SPINE: An Operating System for Intelligent Network Adapters

Marc E. Fiuczynski
Brian N. Bershad
Computer Science and Engineering
University of Washington
Seattle, WA

Richard P. Martin
David E. Culler
Electrical Engineering and Computer Science
University of California at Berkeley
Berkeley, CA

Abstract: The emergence of fast, cheap embedded processors presents the opportunity for processing to occur on the network adapter. We are investigating how a system design incorporating such an *intelligent* network adapter can be used for applications that benefit from being tightly integrated with the network subsystem. We are developing a safe, extensible operating system, called SPINE, which enables applications to compute directly on the network adapter. We demonstrate the feasibility of our approach with two applications: a video client and an Internet Protocol router. As a result of our system structure, image data is transferred only once over the I/O bus and places no load on the host CPU to display video at aggregate rates exceeding 100 Mbps. Similarly, the IP router can forward roughly 10,000 packets per second on each network adapter, while placing no load on the host CPU. Based on our experiences, we describe three hardware features useful for improving performance. Finally, we conclude that offloading work to the network adapter can make sense, even using current embedded processor technology.

1 Introduction

Many I/O intensive applications such as multimedia clients, file servers, and host based IP routers, 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 creating a performance gap that is increasingly becoming apparent for I/O intensive applications.

The exponential growth of processor speed relative to the rest of the I/O system does however present an opportunity for application processing to occur directly on intelligent I/O devices. Several network adapters, such as Myricom's LANai, Alteon's ACEnic, and I_2O systems, provide the infrastructure to compute on the device itself. With cheap, fast embedded processors (e.g., StrongARM, PowerPC, MIPS) used by intelligent network adapters, 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 hardware. In particular, there needs to be a clean framework to move application processing directly onto the network adapter, and thereby reduce I/O related data and control transfers to the host system to improve overall system performance.

This paper explores such a redesign at two levels. At one level, we are investigating how to migrate application processing onto the network adapter. Our approach is empirical: we begin with a monolithic application and move components of its I/O specific functionality into a number of *device extensions*, that are logically part of the application, but run directly on the network adapter. At the next level, we are defining the operating system interfaces that enable applications to compute on an intelligent network adapter. Our operating system services uses two technologies previously developed as a springboard: applications and extensions communicate via a message-passing model based on Active Messages [1], and, the extensions run in a safe execution environment, called SPINE, that is derived from the SPIN operating system [2].

The SPINE software architecture offers the following three features that are key to the efficient implementation of I/O intensive applications:

- *Device-to-device transfers.* Avoiding unnecessary indirection through host memory reduces memory bandwidth as well as bandwidth over a shared I/O bus. 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.
- *Host/Device protocol partitioning.* Servicing messages in the adapter can reduce host overhead and latency, and increase message pipelining, resulting in better performance.
- *Device-level memory management.* Transferring data directly between buffers on the device and the application's virtual address space reduces data movement and improves system performance.

There are a number of I/O intensive applications and system services that benefit from the use of these features. For example, cluster based storage management [3], multimedia applications [4, 5], and host based IP routers [6], benefit from being able to transfer data directly between devices in an application-specific manner. Host/Device protocol partitioning has been shown to be beneficial for application-specific multicast [7] and quality of service [8], and it may be useful for distributed memory management systems [9, 10]. Finally, device-level memory management has been investigated in the context of cluster-based parallel processing systems [11-13], which all require application-specific processing on the network adapter. The goal of SPINE is to provide an execution environment to support a multitude of applications that benefit from being tightly integrated with the network subsystem.

The rest of this paper is organized as follows. In Section 2 we discuss 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 discuss SPINE's programming philosophy of splitting applications between the host and I/O subsystem. In Section 5 we describe example applications that we have built using SPINE. In Section 6 we evaluate the performance of SPINE's message-driven architecture. In Section 7 we describe hardware enhancements for intelligent network adapters that are beneficial for a variety of applications. In Section 8 we review related work, and in Section 9 we present some conclusions drawn from our current experience.

2 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" [14]. 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 orders of magnitude faster than the attached I/O devices. However, existing high-performance devices such as Gigabit Ethernet and Fibre-Channel Arbitrated Loop 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 33 MHz x 32bits for PCI, often negate the performance improvements of using a faster host CPU for I/O intensive applications. The host CPU stalls for a fixed number of bus cycles when accessing a PCI device via programmed I/O regardless of its internal clock rate. 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.

