SPINE: A Safe Programmable and Integrated Network Environment

Marc E. Fiuczynski† Richard P. Martin‡ Tsutomu Owa* Brian N. Bershad†

† Department of Computer Science and Engineering, University of Washington

‡ Department of Electrical Engineering and Computer Science, University of California at Berkeley

* Toshiba, Japan. Visiting Researcher at Department of Computer Science and Engineering, University of Washington

Abstract: The emergence of fast, cheap embedded processors presents the opportunity to execute code directly on the network interface. We are developing an extensible execution environment, called SPINE, that enables applications to compute directly on the network interface This structure allows network-oriented applications to communicate with other applications executing on the host CPU, peer devices, and remote nodes with low latency and high efficiency.

1 Introduction

Many I/O intensive applications such as multimedia client, file servers, host based IP routers often move large amounts of data between devices, and therefore place high I/O demands on both the host operating system and the underlying I/O subsystem. Although technology trends point to continued increases in link bandwidth, processor speed, and disk capacity the lagging performance improvements and scalability of I/O busses is increasingly becoming apparent for I/O intensive applications. This performance gap exists because recent improvements in workstation performance have not been balanced by similar improvements in I/O performance. The exponential growth of processor speed relative to the rest of the I/O system, though, presents the opportunity for application-specific processing to occur directly on intelligent I/O devices. Several network interface cards, such as the Myricom's LANai, Alteon's ACEnic, and I_2O systems, provide the infrastructure to compute on the device itself.

With the technology trend of cheap, fast embedded processors (e.g., StrongARM, PowerPC, MIPS) used by intelligent network interface cards, the challenge is not so much in the hardware design as in a redesign of the software architecture needed to match the capabilities of the raw hardware.

We are working to move application-specific functionality directly onto the network interface, and thereby reduce I/O related data and control transfers to the host system to improve overall system performance. The resulting ensemble of host CPUs and device processors forms a potentially large distributed system.

In the context of our work, we are exploring how to program such a system at two levels. At one level, we are investigating how to migrate application functionality onto the network interface. Our approach is empirical: we take a monolithic application and migrate its I/O specific functionality into a number of *device extensions*. An extension is code that is logically part of the application, but runs directly on the network interface. At the next level, we are defining the operating systems interfaces that enable applications to compute directly on an intelligent network interface. Our operating system services rely on two technologies. First, applications and extensions communicate via a message-passing model based on Active Messages [5]. Second, the extensions run in a safe execution environment, called SPINE, that is derived from the SPIN operating system [1].

Applications that will benefit from this software architecture range from those that perform streaming I/O (e.g., multimedia clients/servers and file-servers), host based IP routers [10], cluster based storage management (e.g., Petal [3]), to support for packet filtering (e.g., Lazy Receive Processing [2]).

SPINE offers developers a software architecture for the following three features that are key to efficiently implement I/O intensive applications:

- **Device-to-device transfers.** By avoiding extra copies of data, we can significantly reduce bandwidth requirements in and out of host memory as well as halving bandwidth over a shared bus, such as PCI. Additionally, intelligent devices can avoid unnecessary control transfers to the host system as they can process the data before transferring it to a peer device. Techniques, such as SPLICE [16], have been introduced to emulate the device-to-device transfer.
- **Host/Device protocol partitioning.** Low-level protocol support for application-specific multicast [6], packet filtering (e.g., DPF [12]) and quality of service (e.g., Lazy Receive Processing [2]) has shown to significantly improve system performance.
- **Device-level memory management.** An important performance aspect of a network system is the ability to transfer directly between the network interface and the application buffers. This type of support has been investigated by various projects (e.g., UTLB [11], AMII [13], and UNET/MM [14]).

The rest of this paper is organized as follows. In Section 2 we describe the technology trends that argue for the design of smarter I/O devices. In Section 3 we describe the software architecture of SPINE. In Section 4 we describe some example applications that we've built using SPINE in the context of Windows NT. In Section 5 we discuss issues in splitting applications between the host and I/O subsystem. In Section 6 we present some conclusions drawn from our current experience.

