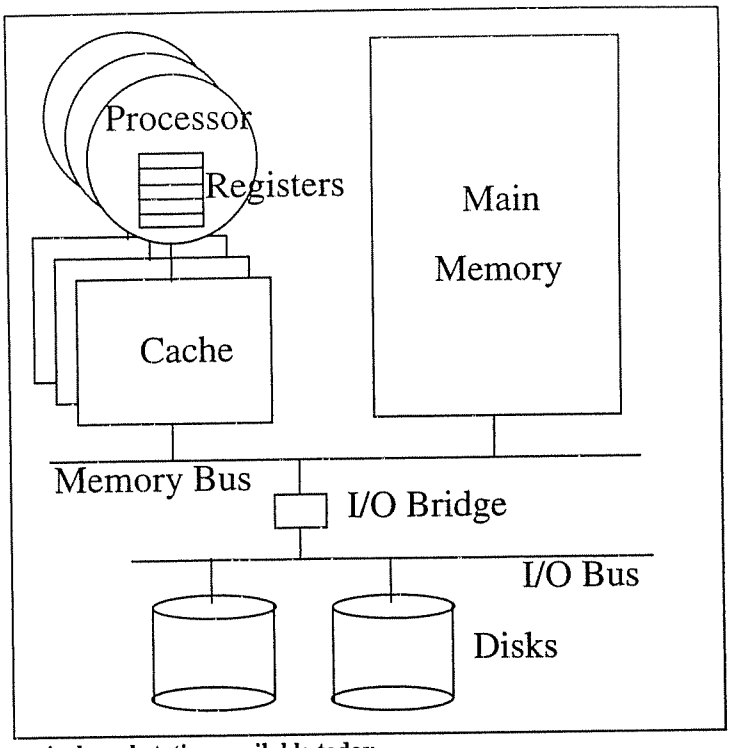

FIGURE 2. Architecture of a typical workstation available today.

4 Location of ULNI Registers

There are two design alternatives for ULNI register location. The ULNI registers (data, status, and control) can either be all located in the same place or they can be decoupled and placed in different levels of the memory hierarchy. Section 4.1 discusses the tradeoffs of ULNI register placement. Section 4.2 discusses alternative implementations of decoupled ULNI architectures.

4.1 Tradeoffs of ULNI Register Placement

In a standard host node such as a workstation or a personal computer the choice of ULNI location is dictated primarily by the tradeoff between the proximity of the ULNI to the processor and the availability of a standardized interface to which ULNIs can be designed. Figure 2 shows the architecture of a typical host node. In such a node, the ULNI can be located in several places—processor register-level, the cache bus,[1] the memory bus, or the I/O bus. Proximity of the ULNI to the processor improves the latency to access network interface registers because ULNI accesses travel shorter distances and tie up fewer resources along the access path. ULNIs mapped to processor registers and located on the I/O bus provide the two extremes of ULNI locations. Register-mapped ULNIs provide the fastest access because processor registers can be quickly accessed (in 5-10 nanoseconds), whereas I/O bus ULNIs have the highest access latency (typically around one microsecond). Accesses to a ULNI on the I/O bus additionally tie up the memory bus and I/O bridge, at least partially, during the duration of the accesses, which can degrade performance because the memory bus and I/O bridge are resources shared by main memory and possibly other processors in an SMP node. Memory bus and cache bus ULNIs provide intermediate points in the access latency spectrum.

Although placing the ULNI close to the processor can improve performance by reducing the access latency, the ULNI location is largely influenced by the availability of a standard interface so that third party vendors can design the ULNIs to the standard specification. Hence almost all third party vendors design the network interfaces to the I/O bus, which usually has a standard interface. Although memory buses are typically proprietary, the SPARC MBus

1. For processors with multi-level caches, one has the option of placing the ULNI at different cache bus levels.

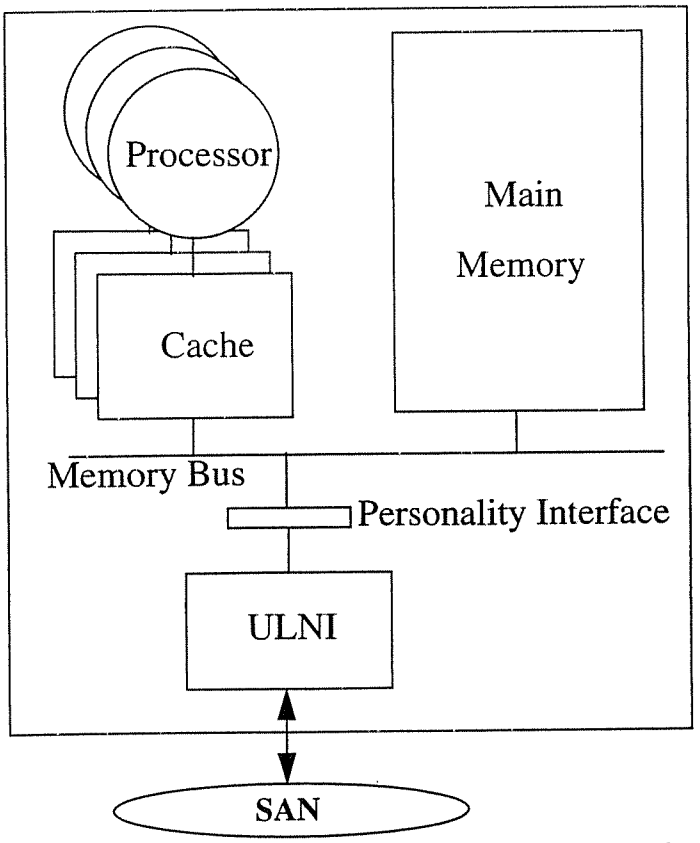

FIGURE 3. This figure shows how a personality interface connects a proprietary memory bus to a ULNI. The I/O bridge is a classic example of a personality interface.

had a standardized interface to which SPARC processors such as Cypress' SuperSPARC and Ross Technology's hyperSPARC were designed. This enabled MPPs such as the TMC CM-5 and Meiko CS2 to design their ULNIs to the SPARC MBus. Designing to the cache bus, although possible, and experimented with by research machines such as the MIT Alewife machine [1] and the MIT *T-NG [9] is not a feasible approach today because microprocessors neither expose their external cache bus interfaces to independent vendors nor promise to preserve that interface across different generations of processors. Processor register-mapped ULNIs are much harder to design because these ULNIs are tightly coupled with the microprocessor. A few research projects such as MIT J-machine [12] and the MIT M-Machine [17] have explored register-mapped ULNIs. Unfortunately, no microprocessor manufacturer have felt the need to provide a ULNI in their microprocessors because microprocessors are produced primarily for the uniprocessor PC market, and not for the multiprocessor market, and current microprocessors do not have a standardized I/O interface to the external world.

Since I/O buses are usually standardized, there is a strong motivation for independent vendors to design ULNIs to standard I/O buses. However, I/O bus performance has lacked behind memory bus performance by more than an order of magnitude. For example, the current generation of 32-bit PCI bus with 33 MHz clock can provide only a peak bandwidth of around 111 megabytes/second [23], whereas SUN's recent memory bus called the Ultra Gigaplane [48] provides a sustained bandwidth of 2.6 gigabytes/second. Clearly, ULNIs designed for I/O buses cannot harness the tremendous bandwidth offered by today's memory buses because the I/O bus itself is a potential bottleneck.

The key to ULNI placement is to find a location that can offer a standard interface and the datapath to which from the processor is not a bottleneck. Register-mapped and cache bus ULNIs are too tightly integrated to the microprocessor and do not offer a standard interface, whereas the I/O bus cannot yield adequate performance. The memory bus, therefore, appears to be the most viable candidate for the next generation of ULNIs. It provides very high bandwidth, particularly with the advent of split transactions, and relatively low latency. The memory bus performance will continue to improve in future to satisfy the memory bandwidth demands of memory-hungry out-of-order and speculative microprocessors.

Although current memory buses do not support a standard interface, there are several solutions to this problem. For companies that manufacture both microprocessors and networks for high performance systems (e.g., Intel, IBM), designing internal network interfaces to their memory bus may not be a problem. Intel's MPP supercomputer called Teraflop, for example, attaches the ULNI device directly on the PentiumPro memory bus [8]. For independent vendors finding a standard interface on the memory bus may imply coordinating with microprocessor companies to get access to their memory bus specification. Alternatively, manufacturers of proprietary memory buses could provide *personality interfaces* to other open standard interfaces such as the PCI interface.

Personality interfaces are interfaces that convert proprietary signals, in our case memory bus signals (Figure 3), into open standard specifications. A classic example of such an interface is the I/O bridge, which connects the memory bus to a standard I/O bus. The SGI Power Challenge [20] used personality interfaces to connect standard interfaces (e.g., the SCSI interface) to its proprietary split-transaction I/O bus. Personality interfaces offer two advantages for ULNIs. First, they decouple the memory bus from the ULNI device, which allows microprocessor designers to independently optimize the memory bus without worrying about the installed base of ULNI hardware. Second, they allow ULNI devices to connect to the memory bus without requiring device requests and responses to additionally traverse the I/O bus. Hence, a ULNI manufactured to a standard I/O bus specification, but attached to the memory bus through a personality interface, can offer significantly lower latencies compared to a ULNI sitting on the I/O bus.