Embedded I/O processors have traditionally been substantially slower compared to processors used in host systems. However, with the concession of the PC and workstation market to x86 processors, many processor manufactures are aggressively pursuing the embedded systems market. As a result, inexpensive yet high-performance processors (e.g., StrongARM, MIPS, and PowerPC) are finding their way into intelligent I/O devices. An important technology characteristic of these 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 64Mbytes of device memory for both logic and buffering.

Considering these technology trends we conjecture that smart I/O devices will rapidly find their way into high volume server systems and eventually client systems. In fact, recent industrial efforts (I_2O disk controllers and LAN devices, Intel's smart Ethernet adapter, and Alteon's gigabit Ethernet adapter) corroborate our conjecture for smarter I/O devices.

This motivates us to develop a software architecture that aims at exploiting the hardware potential of such devices.

3 SPINE Software Architecture

Introducing application-specific code to execute on an intelligent adapter raises many questions. How is code loaded onto the adapter? How and when does it execute? How does one protect against bugs? How does this code communicate with other modules located on the same adapter, peer adapters, remote devices, or host-based applications spread across a network?

We address these questions using technology derived from the SPIN operating system and communication technology from the NOW project to design an extensible runtime environment, called SPINE. SPINE provides a Safe Programmable and Integrated Network Environment.

SPINE extends the fundamental ideas of the SPIN extensible operating system -- type-safe code downloaded into a trusted execution environment -- to the network adapter. Extensibility is important, as we cannot predict the types of applications that may want to run directly on the network adapter. Specifically, SPINE has three properties that are central to the construction of application-specific solutions:

- *Performance*. SPINE extensions run in the same address space as the firmware, resulting in low-overhead access to system 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.

- *Runtime Adaptation and Extensibility.* SPINE extensions may be defined and dynamically loaded onto an intelligent I/O device by any application.
- *Safety and Fault Isolation.* SPINE extensions may 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.

The following subsections describe the system in greater detail.

3.1 SPINE System Structure

The SPINE system structure is illustrated in Figure 1. To program intelligent adapters requires operating system support both on the host and on the I/O processor. SPINE is split across the network adapter and the host into I/O and OS runtime components, respectively. An application defines code to be loaded onto the network adapter as a SPINE *extension*.

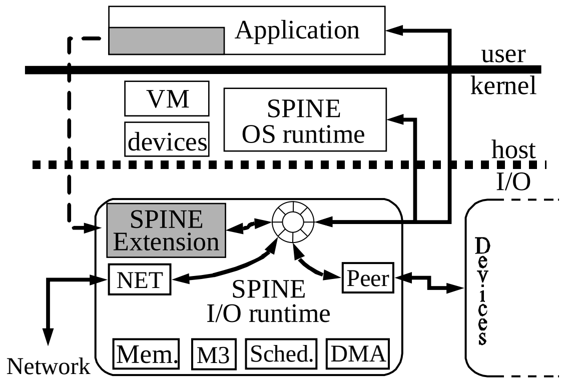

Figure 1. **Spine System Structure.** An Application dynamically loads SPINE extensions onto an I/O device (see dashed arrow) to avoid unnecessary control/data transfer across the host I/O bus boundary. Messages may be sent to or received from the network, user-level applications, kernel modules, or peer devices (see solid arrows).

The SPINE *OS runtime* interfaces with the host operating system's device and virtual memory subsystems. It provides a consistent interface across operating system platforms to SPINE's user-level communication library as well as SPINE's dynamic linker/loader. Applications inject extensions onto the adapter using SPINE's dynamic linker, and send messages directly to extensions via a memory mapped FIFO using SPINE's communication library. The OS runtime currently works in the context of Linux and Windows NT.

The SPINE *I/O runtime* used on the network adapter is a small operating system that controls the underlying device and provides an execution environment to extensions. The I/O runtime exports a set of internal and external interfaces. Internal interfaces are used by extensions that are loaded onto the network adapter, and consist of support for messaging, safe access to the underlying hardware (e.g., DMA controllers), and a subset of the Modula-3 interface. As illustrated by the solid arrows in Figure 1, the external interface consists of message FIFOs that enable user-level applications, peer devices, and kernel modules to communicate with extensions on the network adapter using an active message style communication layer.