2 Technology Trends Argue for Smarter I/O Devices

Here we briefly review why current technology trends argue for smarter I/O devices. System designers tend to optimize the CPU's interaction with the memory subsystem (e.g., deeper, larger caches, and faster memory busses) while ignoring I/O. Indeed, I/O is often the "orphan of computer architecture" [17]. For example, Sun Microsystems significantly improved the memory subsystem for its UltraSparc workstations over older generation Sparcs, yet neglected to improve its I/O bus (the SBUS).

Historically, I/O interconnects (primarily busses or serial lines) have been factors of 10 to an order of magnitude faster than the attached I/O devices. However, existing high performance device such as gigabit Ethernet and Fiberchannel can saturate the bus used in commodity systems. Consequently, fewer bus cycles are available for extraneous data movement into and out of host memory.

The standard speeds and bus widths necessary for open I/O architectures, such as 66 MHz x 64bits for PCI, often negate any of the performance improvements of using faster host CPU's for I/O intensive applications. The CPU will stall for a fixed number of bus cycles when accessing a PCI device via programmed I/O (PIO) regardless of the internal clock speed of the CPU. As internal processor clock rates increase, the relative time wasted due to stalls accessing devices over the bus grows larger. For example, a 400 MHz Pentium will stall for the same number of PCI bus cycles as a 200 MHz Pentium, but will waste twice as many processor cycles in the process.

Moving to a switched I/O interconnect, while improving aggregate bandwidth, does not improve the latency of access to I/O devices. Often, it can make the latency worse. Placing I/O data structures in main memory only shifts the bus-crossing burden to the I/O device, resulting in poor device performance as well as result in extraneous memory copies on the CPU side.

Considering these technology trends we conjecture that smart I/O devices will rapidly find their way into high volume server systems. In fact, recent industrial efforts (I2O systems for disk controllers and LAN devices, Intel's smart Ethernet adapter, and Alteon's ACEnic) corroborate our conjecture for smarter I/O devices. A key technology characteristic of embedded processors is that the processor core and I/O functions are often integrated into a single chip. Programs running on the resulting I/O processor have much faster access to I/O hardware, such as DMA engines, FIFOs and network transmit/receive support, than if they were running on the main CPU.

Although the placement of additional functionality onto the I/O device may require more memory to hold application-specific code and data technology trends point to increasing memory capacity per dollar for both DRAM and SRAM. It is not far fetched to envision I/O device designs that can incorporate up to 64MB of device memory.

In the next section we present a software architecture that aims at exploiting the hardware potential of such devices.

3 SPINE Software Architecture

Our approach is to provide an extensible runtime environment, called SPINE, appropriate for programmable network interface cards. SPINE extends the fundamental ideas of the SPIN extensible operating system -- type safe code downloaded into a trusted execution environment -- to the network interface. Extensibility is important, as we cannot predict the types of applications that may want to process directly on the network interface. Specifically, SPINE has three properties that are key to the construction of application-specific solutions:

- **Runtime Adaptation and Extensibility.** An application, regardless of its privilege level, may define a SPINE extension and dynamically load it onto an intelligent I/O device.
- **Performance.** SPINE extensions run in the same address space as the firmware, with low-overhead access to services and hardware. Extensions may directly transfer data using device-to-device DMA and communicate via peer-to-peer message queues. Overall system performance can be improved by eliminating superfluous I/O related control and data transfers.
- **Safety and Fault Isolation.** The use of a SPINE extension does not compromise the safety of other extensions, the firmware, or the host operating system. Extensions are isolated from the system and one another through the use of a type-safe language, enforced linking, and defensive interface design.

In the next two subsections we describe the SPINE runtime and the SPINE communication layer in more detail.

3.1 SPINE Runtime

The SPINE runtime is split across the network adapter and the host into I/O and kernel runtime components, respectively. The SPINE *kernel runtime* provides access to operating system threads, device and virtual memory subsystems, and provides host-side services to the SPINE I/O runtime. Applications may define SPINE extensions in Modula-3 that are loaded directly onto the network interface card. The SPINE *I/O runtime* used on the programmable network device exports internal and external interfaces. Internal interfaces are used by extensions that are loaded onto the network interface, which consists of the standard Modula-3 interface, memory management, peer-to-peer communication, and safe access to the underlying hardware (such as access to DMA and network send/receive engines). The external interface consists of message FIFOs that enable user-level applications, peer devices, and kernel modules to communicate with extensions on the network interface using an Active Message style communication layer (described in Section 3.2).