Even with a personality interface, a ULNI manufactured to a standard I/O bus specification may have its latency and bandwidth limited by the clock rate and width of the I/O bus. In future this drawback of personality interfaces can be eliminated in three ways. First, ULNI devices can be designed to other higher performance specifications such as the Accelerated Graphics Port (AGP) [57], which extends the basic PCI architecture to offer higher bandwidth through a demultiplexed address bus, pipelined transfers, and 133 MHz effective transfer rate. In this case, we need a personality interface that converts memory bus signals to AGP signals.

Second, one can envision the emergence of a standard coherent memory bus specification, which would allow ULNI vendors to coherently cache network data in the ULNI devices. In the past the SPARC MBus [25] provided such a standard specification, at least for SPARC processors. However, the advent of major architectural changes such as split transactions made this standard obsolete. In the absence of any such major changes in the near future, its possible that such a standard will emerge again. A personality interface would still be useful because that would still allow microprocessor vendors to add little twists to their memory buses to improve the latency to main memory.

Third, its possible that memory buses may be replaced altogether with small high performance networks, which removes the critical timing requirement imposed by coherent memory buses. This shift makes it much easier to manufacture coherent ULNI devices to memory bus personality interfaces because the ULNI device, and hence, the personality interface, is no longer required to snoop on the memory bus within a critical timing interval. Current coherent memory buses (see Section 6.2) require caches to snoop on all memory bus addresses and intercept a cache block read request to main memory within a fixed time interval if they have the most recent copy of the cache block. However, unlike memory buses, devices on general-purpose networks cannot observe all such requests to main memory. Hence, with such networks coherent memory is usually implemented through hardware directory protocols [31, 55], which typically implement coherence through point-to-point messages, at least in the common case. The shift from a broadcast-based memory bus to point-to-point messaging with general-purpose networks will force designers to relax the timing constraint imposed by today's memory buses and make it easier for ULNI designers to manufacture ULNI devices.

4.2 Decoupled ULNIs

Decoupled ULNI architectures can also be a viable alternative in the future, particularly with the advent of high performance coherence protocols that provide a single image of memory locations within a host node. We define decoupled ULNI architectures as those that have their status, control, and data registers located in different places—processor register, processor cache, cache bus, memory bus, or I/O bus—of the host node (see Figure 2). For example, the Cornell U-Net [53] is an example of a decoupled ULNI architecture in which the data registers reside in main memory, while the status and control registers reside in the ULNI device on the memory or I/O bus (Figure 4). The combination of locations within a host node and ULNI registers gives rise to a variety of other

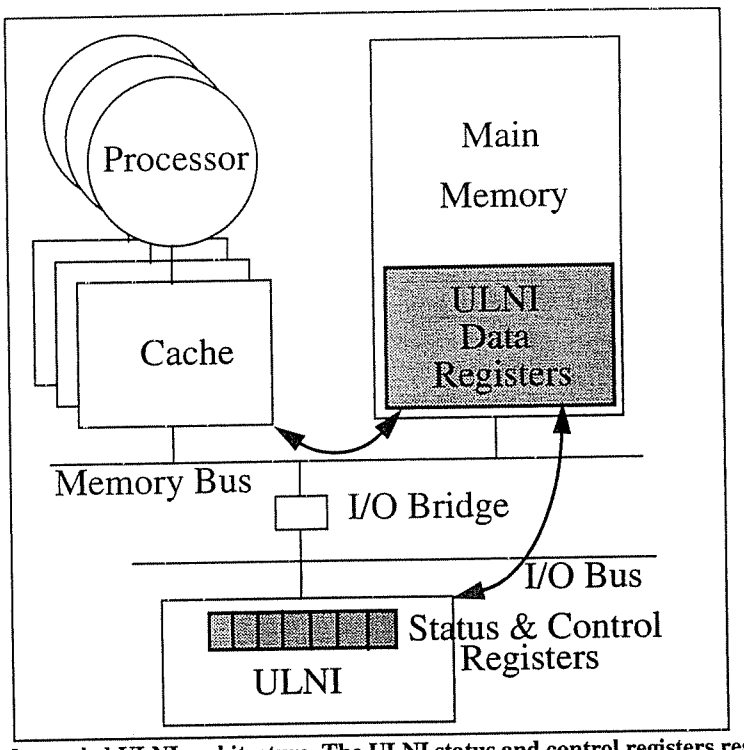

FIGURE 4. An example of a decoupled ULNI architecture. The ULNI status and control registers reside in the ULNI itself. However, the ULNI data registers from which a processor reads messages from or writes messages to are logically mapped to main memory. When a message comes in, the ULNI must write the message (e.g., via DMA) to the ULNI data registers in main memory. Similarly, when a message is ready to be sent, the ULNI must read the message from the main memory data registers. However, since main memory can be cached, the message can physically reside in the processor cache. The Cornell U-Net architecture is an example of such a decoupled ULNI architecture.

decoupled architectures. For example, Stodolsky, et al.'s [51] optimistic interrupt protection scheme offer a decoupled ULNI architecture in which the status and control registers are mapped to a fixed global register in a processor, but data registers can reside in the I/O bus ULNI device itself. When a message arrives at the ULNI device, it informs the operating system about the arrival of a message through an interrupt. The operating system then updates the status register or interrupts the user process based on the current control register information. The advantage of this scheme is that for small critical sections, in which a message rarely arrives, the user process can simply turn interrupts off and on by writing to a control register mapped to its global register. This avoids expensive uncached stores to the I/O bus ULNI control registers to turn interrupts off and on before and after the critical section.

The presence of a coherent memory allows greater flexibility in decoupled ULNI architectures. For example, if ULNIs DMA data into data registers residing in main memory, the corresponding memory locations in the processor caches must be invalidated. In the absence of coherent memory, this data must be invalidated from processor caches under explicit software control, which can slow down accesses to the ULNI data registers. Similarly, Stodolsky, et al.'s optimistic interrupt protection scheme can be implemented in a simpler manner if the control registers could be cached in the processor caches and shared by the ULNI device and the processors. In Section 6 we discuss how the presence of coherent memory allows ULNI registers to be cached. Caching of ULNI registers enhances performance by reducing bus traffic and the latency to access these registers. More aggressive coherence protocols such as an update protocol can further boost performance by allowing ULNI devices to directly inject its registers into processor caches.

5 Dedicated vs. Non-dedicated Memory

High performance ULNIs can require large amounts of memory (e.g., tens of megabytes) to buffer outgoing and incoming network messages. This requirement is because of four reasons. First, the wide variation in microprocessor and SAN switch performance and their growth rates, the advent of a wide variety of custom communication protocols, and the loose coupling of microprocessors and SANs, often manufactured independently by different

vendors, can temporarily create a huge mismatch between the rates at which network messages are generated, transferred, and consumed. Hence the absence of large amounts of message buffer space in the ULNI can create a serious performance bottleneck because both the network and processor must now block waiting for the other to clear and consume network messages.

Second, with limited buffering and large bursts of messages—a common occurrence in loosely synchronized parallel applications—a processor must constantly monitor ULNI status changes and remove messages from the limited ULNI buffers to avoid clogging up the network. Karamcheti and Chien [28] have shown that processor performance can degrade significantly if it is required to constantly monitor ULNI status in this fashion.

Third, since ULNIs provide direct access to the network without operating system intervention, it must be virtualized to allow multiple processes to access the device simultaneously. Limited amounts of buffering in the ULNI implies that either the degree of multiprogramming must be restricted or the ULNI buffers must be context-switched between multiple processes, which degrades performance.

Fourth, SAN interfaces to the external network interface are sometimes designed (e.g., Myricom Myrinet switch) to drop network messages if the external network interface refuses to accept messages for a certain amount of time. This decision, we believe, is motivated by the fact that the network is shared among multiple applications, and so backing up the network instead of dropping the messages can prevent other processes from making progress. To guarantee reliable delivery, however, the ULNIs must quickly buffer the network messages or use some form of flow control (such as all-to-all buffer reservations [35], return-to-sender [17]) to avoid buffer overflow in the ULNIs. In the absence of large amounts of buffering, such flow control strategies can significantly degrade performance.

Current ULNI interfaces for SANs typically provide around hundreds of kilobytes of message buffers in the ULNI. However, this does not provide enough buffering to support large systems with a large degree of multiprogramming. Thus, Fujitsu's AP3000 machine, for example, restricts the degree of multiprogramming of its ULNI to three (one system, two user). Alternatively, ULNIs could provide large memories to buffer messages; but this would drive up the cost of the ULNI.

The problem of limited buffering in the ULNIs can be solved by buffering network messages in the user's virtual space [33]. This provides huge amounts of buffering limited only by the size of main memory (and perhaps swap space), which backs up the user virtual space. However, using user virtual space to buffer messages poses two problems. First, the ULNI must have access to the process identifier and virtual-to-physical translations for the user virtual space. For example, the Cray T3E [46] and Mitsubishi DART [39] ULNIs have user space translations, which allows them to buffer messages in user virtual space. But, this also implies that ULNI must interact with the operating system to obtain the user space translations, which incurs additional complexity at the software and hardware level. The frequency of interaction with the operating system can, however, be minimized by caching user space translations in the ULNI (see Section 12) or exposing the page table structure to the ULNI device, which can then directly read user space translations from coherent memory space. Second, physical pages that back up the user virtual space must either be pinned in memory or the ULNI must have mechanisms to detect a missing physical page and take a page fault to bring the page back into main memory. Missing physical pages can be detected if the operating system provides an extra bit for each virtual-to-physical translation entry to indicate if the page is swapped out or not. The ULNI can interrupt the operating system to take the page fault on its behalf.