3.2 A Message-Driven Architecture

The primary purpose of a network adapter is to move messages efficiently between the system and the network media. The network adapter's basic unit of work is thus a message. To efficiently schedule messages and the associated message processing, our operating system uses a message driven scheduling architecture, rather than the process or thread-oriented scheduling architecture found in conventional operating systems. The primary goal is to sustain three forms of concurrent operation: host-to-adapter message movement, message processing on the adapter, and adapter-to-network message movement. In SPINE the message dispatcher manages these concurrent operations.

An intelligent network adapter not only moves data; it reacts and applies transformations to it as well. On message arrival, the adapter may have to operate on the data in addition to moving it. This style of processing is captured well by the Active Message [1] programming model, which we use to program SPINE extensions on the network adapter. Briefly, an Active Message is a message that carries data as well as an index to a code sequence to execute on message arrival. The code sequence is called a *handler*. Every message in the system must carry an index to a handler. SPINE extensions are thus expressed as a group of active message handlers. On message arrival, the SPINE dispatcher can route messages to the host, a peer device, over the network, or invoke an active message handler of a local SPINE extension.

The two goals of handler execution and rapid message flow implies that handlers must be short-lived. Thus, the "contract" between the SPINE runtime and extension handlers is that the handlers are given a small, but predictable time to execute. If a handler exceeds the threshold it is terminated in the interests of the forward progress of other messages. Long computations are possible, but must be expressed as a sequence of handlers. The premature termination of handlers opens a Pandora's box of safety issues that we address in the next section.

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 256 bytes total, of which 240 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.

To increase the amount of pipelining in the system, message processing is broken up into a series of smaller events. These events are internally scheduled as active messages, which invoke system provided handlers. Thus, only a single processing loop exists; all work is uniformly implemented as active messages. We document the performance of these internal messages, and thus the throughput and latency of the SPINE I/O runtime, in Section 6.

3.3 A Controlled Execution Environment

The choice of our execution environment was dictated by two design goals. First, we wanted to support programmable network adapters whose processor architecture does not support multiple address spaces. Second, we wanted to invoke application-specific code in response to messages with low overhead. These two goals led to a single address space design.

To safely run user provided extensions in a single address space, the system needs to guard against wild memory references, jumps to arbitrary code, and excessive execution time. Our approach is to run extensions in a controlled execution environment that is enforced using a combination of language, runtime, and download-time mechanisms, as described in the following three subsections.

3.3.1 Enforcing Memory Safety

User provided extensions are written in a type-safe language, which enables extensions and the SPINE I/O runtime to safely execute in the same address space, as it provides strong guarantees of protection from imported code. In particular, a program cannot access or jump to arbitrary memory locations, and cannot use values in ways not prescribed by the language.

As our type-safe language we are using a version of Modula-3[1] that has been enhanced to support efficient, low-level, systems code. Hsieh et al. [16, 17] describes these language enhancements in further detail.

SPINE and its extensions use the following two Modula-3 language enhancements: type safe casting and isolation from untrusted code. The former allows data created outside of the language to be given a Modula-3 type, thereby enabling extensions to safely interpret message data using aggregate types (e.g., a record describing the layout of a packet header). The latter allows a caller to isolate itself from runtime exceptions (e.g., nil dereferences and divide by zero) as well as the capability to terminate the execution of (potentially malicious) long running extensions. We have previously used these mechanisms for similar purposes in the context of an extensible protocol architecture for application-specific networking [18].

3.3.2 Guarding Against Excessive Execution Time

To prevent a misbehaving active message handler from stalling the system, it is necessary to asynchronously terminate such a handler. Before invoking an extension handler, the runtime sets a watchdog timer interrupt. If the watchdog timer expires, the runtime aborts the currently running handler and returns control to the system. This simple timeout mechanism enables the system to make forward progress even in the presence of malicious handlers.

It is impossible, though, for the system to determine whether a handler that exceeds its execution time is simply faulty or intentionally malicious. One could imagine maintaining statistics on how often a particular handler exceeds the system threshold and avoid executing it for future messages. However, we have not experimented with such policies. For applications that we have implemented so far, the timeout value can be safely set as low as 10 microseconds. However, for future applications this timeout value may be too low. We are still investigating what a reasonable watchdog timeout value is in general.