As illustrated in Figure 1, an application loads a SPINE I/O extension onto the network interface.

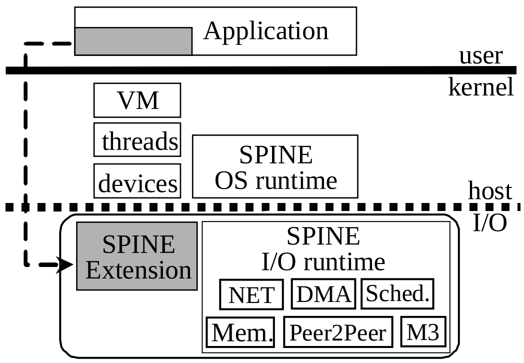

Figure 1. Spine System Structure. An Application dynamically loads extensions onto an I/O device to avoid unnecessary control/data transfer across the host I/O bus boundary.

Although SPINE leverages ideas from the SPIN extensible operating system, it does not depend on the host system running SPIN. In fact, we have ported the SPINE kernel runtime to the Microsoft Windows NT operating system and are using it as our main research vehicle.

The SPINE I/O runtime is responsible for managing resources on the network interface. For example, the data buffers that are used as the source/sink of DMA operations by extensions are one such resource. These buffers are accessed using unforgable references (i.e., an extension cannot create an arbitrary memory address and either DMA to or from it) that may refer to local memory, peer device memory, or host memory.

3.2 SPINE Communication Layer

The SPINE communication layer is a variation of Active Messages [5] used by the NOW project at UC-Berkeley. This message layer has the flexibility to execute extension code either on the network interface or the host CPU. The SPINE I/O runtime implements an Active Message dispatcher that routes the message to the host, a peer device, over the network, or invokes the handler of a local SPINE extension. SPINE extensions register Active Message handlers with the I/O runtime, which are invoked when a message arrives for them. All handlers are invoked with exactly two arguments: a context variable that contains information associated with the handler at installation time; and, a pointer to the message that contains the data as well as control information specifying the source and type of the message. There are two types of messages: small messages and bulk messages. Small messages are currently 64 bytes total, of which 48 bytes can be used to hold arbitrary user arguments. Bulk messages are similar to small messages, but contain a reference to a data buffer managed by the SPINE I/O runtime. The handler associated with the message is invoked only after all of the data for the message has arrived.

4 Example Applications

We have implemented a number of SPINE-based applications on a cluster of Intel Pentium Pro workstations (200 MHz, 64 MB memory) each running Windows NT version 4.0. Each node has at least one Myricom network interface card (Myrinet) on the PCI bus, containing 1MB SRAM card memory, a 33 MHz “LANai” processor, with a wire rate of 160MB/s. The LANai processor is a general-purpose processor that can be programmed with specialized control programs and plays a key role in allowing us to experiment with moving system functionality onto the network interface.

We have constructed two example applications to showcase application-specific extensions on an intelligent network interface card. The first extension is a video client that transfers image data from the network directly to the frame buffer. The second extension implements IP forwarding support and transfers IP packets from a source interface to a destination interface using peer-to-peer communication. The next two subsections describe these applications in more detail.

4.1 Video Client Extension

Using SPINE, we have built a video client application. The application defines an application-specific *video extension* that transfers video data arriving from the network directly to the frame buffer. The video client runs as a regular application on Windows NT. It is responsible for creating the framing window that will be used to display the video and informing the video extension of the window coordinates. The video extension on the network interface maintains window coordinate and size information, and DMA transfers video data arriving from the network to the region of frame buffer memory representing the application’s window. The video client application catches window movement events and informs the video extension of the new window coordinates.

The implementation of the video extension running on the network interface is simple. It is roughly 250 lines of code, which consists of functions to: a) instantiate per-window metadata for window coordinates, size, etc., b) update the metadata after a window event occurs (e.g., window movement), and c) DMA transfer data to the frame buffer. These functions are registered as active message handlers with the SPINE I/O runtime, and are invoked when a message arrives either from the host or the network.