Large buffer requirement of ULNI devices suggests that processor registers may not be right location for the ULNI device. To allow very fast access (e.g., one cycle) to processor registers, the register file is usually made small in size (e.g., 256 bytes); such small register files can only buffer a few network messages. However, its possible for processors to buffer messages in user virtual space; but, this requires explicit processor intervention to copy the data from the processor registers to the user virtual space, which can significantly degrade performance. Additionally, current microprocessor vendors do not want to waste precious on-chip resource for network I/O. Thus, the unavailability of a standard I/O interface (Section 4) and the lack of plentiful buffering strongly suggest that processor registers may not be the appropriate place for the ULNI device.

6 Coherent Caching

Traditionally, the absence of a memory bus coherence protocol that provides a single image of memory locations and presence of side effects on ULNI register access prevented processors from caching ULNI registers. The absence of a coherence protocol on older memory buses required processors to flush their data cache under software control before and after a device DMA-ed data into main memory. Historically, this was known as *non-coherent I/O*. However, flushing the entire data cache degrades performance significantly by increasing cache misses. Alternatively, designers could mark DMA-ed device data as uncachable, which avoided loading device data into the processor's data cache. The advent of coherence protocols on almost all high performance memory buses removes this problem because the hardware automatically removes all stale copies of cache blocks in all processor caches when an I/O device DMAs data into main memory. This is known as *coherent I/O*.

The presence of side effects on device register access also forced designers to mark device register access uncachable because every access to device registers had to be made visible to the ULNI device. For example, ULNI devices that use FIFO buffers to send and receive messages typically pop the FIFO and change internal FIFO pointers when the last word of a network message is loaded from or stored to the ULNI device. This requires every processor access to the FIFO be visible to the ULNI device, and hence uncachable, so that the device can detect access to the last word of a network message. This prevents processors from caching ULNI device registers because cached loads and stores are not visible to a ULNI device. Such side effects, which are a result of the tight coupling of device memory access and device commands, are simply artifacts, and not features. These side effects can be easily separated by design. We will explore this in Section 9.

The advent of coherence protocols on almost all high performance memory buses today, allow processors and memory bus ULNIs to cache ULNI registers. In Section 6.1 we examine how caching ULNI registers help improve ULNI performance. Both processors and ULNIs can easily cache ULNI registers if the ULNI is located on the memory bus because these ULNIs directly observe the memory bus coherence protocol. I/O bus ULNIs, however, allow limited flexibility in caching ULNI registers (Section 6.2). This again argues that high performance ULNIs should be located on the memory bus.

6.1 Caching with Memory Bus ULNIs

Since memory bus ULNIs observe the coherence protocol directly, processors can cache ULNI registers just like they cache regular memory locations. Caching ULNI registers in processor caches offer three advantages [36]. First, caching status registers in processor caches helps remove unnecessary memory bus traffic. If a processor were polling on an uncached status register, every processor poll would go across the memory bus to the ULNI device. In the absence of any message in the ULNI, unsuccessful polls that do not find a message in the ULNI device waste precious memory bus bandwidth. Instead, if the processor polls on a cached memory location, which contains the ULNI status information, 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. With an update-based coherence protocol, even this cache miss can be eliminated because the ULNI device can directly update the status information in the processor's cache.

Second, uncached accesses transfer only a few bytes of data (e.g., between 1 - 16 bytes) and cannot always take advantage of the full transfer bandwidth of today's memory buses. For example, Sun Microsystems' Ultra Gigaplane has a width of 32 bytes, but uncached loads and stores on SPARC processors can load or store only upto 8 bytes at a time, which wastes three-quarters of the bus bandwidth on every access to the ULNI registers. Memory buses are, however, designed to speed cache block transfers. Since cache blocks are typically much larger (e.g., 32 - 128 bytes), they can exploit the full transfer bandwidth of memory buses. Caching ULNI data registers that contains network messages enables us to transfer messages in cache block units and makes accesses to ULNI data registers as cheap as cache misses. See Section 8 for more details on data movement alternatives between the processor and the ULNI device.

Third, frequent processor updates to control registers can proceed very fast if they are cached in the processor's data cache. For example, we can have a better implementation of Stodolsky, et al.'s [51] optimistic interrupt protection scheme (described briefly in Section 4) by using a cached memory location, instead of reserving a register

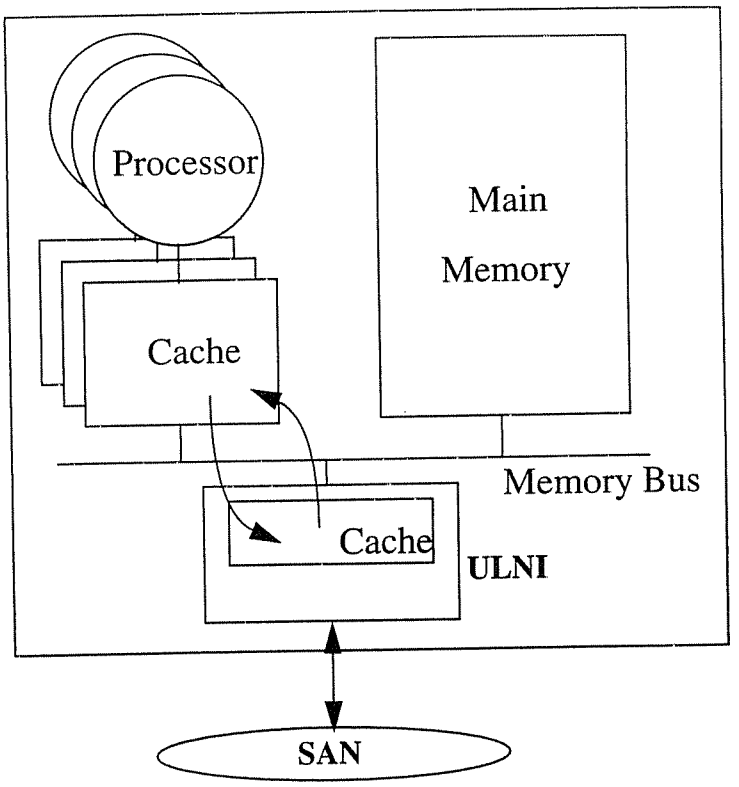

FIGURE 5. ULNI cache

from the limited global register pool, to hold the control information. In the absence of any message, the ULNI does not read the control information, which allows processor accesses to cached control registers to proceed at cache hit rates for write-allocate caches. Only when a message arrives, the ULNI reads the cached control register to decide if it should interrupt the processor or simply set the status register.

Caching ULNI registers in processor caches requires us to identify the *home* for ULNI registers. The home of a physical address is the I/O device or memory module that services requests to that address (when the address is not cached) and accepts the data on writebacks (e.g., due to cache replacements). Either dedicated memory in the ULNI device itself or main memory can serve as the home for the ULNI registers. As we discussed in Section 5, to make ULNIs cost-effective, only limited amounts of dedicated memory can be put on the ULNI. So, main memory, which is plentiful, is more attractive as the home for ULNI registers. Small amounts of ULNI memory can instead be used just like processor caches to cache the ULNI registers [36] (Figure 5).

ULNI caches help improve performance in three ways. First, processor cache misses for ULNI registers can be intercepted and satisfied directly by the ULNI cache through a fast cache-to-cache transfer. Second, when bursts of messages arrive at an ULNI, the ULNI cache may overflow; but, ULNI cache replacements to main memory will automatically buffer these messages without any processor intervention. Contrast this with cache bus or register-mapped ULNIs in which processors must explicitly copy the data from the ULNI registers to the user's virtual space because coherence is usually not an option on register-mapped or cache bus ULNIs. Third, if a data block is packaged in multiple messages and sent to different host nodes over the network (e.g., in an update protocol in a distributed shared memory machine), in the absence of a cache, the ULNI must fetch the data block for each message. However, with a cache, the ULNI has to fetch the block only the first time. Subsequent accesses to these data blocks will hit in the ULNI cache. Thus, ULNI caches help improve both the latency and bandwidth of ULNI accesses. It improves latency by directly transferring data through fast cache-to-cache transfers. It improves bandwidth through automatic message buffering in main memory and by reducing the number of accesses to message data blocks.

Home	Caching ULNI Registers In	Non-Coherent I/O	Coherent I/O	Coherent I/O + I/O Bridge Invalidation Support
Main Memory	ULNI Cache	No	No	No
	Processor Cache	Slow	Yes	Yes
ULNI	Processor Cache	Slow	Slow	Yes

TABLE 2. Caching I/O bus ULNI registers

6.2 Caching with I/O Bus ULNIs

Although coherent memory buses allow processors and ULNIs to coherently cache ULNI registers, the same may not be possible with standard I/O buses such as the SBus or the PCI bus (even with its coherent extensions) (Table 2). Two key mechanisms are necessary in the interconnect between the processor, the ULNI, and the home to coherently cache ULNI registers in a processor or ULNI cache. First, if a processor or ULNI cache contains the most recent copy of a ULNI register, it must be able to intercept a coherent read request for the register and prevent the home from responding. Second, a cache must be able to invalidate (or update) stale copies of ULNI registers residing in other caches.