Although preventing excessive run time of handlers guarantees forward progress of the system, obliviously terminating handlers introduces a new problem: unchecked handler termination could damage system state or other extensions. For example, a terminated handler cannot be allowed to hold locks or insert into system linked lists. These types of operations may leave the application or the system in an inconsistent state if terminated arbitrarily.

Our approach is to explicitly label extension procedures with a special `EPHEMERAL` type. The semantics of an EPHEMERAL procedure is that it may be terminated at any point in time. An obvious restriction on ephemeral procedures is that they can only call other ephemeral procedures.

This type-modifier enables SPINE to rely on compile checks to ensure that the system can tolerate premature termination of active message handlers. SPINE carefully defines a subset of its interfaces as ephemeral and others as non-ephemeral. In addition, Modula-3's LOCK

[1] Although Modula-3 [15] is less popular compared to Java, it enables us to focus on system design issues using a high-level language rather than on interpretation efficiency. The two languages are sufficiently similar that results from our research may be applicable to Java once a good compiler for it exists.

statement and other non-ephemeral language constructs are also defined as non-ephemeral.

Figure 2 illustrates the use of `EPHEMERAL` in the context of SPINE. At the top of the figure is the system interface to install active message handlers, called `AddHandler`. Extensions using this interface must provide as an argument a procedure whose type is `HandlerProc` that is defined as an ephemeral procedure type. The bottom of Figure 2 shows the use of the `AddHandler` interface. Providing a non-ephemeral

```
(** Interface to install AM handlers **)
TYPE HandlerProc = EPHEMERAL
 PROCEDURE (message: Msg;
            context: Context);

PROCEDURE AddHandler(
            handler: HandlerProc;
            hNumber: HandlerNum;
            context: Contex):BOOLEAN;

(**** User Provided Extension Code ****)
EPHEMERAL PROCEDURE
  GoodHandler(m:Msg; c:Context)=
 BEGIN
  (* Process message and return. *)
 END GoodHandler;

(* This procedure will be terminated *)
EPHEMERAL PROCEDURE
 InfiniteHandler(m:Msg; c:Context)=
 BEGIN
  WHILE TRUE DO (* infinite loop *) END;
 END InfiniteHandler;

(* This procedure is not EPHEMERAL *)
PROCEDURE NonEphemeral(m:Msg) =
 BEGIN
  LOCK myExtensionMutex DO ... END;
END NonEphemeral;

(* This procedure will not compile as *)
(* it calls a non-ephemeral procedure *)
EPHEMERAL PROCEDURE
  IllegalHandler(m:Msg; c:Context)=
 BEGIN
  NonEphemeral(m);
 END IllegalHandler;

PROCEDURE Init() = BEGIN
 AddHandler(IllegalHandler, 5, NIL);
 AddHandler(GoodHandler, 6, NIL);
 AddHandler(InfiniteHandler, 7, NIL);
END Init;
```

Figure 2. **Example uses of EPHEMERAL.** Active message handlers in SPINE are defined as ephemeral and installed using the systems AddHandler interface. The compiler will generate an error when compiling Illegalhandler since it calls a procedure that is not ephemeral.

procedure as an argument, for example, will result in a type mismatch and the code will be rejected at compile time. Also shown in Figure 2 is a compile-time error in the body of an ephemeral procedure, `IllegalHandler.` This procedure will result in a compilation error, because it is typed as `EPHEMERAL`, but makes a call to a non-ephemeral procedure.

Both `GoodHandler` and `InfiniteHandler` are valid ephemeral procedures that can be installed as active message handlers. `InfiniteHandler`, though, when invoked by the system will execute an infinite loop and not return control. However, since it is defined as an ephemeral procedure, the system may safely abort such a procedure at any time.

The use of ephemeral however does not prevent an application from harming itself. For example, an extension may synthesize its own lock, thereby potentially causing it to deadlock on itself. We do not expect this to be a problem, as `EPHEMERAL` serves as a reminder that little should be done within an ephemeral procedure. More importantly, `EPHEMERAL` prevents an application from harming other extensions or the system, which is our primary goal.