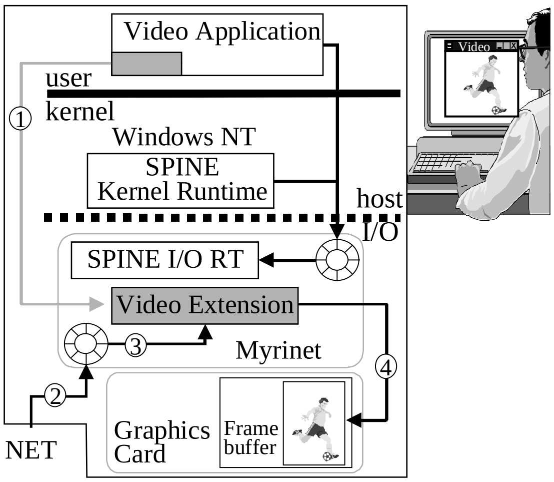

Figure 2 Network Video. The video extension transfers data from the network directly to the frame buffer, which reduces I/O channel load, frees host resources, and reduces latency to display video from the time it arrives from the network.

Figure 2 depicts the overall structure in more detail. The numbered arrows have the following meaning:

1. The video application loads extensions onto the card.
2. Packets containing video arrive from the network.
3. SPINE dispatches the packet to the video extension.
4. The video extension transfers the data directly to the frame buffer, and the video image appears on the user's screen.

Using this system structure the host processor is not used during the common case operation of displaying video to the screen. In our prototype system we've been able to support several video streams on a single host, each at sustained data rates of up to 40 Mbps, with a host CPU utilization of zero percent for the user-level video application. Thus, regardless of the operating systems I/O services and APIs, we can achieve high-performance video delivery.

Although the Myrinet's DMA engines can move large quantities of data, its LANai processor is to slow to decode video data on the fly. The LANai is roughly equivalent to a SPARC-1 processor (i.e., it represents roughly 1989 processor technology). Consequently, our video server takes on the brunt of the work and converts MPEG to raw bitmaps that are sent to the video client. Thus the video extension essentially acts as an application-specific data pump; taking data from the network and directly transfering it to the right location of the frame buffer. We expect fast, embedded processors to be built into future NICs that will enable on-the-fly video decoding, or one could use a graphics card that supports video decoding in hardware to avoid decoding on the NIC.

4.2 Internet Protocol Routing

We have built an Internet Protocol (IP) router based on SPINE. Fundamentally, an IP router has two basic operations. First, a router participates in some routing protocol (e.g., OSPF or BGP) with a set of peer routers. Second, a router forwards IP packets arriving from one network to another network. A busy router processes route updates roughly once a second, which is a computationally expensive operation. However, in the common case a router spends its time forwarding anywhere from 10^3 to 10^6 IP packets per second, which is an I/O intensive operation. Therefore it makes sense to leave the route processing on the host and migrate the packet forwarding function to the I/O card. This design is not uncommon, as many high-end routers (e.g., Cisco 7500 series) use specialized line cards that incorporate an IP forwarding engine and a centralized processor to handle route updates.

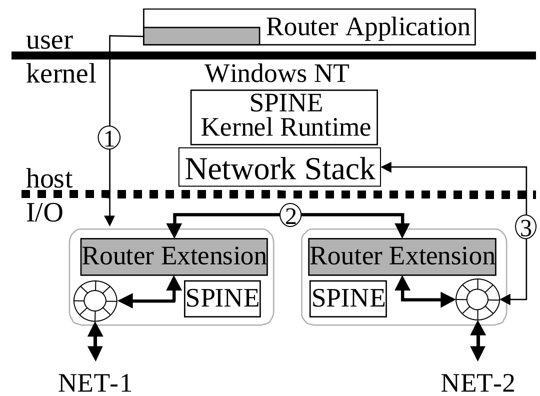

Figure 3 IP Router. In the common case the Router Extension independently forwards IP packets directly between network interface devices.

Figure 3 illustrates the overall architecture of the router that we built using SPINE. The first thing to note is that it is quite similar to the SPINE video client. The router application on the host loads the IP routing extensions onto the network interfaces (label 1) and initializes the forwarding table. IP packets arriving from the network are dispatched to the router extension, which determines how to forward packets by looking into the IP forwarding table. If the packet should be forwarded to another network interface (label 2), then the router extension can use the peer-to-peer communication support provided by SPINE. On the other hand, if the IP packet is intended for the host, then it is handed to the operating system's networking stack (label 3). The router, perhaps more so than the video client, demonstrates the distributed systems nature of SPINE. That is, extensions can communicate with the host, peer devices, or via the network.