The absence of the first mechanism—the ability to intercept coherent read requests for ULNI registers—makes it difficult for I/O bus ULNIs to coherently cache ULNI registers whose home is in main memory. Main memory resides on the memory bus, while the I/O bus is usually connected to the memory bus through an I/O bridge. Because I/O buses are usually slower than memory buses and the I/O bridge introduces additional delay on the memory bus to I/O bus path, it is difficult for I/O bus devices to intercept a memory bus coherent read request and prevent main memory from responding in a timely fashion. Memory bus ULNI caches, on the other hand, directly observe the memory bus coherence protocol and therefore can intercept a coherent read request, inhibit memory from responding (through the memory inhibit signal on the memory bus), and respond with the most recent copy of the cache block with the ULNI register.

Although I/O bus ULNIs cannot cache ULNI registers that reside in main memory, they could write messages directly to main memory, just like regular network interfaces that DMA messages to main memory. The memory bus coherence protocols ensures that all stale copies of the data written to main memory are invalidated or updated in the processor caches. Traditionally, this approach has been called coherent I/O.

The absence of the second mechanism—the ability to invalidate (or update) stale ULNI registers in other caches—in today's I/O buses makes it difficult for processor caches to coherently cache data from an I/O bus ULNI device when the home is in main memory. This is because when a ULNI updates main memory the processor cache can still contain stale data, which must somehow be invalidated (or updated). Although invalidation signals are absent in today's I/O buses (including PCI), such invalidations can be synthesized both at the software level or at the I/O bridge. At the software level, a processor could explicitly flush its entire cache (or the ULNI registers selectively, if selective invalidations are allowed) before a ULNI writes new ULNI register data to main memory. However, this solution—adopted in today's systems that support only non-coherent I/O—is slow because this requires a cache flush and an explicit handshake between a ULNI and processor before the ULNI can write new messages to main memory.

Alternatively, many systems support coherent I/O by adding functionality to the I/O bridge. When a I/O device writes data to main memory, the I/O bridge invalidates all stale copies of data residing in memory bus caches. The same mechanism could be used by ULNIs to allow processors to cache ULNI registers. However, this mechanism only allows main memory, and not ULNI memory, to be the home for cachable ULNI registers. Below we describe how we can relax this restriction and allow ULNI memory to serve as the home for cachable ULNI registers.

A third, and unconventional, alternative that we propose allows a ULNI device to serve as the home for cachable ULNI registers. In this method the I/O bridge fakes invalidation signals on the I/O bus using a technique called the *shadow address space*. The shadow address space technique has been used before to communicate special signals and address translations from a processor to an I/O device [5, 22], but not in an I/O bridge. In this technique, the I/

O bridge creates a shadow space for the regular I/O space by some invertible function such as flipping a bit. Thus, if 0xxx represents an I/O space physical address, 1xxx will represent its shadow physical address. Reads to 0xxx will proceed normally, but reads to 1xxx will be interpreted by the I/O device as a special control operation, which in our case is an invalidation on the physical address 0xxx. Thus, when the I/O bridge observes an invalidation signal for address 0xxx on the memory bus, it will convert it to a read signal on the address 1xxx. Conversely, when the I/O bridge observes an I/O bus read signal on 1xxx, it will convert it to a memory bus invalidation signal on the address 0xxx. These enable a ULNI device to observe all memory bus invalidation signals for ULNI registers and send invalidation signals for ULNI registers to memory bus caches.

7 Memory Bus Alternatives

Earlier we argued that it is critical to place the ULNI on the memory bus to take advantage of its high bandwidth and cache coherence protocol. In this section we examine some of the techniques memory buses use today and can use in future to improve their bandwidth (Section 7.1) and coherence protocols (Section 7.2) and how ULNIs can take advantage of these techniques.

7.1 High Bandwidth

Today's memory buses support very high bandwidth—between one to three gigabytes/second—through high clock rates, large bus widths, and split transactions. High clock rates improve latency and, hence, bandwidth. Many memory buses are clocked at 66 MHz or more. 32-byte wide memory buses are not uncommon today. Splitting a transaction into a request transaction and a reply transaction for each access allows multiple accesses to proceed simultaneously.

Memory buses are primarily targeted at improving cache block transfers and hence uncached accesses often cannot exploit the full transfer bandwidth of today's memory buses. For example, increasing the memory bus width to 32 bytes helps cache block transfers because cache blocks are usually between 32 to 128 bytes. Uncached loads or stores, however, operate on very small amounts of data, typically between one to eight bytes, and hence can only use less than a quarter of the memory bus bandwidth. A few processors (e.g., R10000) offer special mechanisms to combine consecutive uncached accesses, which allows these accesses to better use the memory bus bandwidth.

Even though split transactions allow multiple uncached accesses to proceed simultaneously, the number of outstanding transactions supported by a memory bus interface is limited. Since multiple transactions can be in flight, a memory bus interface must have a transaction identifier that pairs up the request transaction with the reply transaction. The number of transaction identifiers supported in the memory bus interface limits the number of outstanding transactions from that interface. With the same number of transaction identifiers, cache block transfers can fetch more data than can uncached loads because uncached loads carry very small amounts of data compared to cache blocks.

7.2 Cache Coherence Protocols

The cache coherence protocol supported by most memory buses today is a write-invalidation-based single-writer protocol. The protocol maintains a single image of all cached memory locations by invalidating all copies of a cache block in different processor or device (e.g., I/O bridge) caches whenever a new writer writes data to the cache block.

Future memory buses may also support more aggressive coherence protocols that can further boost the performance of ULNIs by allowing processors and devices to update other processor caches over the memory bus. These protocols come in three flavors—snarf, update, and push. Snarfing [21] is a technique in which a cache controller reads data in from the bus whenever it has a tag match (i.e., space already allocated) for a block in the invalid state. Snarfing can be implemented as an extension to standard invalidation-based coherence protocols. An update protocol goes a step further and updates a cache block whenever the block is present (i.e., has space allocated) in the processor's cache independent of the block's state in the cache. Finally, push forces a cache block into a cache independent of whether the block is allocated or not in a processor's cache. This may cause cache replacements; so this technique is harder and more complex to implement.

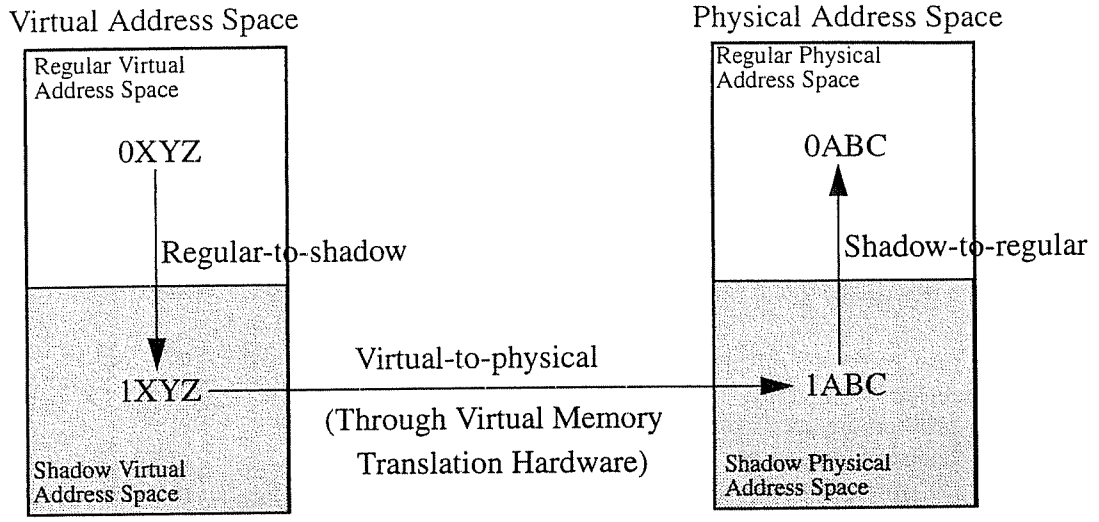

FIGURE 6. This figure shows how the shadow address space scheme in Princeton's UDMA mechanism [5] allows the UDMA device to obtain physical addresses for the DMA transfer. Both virtual and physical address spaces are divided into two regions—a regular space and a shadow space. For each address space there exists one-to-one mappings from the regular space to the shadow space. To initiate UDMA to a destination virtual address 0XYZ, the user process does a store to the corresponding shadow virtual address 1XYZ. We obtain the shadow mapping through a simple invertible function such as flipping a bit. The virtual memory hardware translates the shadow virtual address to the shadow physical address 1ABC, which the UDMA device observes on the bus. Finally, the UDMA device converts the shadow physical address back to the regular physical address by flipping the first bit and interprets the store to 1ABC as the user process' intention to initiate DMA to the destination physical address 0ABC. Thus, a user process delivers authentic physical addresses (in this case the translation from 0XYZ to 0ABC) using commodity virtual memory hardware to the UDMA device without invoking the operating system.

ULNIs can use these mechanisms to conveniently push data into a processor's cache. However, since a cache is limited in size and, therefore, a precious resource, the ULNI must carefully select which data should be pushed into the processor cache because pushing all incoming message data into the processor cache can cause cache pollution and waste memory bus bandwidth. A ULNI then has three choices. It can leave the data in the ULNI cache, if there is one; it can push the data into the processor cache; or, it can write the data to main memory. Careful analysis and experiments are needed to evaluate the potential for each of these approaches in the future.

8 Data Movement

Current systems offer three ways to move data between the processor and the ULNI. First, the processor can perform uncached loads or stores to memory-mapped device registers and directly load or store data to or from processor registers. Second, the processor can initiate a DMA request either through a system call or through a User-Level DMA request, which activates the DMA engine in the ULNI. The DMA engine transfers the message data from the ULNI registers to main memory. With coherent I/O—a common support in most systems today—the corresponding data, if any, in the processor cache is invalidated. Third, a processor can read data directly from the ULNI through cache misses. With more aggressive protocols that allow the ULNI to update the processor cache, even these cache misses can be eliminated.

As we discussed earlier, uncached loads or stores have very low bandwidth and is not a viable alternative without special processor support such as coalescing store buffers that can merge multiple uncached stores to device registers (e.g., R10000) or block moves to floating point registers (e.g., ultraSPARC). However, these mechanisms are highly processor-dependent and limited in scope and third party vendors cannot always rely on such support to design their ULNIs.

DMA is a viable alternative to uncached accesses for data movement, particularly with the advent of User-Level DMA. Traditional DMA allowed users to access the DMA engine of devices only through a system call because DMA engines expect physical addresses; but, operating systems cannot rely on user processes to provide correct physical addresses. User-Level DMA [5] solved this problem through a two-instruction sequence that allows DMA initiation from user space without a system call. The UDMA sequence looks as follows:

store <DMA transfer size> to shadow(<destination virtual address>)

load <status> from shadow(<source virtual address>)

Figure 6 shows how the shadow address space technique works for Princeton's UDMA scheme. Thus, this two-instruction sequence reduces the DMA initiation overhead significantly and makes DMA a powerful data movement mechanism even for small transfers.

Although DMA has traditionally been used to transfer large amounts of data between devices and main memory, on current systems that support coherent I/O, DMA is usually implemented through a sequence of coherent, cache block transfers. For NI devices on the I/O bus, the I/O bridge converts the non-coherent I/O bus block transfer signals to memory bus coherent signals. Memory bus NI devices can directly issue coherent signals to implement coherent DMA.

The third alternative is to use coherent, cache block transfers to directly read data from the ULNI. Like DMA, this scheme has the advantage of transferring data in cache block units. Unlike DMA, however, message data can be transferred directly from the ULNI to the processor cache without transferring the message to main memory and then to the processor cache. With large amounts of dedicated memory or a cache (with main memory serving as home) for messages in the ULNI, message data can be read directly into the processor cache from the ULNI memory or cache. With a ULNI cache, the processor reads the data from main memory only if the ULNI cache overflows and messages get automatically buffered in main memory. In the absence of memory on the ULNI, the ULNI must write messages directly to main memory, just like DMA, so this scheme requires two hops (ULNI to main memory and main memory to processor cache) to reach the processor cache. [1]

9 Application Programming Interface (API)

Computer systems have often directly exposed the underlying data movement primitives between the processor and the I/O device as the user's Application Programming Interface (API) to the I/O device. There exists two such user APIs—program-controlled I/O (PIO) and user-level direct memory access (UDMA). In PIO a processor reads from or writes to a device through uncached loads or stores. For UDMA a processor initiates the DMA engine in the I/O device, which interrupts the processor on completion of the DMA transfer. In both PIO and UDMA the underlying data movement mechanisms—uncached access and DMA—are directly exposed as the API.

Recently, we have seen the evolution of a number of different user-level APIs such as Arizona Application Device Channels (ADCs) [15], Cornell U-Net [53], Mitsubishi DART [39], and the Wisconsin Coherent Network Interfaces (CNIs) [36])—none of which can be classified as PIO- or DMA-based. All these APIs have simple memory-based queue semantics without any explicit notion of data movement. The sender enqueues network messages at the tail of the send queue and the receiver dequeues messages from the head of the receive queue. For message sends, the sender is a processor and the receiver is the ULNI. For message receipt, the sender is the ULNI and the receiver is a processor. Device commands for such APIs are no longer explicit DMA-initiation requests; instead, device commands simply increment or decrement the queue head or tail pointers. For example, when a processor writes a message to a queue and increments the tail pointer, the ULNI interprets this as a device command to send a message out to the network.

There are three key features of these new APIs. First, these APIs are *memory-based* queues. Sending or receiving messages involve simply reading and writing the queue memory and associated data structures (e.g., head and tail pointers). Both the ULNI and the processor must follow a fixed protocol to read and write data from these queues; otherwise, the message data may get corrupted or overwritten. The four APIs, however, differ in the location of this queue memory. For Arizona's implementation of ADCs both the send and receive queues reside in dedicated memory in the ULNI; for the Mitsubishi DART interface, the send queue resides in host memory, but the receive queue resides in dedicated memory in the ULNI; for Cornell's U-Net implementation both the queues reside in main memory; finally, for CNIs the home for both queues is in main memory, but they can also be cached in the ULNI cache.

Second, these APIs separate data movement from the user API itself. Sending or receiving messages is simply reading and writing queue memory; the user has no knowledge of how the data is actually moved between the pro-

1. Transferring messages from the ULNI to the processor cache may, however, be wasteful if the application simply intends to transfer data from the network interface to another I/O device (e.g., disk or graphics buffer).

cessor and the ULNI. The implementor can choose the data movement primitive best suited for a particular system. For example, both the implementations of ADCs and DART use uncached accesses for device commands. However, for message sends, the ULNIs use DMA to transfer data between host processor cache or main memory and the ULNI. Message reception involves two kinds of transfers—DMA to transfer data from the ULNI to main memory and coherent, cache block transfers from main memory to the host processor cache. The Cornell U-Net API was implemented on two different interface cards—Fore Systems' SBA-100 and SBA-200 interfaces. For the SBA-100 card, U-Net transfers both data and device commands between the processor and device through uncached accesses. The SBA-200 card supports DMA for data movement; so, the SBA-200 implementation of U-Net, like ADCs and DART, use uncached accesses for device commands and a combination of DMA and coherent, cache block transfers for data movement. Unlike the three other ULNIs, CNIs primarily use coherent, cache block transfers to move data between the processor and ULNI.

The decoupling of data movement from user API to the ULNI device provides three key advantages. First, such decoupling provides a portable interface to the ULNI. This portable interface is independent of the underlying data movement primitives. Second, the implementor can choose the optimal data movement primitives offered by a particular system to implement the user API. The two U-Net implementations show how data movement can be optimized depending on the capabilities of the ULNIs. In the future, systems may support aggressive update-based coherence protocols, which can offer new opportunities for data movement optimizations. With an update protocol a ULNI can push data directly into the processor cache on message reception. The ULNI is also free to dynamically decide where to place the data— in the ULNI, in the processor cache, or in main memory. Third, the API can itself be optimized independent of the underlying data movement primitives. For example, these queue-based APIs can use techniques such as lazy pointers, message valid bits, and sense reverse [36] to optimize accesses to the queues.

Third, for all these APIs, the ULNIs are aware of the structure of the queues. Contrast this with the Cray T3E queues [46] or Brewer, et al.'s Remote Queues [7] in which the queues provide a higher level programming abstraction and the underlying ULNI implementation may not necessarily be aware of the structure of the queues. Being aware of the structure of the queues has both its advantage and disadvantage. The advantage is that ULNI accesses can be optimized. For example, the Wisconsin Coherent Network Interfaces optimize accesses to the send queue using a variety of techniques mentioned above. The disadvantage is that the queue structure follows a fixed protocol and cannot be customized to an application's need.

10 Notification Alternatives

A user process must be notified when its ULNI status changes (e.g., when message arrives for that process). Two common notification strategies are: interrupt the user process when the ULNI status changes (e.g., in Unix, for example, this is done through the signal interface) or allow the user process to monitor changes in ULNI status by polling on a memory-mapped ULNI status register. However, notification through interrupts can be slow on today's processors and commodity operating systems; hence, many ULNI designs offer polling as an alternate notification strategy. In this section, we first examine why notification through interrupts is expensive in current systems and how this cost may be reduced. Finally, we examine how polling and hybrids of polling and interrupts may allow faster notification.

In today's processors notification through interrupts is very slow because these notifications must be vectored to the user process through the operating system, which executes hundreds of instructions before the interrupt is delivered to the user. For example, on a message arrival, the ULNI device activates one of the few hardware interrupt lines for the processor. This inserts a trap instruction into the processor pipeline. The trap instruction switches the user process out and puts the processor in the privileged operating system mode. The operating system saves the user process' state, figures out the cause of and handler for the interrupt, executes the handler, restores the user process' state, and switches back to the user process, which can now handle the interrupt. Switching back and forth between the user process and operating system also causes pollution in the processor's hardware structures and tables such as the instruction and data caches, TLB, branch prediction table, etc., which can severely degrade performance. Additionally, these interrupts may also force out-of-order and speculative processors (see Section 13.2.1) to stop their out-of-order and speculation engines, which can again affect performance. Modern processor architectures such as the SPARC Version 9 [50] have added special support (e.g., eight scratch registers that interrupt handlers can use, dirty bits for floating point registers, etc.) to somewhat reduce the overhead of an

interrupt. Nevertheless, vectoring an interrupt through the operating system is and will continue to be expensive because interrupts are treated both by the processor and operating system alike as an exception condition and not a common occurrence.