In our experimental setup each network interface can forward 11,800 packets per second using the SPINE router extension, while placing zero load on the host CPU and memory subsystem as neither control or data needs to be transfer to the host system. In comparison, a host based IP forwarding system using identical hardware (i.e., multiple Myrinet interfaces plugged into a 200Mhz Pentium PC) built at USC/ISI achieves 12,000 packets per second while utilizing 100% of the host CPU [10]. The USC/ISI implementation optimizes the data path and only the IP packet header is copied into the host system, the remaining IP packet data is DMA transferred directly between the source and destination Myrinet interfaces. Note that the SPINE based implementation is only 2% slower using the slow embedded processor on the Myrinet interface compared to the host based forwarding implementation that uses a 200Mhz Pentium processor. However, our approach places zero load on the host system. As a result of this system structure, the host has plenty of processing cycles available to handle routing updates or more complex protocol processing while the intelligent network interface cards independently forward IP packets.

4.3 Other SPINE Uses

In the previous two subsections we described example applications that we were able to implement using a slow (33 MHz) embedded processor. Other example applications include packet filtering, accounting, masquerading, and virtual hosting that may also benefit from application-specific code running on the network interface.

The ORCA project demonstrated that application-specific multicast protocols perform better when handled directly from the network interface [6]. Lazy Receive Processing [2] requires operating system aware support directly on the network interface in order to processes packets fairly on a per socket basis. Both these device level protocols could be implemented straightforwardly using SPINE.

More powerful processor on the network interface will enable a number of additional applications. For example, using a vector processor, as suggested in [15] of the IRAM project, would enable data touching intensive applications, such as encryption, compression, video decoding, and data filtering, to be implemented on the network interface.

5 Discussion

The collection of processors on a given host in SPINE forms a small and well-integrated distributed system. However, the messaging architecture of SPINE is designed to scale to 100's of processors for a cluster of systems. How to make best use of the hardware potential is the key research question we are investigating in this work.

In an asymmetric multiprocessing system such as SPINE, the partitioning of work between the host and I/O processors may seem unclear at first. When does it make sense for the programmer to break up the application into a mainline program and extensions? How can a small set of helpers residing on I/O devices be of any use?

Often, the relative advantages of the different types of processors provide a clear path to a logical partitioning. The processor on the I/O device has inexpensive access to data from the communications media, while access to data in main memory, and the host processor's caches, is costly. The circumstances are reversed from the host processor perspective. As a result, code that frequently accesses data from the media should be placed on the device.

However, the price of this quick access to the bits coming off the media is constrained memory size, processing power and operating environment. Clearly, if much of the host OS functionality must be duplicated to the device, the potential benefits of using an intelligent interface may not be realized.

The asymmetric nature of the SPINE model leads to a methodology where the programmer looks for portions of the application where data movement is large, as in the video client, or control transfers among devices are frequent, as in the IP router. These parts of an application are ideal candidates for extensions to the network interface. Complex portions of the application, such as the routing protocol or interactions with operating system services (e.g., file system or the GUI), should remain on the host system.

A key limitation of SPINE as in most distributed system is that synchronization between components is expensive. The video application provides an instructive example. In our original implementation the movement of a window on the host was not synchronized with the transfer of data from the network interface to the frame buffer. Consequently, a few lines of image data would appear in locations of the screen where the window was prior to being moved. A work-around was to simply pause the video updates during window movement. However, a true solution would require the host and LANai extension to maintain the same view of the new window coordinates before the window can be moved.

6 Conclusions

Using SPINE, we have demonstrated that intelligent devices make it possible to implement application-specific functionality inside the network interface. Although hardware designs using a "front-end" I/O processors are not new, they traditionally have been relegated to special purpose machines (e.g., Auspex NFS server), mainframes (e.g., IBM 390 with channel controllers), or supercomputer designs (e.g., Cray Y-MP).

We believe that current trends will continue to favor the split style of design reflected in SPINE. Two technologies though could challenge the soundness of the SPINE approach. First, I/O functions could become integrated into the core of mainstream CPUs --- an unlikely event given pressures for cache capacity. Second, a very low latency standard interconnect could become available. However, given that I/O interconnects by their very nature must be both open and enduring, we believe these non-technical forces hinder growth of I/O performance more than anything will.