With adequate support from the processor and ULNI, it should be possible to vector an interrupt directly to a user process without any operating system intervention, at least in the common case. We outline one such method here. This method implements user-level interrupts with very two minor changes to current processors: the processor must support three extra registers and an extra hardware interrupt line. We require three extra registers to hold the current process' id, the virtual address of the currently executing process' interrupt handler routine [27], and the target process id of an arriving user-level interrupt. The extra interrupt line must be devoted for user-level interrupts. Now when a message arrives, the ULNI will extract the destination process id from the message (either directly or through a global to local id translation), ensure that the user-level interrupt line is not busy, write the target process id to the processor's third extra register, and activate the user-level interrupt line. When the processor sees an active user-level interrupt line, it will compare the target process id of the interrupt with the current process id. If both are same, the processor will save the target process' state in some user-accessible area, branch to the user-provided interrupt handler routine, and clear the user-level interrupt line. If the ids are different, then the processor can take the normal path and insert a trap into the processor pipeline. For the case when the user-level interrupt line is busy when a ULNI message arrives, the ULNI must wait until the user-level interrupt line is cleared by the processor.

Since modern processors do not yet support such user-level interrupts, system architectures rely on two other notification alternatives—polling and hybrids between polling and interrupt. In polling, the user process is required to periodically montor the state of the ULNI device, detect any status changes, and take appropriate actions. Polling can be much cheaper than interrupts because a user process can check the status of an ULNI device through uncached loads memory-mapped ULNI registers. However, polling can be harmful if the frequency is polling is much higher than the rate at which messages arrive, particularly if the processor must wait for the uncached loads from ULNI status register to complete. The cost of polling can, however, be reduced significantly if the processor caches the ULNI status register in its cache. In the absence of a message, the processor simply performs a cached load to the ULNI status register. The first access will miss, but subsequent accesses will hit in the processor's cache. If the ULNI status changes, the ULNI simply invalidates the ULNI status register in the processor's cache. When the processor tries to read the status again, it incurs a cache miss and fetches the status register. Even this cache miss can be eliminated if the system supports some form of update-based coherence protocol.

Alternatively, to remove the cost of polling in the absence of messages, many systems use a combination of interrupts and polling [23]. When a message arrives at the ULNI, the ULNI interrupts the user process, which starts polling for incoming messages. The user process resumes execution only when there is no pending message in the ULNI. This method tries to amortize the cost of an interrupt over several messages. Macquelin, et al.'s [34] polling watchdog further optimizes this method by not interrupting a user process when a message arrives; it allows the processor a certain time interval to poll for the incoming messages. If the user process fails to remove the message within this time interval, then the ULNI posts an interrupt.

11 Data Copies

Three types of data copies may exist within a host node both before message injection and after message reception: copies within a virtual address space, copies across multiple virtual address spaces, and copies across multiple physical locations. Data copies happen within a virtual address space either to copy data from/to user data structures to/from the ULNI queues or to buffer messages when the ULNI queues overflow. Data copies across multiple virtual address spaces happen when data injected or received from the network is passed to the user address space through a different address space (e.g., a server's address space or the operating system's virtual address space). Finally, since messages sent to or received from the network can be located in different places—processor cache, main memory, and ULNI memory/cache—the message data may be copied multiple times from one physical location to another. With a cache this replication happens without renaming. Without a cache, data must be explicitly copied between these physical locations.

Copying a message multiple times before injecting or after receiving it from the network has both its advantages and disadvantages. Multiple copies can incur undue overhead and slow down access to the ULNI, particularly if it

is the critical path of message sends and receives. However, there are two key advantages to copying data. First, explicitly copying data from one buffer to another frees the original buffer, which can be reused immediately. Without a copy the user of the original buffer must either be notified through an interrupt or the user must check some data structures to ensure that the buffer is free and can be reused. Second, when the fixed amount of ULNI buffers overflow, it may be worthwhile copying the data (either explicitly or implicitly through cache write backs from the ULNI) to main memory, which provides plentiful buffering and allows the ULNI to drain the network.

ULNIs must be designed carefully to avoid overhead due to multiple copies. The overhead due to copy data to/from user data structures can be avoided if the ULNI API allows a user to specify the virtual address of the data he/she desires to send to the network. The ULNI must then translate the virtual address into the corresponding physical address and fetch the data into the ULNI. The APIs with memory-based queues can indicate to the user that the buffer is free (and ready for reuse) by incrementing the head pointer of the queue. Additionally, the ULNI can generate an interrupt, if necessary. Data copies arising from ULNI queue overflow can be avoided, at least in the common case, by allowing large amounts of non-dedicated memory (e.g., main memory) to hold the ULNI queues and message buffers. However, to do this kind of buffering in the common case without any processor intervention requires the ULNI to have some form of address translation support, which we discuss in Section 12.

With ULNIs copies across multiple address spaces are mostly unnecessary because the user has direct access to the ULNI. Nevertheless, in situations where when the message data must pass through multiple address spaces (e.g., to ensure paranoid security constraints), instead of copying the data across address spaces, we could simply double map the virtual address into the two address spaces [14].

Finally, copies across multiple physical locations can be avoided by caching message data in the ULNI cache. Without a cache overflow, the data can be directly read from the ULNI cache to the processor cache. When the ULNI cache overflows—a situation that does not happen on the critical path of message sends and receives—the messages are automatically buffered in main memory without any processor intervention. Its also possible to design a system without a ULNI cache that provides the same ability—direct read from the ULNI memory and automatic buffering on memory overflow; but, since data is renamed due to the absence of a coherent memory, these copies must be explicitly managed.

12 Protection and Address Translation

Direct user access to a ULNI requires a ULNI to guarantee protection across different processes that are sharing the same network interface device. There are three ways to guarantee protection. First, we can disallow multiprogramming on the host node, which will ensure that only one process runs on the node. For host nodes connected with SANs, this may be infeasible because multiple requests may arrive simultaneously for multiple software servers. Second, like the TMC CM-5, we can tightly couple a user process and the ULNI and context-switch the ULNI along with the process. This will still allow only a single process to access the ULNI at any given time. This solution becomes infeasible with SMP nodes, where multiple processes may simultaneously access the ULNI. Third, we can map the ULNI memory into a user's virtual address space, which will allow the user protected access to the ULNI memory. If the ULNI supports small amounts of dedicated memory, then it is a scarce resource and must be allocated carefully among multiple processes. However, the constraint of limited memory is relaxed if main memory is used as ULNI message buffers because main memory provides plentiful buffering. Nevertheless, the range of user virtual addresses that a ULNI can access depends on the address translation scheme used in the ULNI.

There are two alternate address translation schemes that a ULNI can support. First, the ULNI can restrict the range of addresses in user virtual space that a user can access. This implies that a user must allocate all its communicated data structures (e.g., memory-based ULNI queues) in this address range. This may also restrict the user's ability to avoid multiple copies of the message data because the user may have to explicitly copy the message data from fixed address range to other user data structures. Second, the ULNI can have access to the entire user virtual address range, which can entirely avoid multiple copies of data within the user's virtual space. Such access requires the ULNI to support a full-blown address translation scheme. Several alternatives exist—from stashing the entire page table into the ULNI [46] to caching the translations in a ULNI data structures, either in software [22] or in hardware [39].

There are two problems associated with caching the translations in the ULNI: how to fill the ULNI translation buffer and how to avoid stale copies of translations when the operating system has remapped a page or swapped a

page to disk. The ULNI translation buffer can be filled in two ways. First, on a translation buffer miss, the ULNI can interrupt the operating system, which will insert the requested translation into the ULNI translation buffer. Second, translations can be inserted into the ULNI translation buffer directly by the user through the shadow address space technique [5, 22, 42] using simple uncached loads or stores to the ULNI device. The user loads/stores data from/to an alternate (or shadow) virtual address space. Accesses to the shadow virtual address space are translated into shadow physical addresses, which are observed by the ULNI device. The ULNI device knows the mapping between the regular and shadow spaces; hence, it can reconstruct the original physical address from the shadow physical address. The second solution's merit lies in the observation that the user knows the virtual addresses he/she is sending data from or receiving data to. However, the second solution is inferior to the first in two respects: it requires direct processor intervention even in the common case of message sends and receives and it uses uncached loads and/or stores, which prevent network I/O to be treated as regular memory accesses.

The translation table in the ULNI must be updated if the operating system either remaps a page or swaps a page out to disk. If the operating system decides to remap a page, it can simply invalidate the corresponding translation in the ULNI. The operating system may have to wait for an access is in progress from the ULNI on that particular page to complete. Page swaps can be handled in two ways. First, all physical pages may be pinned in main memory for the duration of the program. For long-running programs, this solution may be infeasible. Second, the operating system can again invalidate the translation in the ULNI translation buffer when it swaps a page out to disk. When the ULNI accesses the page again, it will have a ULNI translation buffer miss and request the host processor to remap the page. The host processor will swap the page back into physical memory and reinsert the translation into the ULNI translation buffer. [1]

13 Future System Alternatives

Three trends in processor and memory technologies and system designs are making it more and more important to treat network I/O as memory, and not as traditional I/O that is regarded as a low performance subsystem of a computer node. These two trends are the advent of SMP nodes and the increasing gap between processor and DRAM performance. In the following two subsections, we discuss the impact of each of these trends on ULNI design.