Our two example applications show that many extensions are viable *even with an incredibly slow I/O processor*. Based on our experience with the LANai, we believe that more aggressive processor and hardware structures would have a large positive impact on performance. For example, hardware FIFOs could eliminate much of the coordination overhead in our current system. A faster clock rate alone would significantly improve the Active Message event dispatch rate as well. We expect that a system using a current high-end I/O processor (clocked at roughly 250 MHz and with a cache size of 16KB) could improve performance by a factor of five over our current system.

As embedded processors continue to increase in power relative to I/O rates, the number of extensions that are possible will greatly increase. For example, having several megabytes of memory on the device enables NFS and HTTP caching extensions. A vector unit would allow many multimedia extensions. A faster CPU would allow the use of a virtual machine interpreter, enabling transparent execution of extensions regardless of the instruction set.

Acknowledgements

The authors would like to thank Intel for the generous supply of Pentium based PCs.

References

[1] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. *Extensibility, Safety and Performance in the SPIN Operating System*. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Dec. 1995.

[2] P. Druschel and B.Gaurav. *Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems*. In Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation (OSDI), Oct. 1996.

[3] E.K. Lee and C.A Thekkath. *Petal: Distributed Virtual Disks*. In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Oct. 1996.

[4] M.J. Feeley, W.E. Morgan, F. Pighin, A. Karlin, and H.M. Levy. *Implementing Global Memory Management in a Workstation Cluster*. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP). Dec. 1995.

[5] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. *Active Messages: A Mechanism for Integrated Communication and Computation*. Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA). May 1992.

[6] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K Langendoen, T. Ruhl, and K. Verstoep. *Performance of a High-Level Parallel Language on a High-Speed Network*. In Journal of Parallel and Distributed Computing, Jan 1997.

[7] Alteon Networks. *Gigabit Ethernet Technology Brief*. Available from: http://www.alteon.com/techbr.html

[8] W.J. Bolosky, J.S. Barrera III, R.P. Draves, R.P. Fitzgerald, G.A. Gibson, M.B. Jones, S.P. Levi, N.P. Myhrvold, and R.F. Rashid. *The Tiger Video Fileserver*. In Proceedings of the 6th Network and Operating System Support for Digital Audio Video (NOSSDAV) workshop, Apr. 1996.

[9] K. Jeffay, D.L. Stone, and F.D. Smith. *Transport and Display Mechanisms for Multimedia Conferencing Across Packet-Switched Networks*. Computer Networks and ISDN Systems. Jul. 1994.

[10] S. Walton, A. Hutton, and J. Touch. *Efficient High-Speed Data Paths for IP Forwarding using Host Based Routers*. In Proceedings of the 9th IEEE Workshop on Local and Metropolitan Area Networks. May 1998

[11] Y. Chen, C. Dubnicki, S. Damianakis, A. Bilas, and K. Li. *UTLB: A Mechanism for Address Translation on Network Interfaces*. In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). Oct. 1998.

[12] D.R. Engler and M.F. Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In Proceedings of the ACM SIGCOMM '96 Symposium on Communication Architectures, Protocols, and Applications.

[13] B.N. Chun, A.M. Mainwaring, and D.E. Culler. *A General-Purpose Protocol Architecture for a Low-Latency, Multi-gigabit System Area Network*. In Proceedings of the 5th Hot Interconnects Symposium. Aug. 1997.

[14] M. Welsh, A. Basu, and T. von Eicken. *Incorporating Memory Management into User-Level Network Interfaces*. In Proceedings of the 5th Hot Interconnects Symposium. Aug. 1997.

[15] K. Keeton, R. Apraci-Dusseau, and D.A. Patterson. *IRAM and SmartSIMM: Overcoming the I/O Bus Bottleneck*. Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA '97. Jun. 1997.

[16] K. Fall and J. Pasquale. *Exploiting In-kernel Data Paths to Improve I/O Throughput and CPU Availability*. In Proceedings of the 1993 Winter USENIX Conference. Jan 1993.

[17] J.L. Hennessy and D.A. Patterson. Computer Architecture: A Quantitative Approach, Second Edition. Morgan Kaufman Publishers, 1996.