13.1 SMPs

Unlike the previous generation of machines, the current generation of workstations are SMP nodes with two or more processors connected to the memory bus. SMP nodes complicate the ULNI design in two ways. First, we must now allow several processes to access the ULNI device simultaneously. This makes multiprogramming the ULNI device much harder. Because multiple processes can access the ULNI device simultaneously, simple solutions such as the one adopted by TMC CM-5 in which the ULNI and user process are context switched together are no longer feasible. Worse, the ULNI device has no apriori knowledge of how many processes can simultaneously access the device. Hence, it must be prepared to allow some large number of processes to simultaneously access the ULNI device.

An elegant way to allow protected access to multiple processes to the ULNI device is to allow each process to open its own communication channel with the ULNI device. Such communication channels can be implemented with memory-based ULNI queues mapped to user virtual address space [53, 15]. Thus, each process will have its own send and receive queues to the ULNI. These queues introduce two additional complexities. First, the ULNI device must multiplex the internal and external network interface ports among these queues. Second, since ULNIs can only support limited amount of memory, ULNIs must be prepared to context-switch the ULNI state for the queues and queue data structures if they reside in dedicated ULNI memory. However, if the ULNI memory is treated as a cache, then ULNIs do not have to explicitly manage these queues and context-switch them because the queues will automatically get displaced to main memory if the ULNI cache overflows.

Second, multiple processes accessing the ULNI device simultaneously can create severe contention on the memory bus. To reduce this contention a processor's access to the ULNI device must be minimized. This can be achieved in

1. For more detailed discussions on protection and address translation issues in the network interface, please see Heinlein, et al. [22], Blumrich, et al. [5], and Schoinas and Hill [43].

current systems by caching the ULNI queues in the processor caches. As discussed in Section 6, caching helps reduce contention in several ways. When the ULNI status does not change, processor accesses to cached ULNI status and control registers will be direct cache hits in the processor's cache, which avoids accesses to the memory bus. Caching ULNI queues allows us to transfer data in cache block units, which can transfer significantly larger amounts of data compared to uncached accesses. Also, by collocating the status register and the first part of a message in the same cache block, the first part of a message can be directly transferred from the ULNI to the processor cache, when the processor incurs a cache miss for the cached status register. Additionally, future update protocols (see Section 7) should allow a ULNI to directly update processor caches, which can further reduce the traffic, and hence the contention, on the memory bus.

13.2 Bridging the gap between processor and DRAM performance

Processor performance is improving at 55% per year, but access latencies of DRAM, which is typically used as main memory, is improving only at 7% per year [23]. This huge gap in the performance of processors and DRAM have forced microprocessor designers to innovate new techniques to tolerate and reduce the huge latency to access main memory. Two approaches have gained importance: out-of-order accesses and speculative execution, and integration of microprocessor and DRAMs on the same chip. We discuss both of these in the following subsections.

13.2.1 Out-of-Order (OOO) Accesses and Speculative Execution

To tolerate the latency of main memory access, processors allow loads and stores to bypass earlier loads or stores. This is called out-of-order memory accesses. Four key system changes have taken place to realize out-of-order accesses to main memory: processors can issue out-of-order loads and stores, caches are non-blocking so that they do not stall on consecutive cache misses; memory buses can support several outstanding requests to main memory; and finally, the memory controller can handle multiple requests to the memory system.

Speculative execution is more aggressive than OOO accesses in tolerating memory access latency. Processors speculate on control dependence (e.g., branch prediction), data dependence, and data values (e.g., value prediction [32]) 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 speculative accesses must be squashed and any process-specific state must be rolled back to the point from where the speculation failed.

Current microprocessor designs typically disallow both OOO and speculative accesses to the ULNI. This limitation is a result of two factors. First, several ULNI devices expect message buffer access and/or ULNI device commands to arrive in order, which prevents both OOO and speculative accesses to the ULNI device. In the TMC CM-5, for example, every word (or double word) of a message must be sent and received in order. User-Level DMA relaxes this restriction on the message buffer. It allows messages to be directly written to main memory from which processors can perform both OOO and speculative accesses. However, the UDMA device expects a two-instruction store/load sequence that must occur in order. This prevents OOO and speculative accesses across multiple messages.

Second, ULNI devices often have side effects that also prevent OOO and speculative accesses to the ULNI registers. Side effects are manifested in two forms. First, message buffer access and device commands often offer a single point of access. A load or store to such a single address has the side effect that the data or command is committed to the ULNI and is no longer be visible to the processor, and hence cannot be unrolled if the processor so desires. For example, in the CM-5, message sends (or receives) require uncached stores (or loads) to a fixed memory-mapped ULNI register. Similarly, the UDMA device command sequence is accessed through a single point for multiple message sends. Second, device memory access is often overloaded with device commands. Such overloading has the side effect that a device memory access automatically triggers a special action in the ULNI device, which is not visible and reversible by the processor. For example, uncached load of the last word of a network message in the CM-5 has the side effect of popping the hardware FIFO that contains the network message. However, even if a ULNI exposed the message buffers, performing the last uncached load first—either through OOO or speculative access—would pop the hardware FIFO and the earlier words of the message would be lost forever.

ULNIs can be designed to allow OOO and speculative accesses. This requires two key design guidelines. First, we must decouple ULNI message buffer accesses from ULNI commands and treat ULNI message buffer accesses as regular memory accesses. Like DMA or UDMA, this method allows processors to perform OOO and speculative accesses to ULNI message buffers, just like regular memory space because regular memory space accesses need neither be in order nor cause side effects.

Second, ULNI device commands themselves, whenever possible, should be treated as regular memory accesses as well. For example, instead of having a single status register, which a processor reads through uncached loads to determine the presence of a message, message buffers can be exposed to the processor as message queues (Section 9) and each queue location can be augmented with a status bit (or bits) to indicate the presence or absence of a message. Exposing the message buffers in this way allows OOO and speculative accesses because device commands such as status register loads for multiple messages no longer need to be in order and messages are not accessed through a single point of access. Also, loading a status register no longer triggers any special action in the ULNI device; instead, messages buffers are explicitly freed by incrementing the queue head pointer (see Section 9).

Its possible to envision that in future ULNIs will send and receive messages speculatively. This speculative message send and receive would, however, require four changes to the processor and memory bus designs. First, processors must expose speculative stores either to the regular memory space or at least to the ULNI device. Current microprocessors do not expose speculative stores outside the processor because its hard to unroll regular loads to speculatively stored memory. However, designs such as the address resolution buffer [18] can relax this constraint and expose speculative stores to regular memory space and hence to the ULNI device. Second, ULNIs should have the ability to mark a message speculative when it sends it to another ULNI device over the network. Additionally, if the message is squashed due to an incorrect speculation, the ULNI device must be able to send an *anti-message* to the original receiving ULNI of the message. Third, ULNIs must have the ability to squash an entire sequence of computations when it receives an anti-message. Support for this is already in-built in modern microprocessors. For example, in the R10000 processor, an invalidation to a speculatively loaded word would squash the entire computation following the speculative load. Fourth, processors should have the ability to distinguish between loads that are already committed and loads that have been speculatively given to the processor. Support for speculative stores should already ensure this capability. This would allow ULNIs to speculatively offer a message to the processor.

13.2.2 Processor-DRAM Integration

The concept of processor-DRAM integration attempts to directly bridge the performance gap between the processor and DRAM; instead of tolerating the memory access latency with novel techniques such as OOO and speculative accesses, it tries to reduce it by putting both the processor and DRAM on the same chip. Putting the processor and DRAM on the same chip removes several chip boundary and interface crossings, and hence reduces the latency of accessing main memory.

There are two ways to integrate processors and DRAMs. Traditionally, processors and DRAMs have used separate technologies; so, either DRAMs could be put on a processor chip or processors could be put on a DRAM chip. However, processor and DRAM technologies are optimized for different purposes—processor technology is optimized for speed while DRAM technology is optimized for density. Hence, DRAMs on processor chip would have a powerful processor core, but reduced amount of memory; processors on a DRAM chip, on the other hand, will have plenty of memory, but a less powerful processor than available today. We already have processors chips with plenty of SRAMs (Static RAMs) that are typically used as caches. Supplementing SRAMs with DRAMs may just be an evolutionary step. Some DRAM manufacturers such as Mitsubishi [52] have taken the second approach. They have integrated processor logic on a DRAM chip.

We believe that with both approaches designers should still treat ULNI register access as regular memory accesses. The first approach is evolutionary; main memory migrates to the processor chip, instead of residing off-chip. The system architecture, however, remains the same; so, all our previous discussions and assumptions hold.

Even with the second approach, we believe that designers should treat ULNI register access as regular memory access because several of our earlier assumptions for non-integrated systems still hold. ULNIs will still require plentiful buffering because of the mismatch between the performance of processors and SANs, the presence of wide variety of protocols and bursty message traffic patterns, the requirement for multiprogramming, and manage-

ment of ULNI buffers through some form of flow control (see Section 5). Main memory can easily provide plentiful buffering at a reasonable cost. Memory-based queues as the API to the ULNI allows the decoupling and independent optimizations of data movement and the API itself. Directly depositing data into user memory (Section 11) avoids unnecessary data copies. Additionally, this approach makes it less important to map ULNI registers on processor registers because main memory accesses do not cross chip boundaries, at least in the common case, and provide higher bandwidth since DRAM interfaces do not necessarily have to be standardized internally to the processor.

Although current proposals for the second approach advocate a single processor per DRAM chip, to better utilize memory [56], its possible that in the long run the DRAM banks will be distributed among multiple on-chip processors, which will be connected through a coherent memory bus or network. In other words, we will have an on-chip SMP. In this case, we should be able to cache ULNI registers in processor caches and treat the ULNI memory as a cache with the DRAM banks serving as the home for the ULNI queues.

14 Conclusions

To satisfy the increasing demand for high performance networks a new generation of networks called System Area Networks (SANs), such as the Myricom Myrinet and IBM Vulcan, have evolved. SANs offer latency, bandwidth, and reliability that are orders of magnitude better than traditional LANs; but, like LANs, they are highly scalable, and their host interfaces are reusable across different computer systems. However, SAN benefits are proferred to applications only if light-weight protocols (not TCP/IP) and efficient network interfaces are used. SAN benefits 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 directly access the network interface without compromising protection by memory mapping internal interface registers into user space (e.g., the Myricom Myrinet network interface).

Future trends, such as the exponential improvement in microprocessor and SAN performance and the advent of SMPs, indicate that processor accesses to ULNIs will become a critical bottleneck—if the not the most critical bottleneck—for computer systems built with SANs. Processor accesses to ULNI registers is simply reading and writing ULNI memory. However, most ULNIs today treat such accesses as I/O operations that can have side effects (e.g., a message send). Such treatment disallows current ULNIs to take advantage of traditional memory optimization techniques such as caches, and novel memory access optimization techniques such as out-of-order accesses, speculative loads, lock-up free caches, split-transaction memory buses, etc.

In this paper we argued that to improve processor accesses to ULNI registers, ULNI memory accesses must be treated as regular side-effect-free memory accesses, and not as I/O operations; this allows processors to hide and tolerate the latency to access ULNI registers through traditional and novel memory access optimization techniques. To substantiate our claim we examined several ULNI design options, including ULNI register location, dedicated vs. non-dedicated ULNI memory, coherent caching, memory bus alternatives, data movement alternatives, application programming interface, notification alternatives, data copies, protection and address translation, and future system alternatives (e.g., SMPs, speculative processors, etc.). For each design option we showed how ULNI performance can be improved by treating ULNI accesses as regular memory operations.

Acknowledgments

We would like to thank Yannis Schoinas, Toshi Shimizu, and T. N. Vijaykumar for their very helpful comments on different drafts of this article.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 108–120, April 1991.

[3] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles R. Thacker. High Speed Switch Scheduling for Local Area Networks. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 98–108, 1992.

[4] Gordon Bell. 1995 Observations on Supercomputing Alternatives: Did the MPP Bandwagon Lead to a Cul-de-Sac? *Communications of the ACM*, 39(3):11–15, March 1996.

[5] 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.

[6] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[7] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John Kubiatowicz. Remote Queues: Exposing Message Queues or Optimization and Atomicity. In *Proceedings of the Sixth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 42–53, 1995.

[8] Joseph Carbonaro and Frank Verhoorn. Cavallino: The Teraflops Router and NIC. In *Hot Interconnects IV*, pages 157–160, 1996.

[9] Derek Chiou, Boon S. Ang, Arvind, Michael J. Becherle, Andy Boughton, Robert Greiner, James E. Hicks, and James C. Hoe. StartT-ng: Delivering Seamless Parallel Computing. In *Proceedings of EURO-PAR '95*, Stockholm, Sweden, 1995.

[10] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.

[11] 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.

[12] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Nuth, Scott Wills, Paul Carrick, and Greg Flyer. The J-Machine: A Fine-Grain Concurrent Computer. In G. X. Ritter, editor, *Proc. Information Processing 89*. Elsevier North-Holland, Inc., 1989.

[13] Martin de Prycker. *Asynchronous Transfer Mode: Solution for Broadband ISDN, Second Edition*. Ellis Horwood Publishers, 1993.

[14] Peter Druschel and Larry L. Peterson. Fbufs: A High-Bandwidth Corss-Domain Transfer Facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 189–202, 1993.

[15] 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.

[16] Craig Stunkel et al. The SP2 High-Performance Switch. *IBM System Journal*, 34(2), 1995. to appear.

[17] Marco Fillo, Stephen W. Kekler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. Technical Memo A.I. Memo No. 1532, MIT, March 1995.

[18] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Memory Disambiguation. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[19] Mike Galles. The SGI Spider Chip. In *Hot Interconnects IV*, pages 141–146, 1996.

[20] Mike Galles and Eric Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, 1994.

[21] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, 1988.

[22] John Heinlein, Kourosh Gharachorloo, Scott A. Dresser, and Anoop Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 38–50, 1994.

[23] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[24] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[25] Sun Microsystems Inc. *SPARC MBus Interface Specification*, April 1991.

[26] Tandem Computers Inc. ServerNet Interconnect Technology. Available from http://www.tandem.com/INFOCTR/PROD_DES/SRVNETPD/SRVNETPD.HTM.

[27] Douglas Johnson. Trap Architectures for Lisp Systems. In *1990 ACM Conference on Lisp and Functional Programming*, pages 79–86, 1990.

[28] Vijay Karamcheti and Andrew A. Chien. A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 298–307, 1995.

[29] Jonathan Kay and Joesph Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *SIGCOMM93*, pages 259 – 268, 1993.

[30] 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.

[31] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[32] Mikko H. Lipasti, Christopher B. Wilkeson, and John Paul Shen. Value Locality and Load Value Prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 138–149, 1996.

[33] Kenneth Mackenzie, John Kubiatowicz, Anant Agarwal, and Frans Kaashoek. Fugu: Implementing Translation and Protection in a Multiuser, Multimodel Multiprocessor. Technical Memo MIT/LCS/TM-503, MIT Laboratory for Computer Science, October 1994.

[34] Olivier Maquelin, Guang R. Gao, Herber H. J. Hum, Kevin Theobald, and Xinmin Tian. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 179–188, 1996.

[35] Richard Martin. HPAM: An Active Message Layer for a Network of HP Workstations. In *Hot Interconnects II*, 1994.

[36] 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.

[37] Andreas G. Nowatzyk, Miachael C. Browne, Edmund J. Kelly, and Miachael Parkin. S-Connect: from Networks of Workstations to Supercomputer Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 71–82, 1995.

[38] Andreas G. Nowatzyk and Paul R. Prucnal. Are Crossbars Realy Dead? The Case for Optical Multiprocessor Interconnect Systems. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 106–115, 1995.

[39] Randy Osborne, Qin Zheng, John Howard, Ross Casley, and Doug Hahn. DART - A Low Overhead ATM Network Interface Chip. In *Hot Interconnects*, 1996.

[40] John K. Osterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *USENIX Summer Conference*, June 1990.

[41] Steven A. Przybylski. *New DRAM Technologies: A Comprehensive Analysis of the New Architectures*. MicroDesign Resources, 1994.

[42] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[43] Ioannis Schoinas and Mark D. Hill. Address Translation in Network Interfaces. Unpublished Manuscript.

[44] Steve Scott. The SCX channel: A new, supercomputer-class system interconnect. In *Hot Interconnects III*, 1995.

[45] Steve Scott and Gregory M. Thorson. The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus. In *Hot Interconnects IV*, pages 147–156, 1997.

[46] Steve L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 26–36, 1996.

[47] O. Shiraki, M. Nagatsuka, T. Horie, Y. Koyanagi, T. Shimizu, and H. Ishihata. AP-Net Advanced High-Performance Network for Scalable Parallel Server. In *Hot Interconnects IV*, 1996.

[48] Ashok Singhal, David Broniarczyk, Fred Ceraukis, Jeff Price, Leo Yuan, Chris Cheng, Drew Doblar, Steve Fosth, Nalini Agarwal, Kenneth Harvey, Erik Hagersten, and Bjorn Liencres. Gigaplane (TM): A High Performance Bus for Large SMPs. In *Hot Interconnects IV*, pages 41–52, 1996.

[49] Dolphin Interconnect Solutions. Dolphin SCI Switches. Available from http://www.dolphinics.no/Products/Switches.html.

[50] Inc. SPARC International. The SPARC Architecture Manual (Version 9), 1994.

[51] Daniel Stodolsky, J. Brad Chen, and Brian Bershad. Fast Interrupt Priority Management in Operating Systems. In *Second USENIX Symposium on Microkernels and Other Kernel Archtitectures*, pages 105–110, September 1993. San Diego, CA.

[52] Jim Turley. Mitsubishi Mixes Microprocessor, Memory. *Microprocessor Report*, 10(7):10–12, May 1996.

[53] 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.

[54] 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.

[55] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993. Also appeared in it CMG Transactions,/ Spring 1994.

[56] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, February 1995.

[57] Yong Yao. AGP Speeds 3D Graphics. *Microprocessor Report*, 10(8):11–15, June 17 1996.