3.3.3 Safe Code Installation

Code installation is solved by SPINE's dynamic linker, which accepts extensions implemented as partially resolved object files. The linker resolves any unresolved symbols in the extension against *logical protection domains* [2] with which the extension is being linked. A logical protection domain (henceforth referred to simply as a *domain*) defines a set of visible interfaces that provide access to procedures and variables. This is similar to a Java class name space. The I/O runtime and extensions exist within their own domains and export interfaces that may be used by other extensions. If an extension imports an interface not contained within the set of domains that it has access to, then the extension will be rejected by the system at download-time. This mechanism prevents extension code from accessing low-level system functionality or state that it could use to violate system safety. Sirer et al. [19] describes the name space management provided by domains and safe dynamic linking in further detail.

To economize network adapter resources, the dynamic linker implementation as well as domain and symbol table information is left on the host system. Only code/data sections are downloaded onto the adapter. As a result, the network adapter has more resources for the application, as memory is not consumed by the linker or by a potentially large symbol table. This approach is not new; Linux and various embedded systems use similar technology.

Finally, the SPINE linker assumes that digital signatures can be used to ensure that a trusted compiler created the object code for an extension, as in the Inferno system

[20]. In the future, it may be possible to use Proof Carrying Code [21] generated by a Certifying Compiler [22] to ensure that extensions are type-safe. In either case, a dynamic check is necessary by the SPINE linker to ensure that compile-time restrictions (such as type safety) are obeyed by the submitted object code.

3.4 Memory Management

To simplify resource management, extensions may not allocate memory dynamically. Instead, at initialization time an application defined memory region is allocated by SPINE, which may be used by the extension as a scratch area to hold application-specific information. However, there are two resources (data buffers and messages) that require dynamic resource management. Data buffers and messages are created for extensions on message arrival, and careful management is required to reclaim these resources safely from extensions.

Data buffers are used as the source/sink of data transfers operations (such as DMA and programmed I/O) and may refer to local memory, peer device memory, or host memory. Messages provide the means to communicate between extensions on peer adapters, remote devices, or host-based applications spread across a network, and may contain references to data buffers. These two resources are named by type-safe references that cannot be forged by an extension (e.g., an extension cannot create an arbitrary memory address and either DMA to or from it). Extensions gain access to messages and data buffers through system interfaces. For example, an active message handler is passed a message as an argument, which may contain a reference to a data buffer.

We are still investigating how to properly manage these resources in an extensible system. At one level, the system cannot depend on extensions to correctly release messages or data buffers for reuse (e.g., a buggy implementation may simply fail to do so). Our approach for this is to use a reference counting garbage collector that returns unused messages and data buffers to the system pool as soon as possible. At another level, the system may need to forcefully reclaim resources from extensions. We are investigating language level support, similar to ephemeral procedures, in order to asynchronously reclaim resources safely from extensions.

3.5 Summary

Enabling application-specific processing to occur on the network adapter places unique requirements on the operating system in terms of performance and safety. We address these requirements by combining technology from two independent projects. From the NOW project, the Active Message programming model defines how applications may process messages on the adapter. From the SPIN project, the use of a type-safe language and enforced linking ensures that applications cannot violate system safety on the adapter.

The key idea in SPINE is to partition an application into a host-resident component and an adapter-resident component. When and how this should be done is the topic of the next section.

4 SPINE Programming Philosophy

In an asymmetric multiprocessing system such as SPINE, the development of applications raises several serious questions. When does it make sense for the programmer to spend the effort 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? What functions should be offloaded to the I/O processor, and which are best left on the host?

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 of this "inverted memory hierarchy", 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 limited memory size, processing power and operating environment. Clearly, if much of the host OS functionality for an extension must be duplicated on the device, the potential benefits of using an intelligent adapter may not be realized.

The asymmetric nature of the SPINE model thus leads to a methodology where the programmer looks for portions of the application where data movement into host memory is unnecessary (e.g., a video client) or control transfers among devices are frequent (e.g., an IP router) to become extensions to the network adapter. 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.

5 Example Applications

We have implemented a number of SPINE-based applications on a cluster of Intel Pentium Pro workstations (200MHz, 64MB memory) each running Windows NT version 4.0. Each node has at least one Myricom network adapter (LANai) 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 adapter.

We discuss two example applications that showcase application-specific extensions on an intelligent network adapter. 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 adapter to a destination adapter using peer-to-peer communication. The next two subsections describe these applications in more detail.

5.1 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 displays the video and informing the video extension of the window coordinates. The video extension on the network adapter 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 adapter 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.

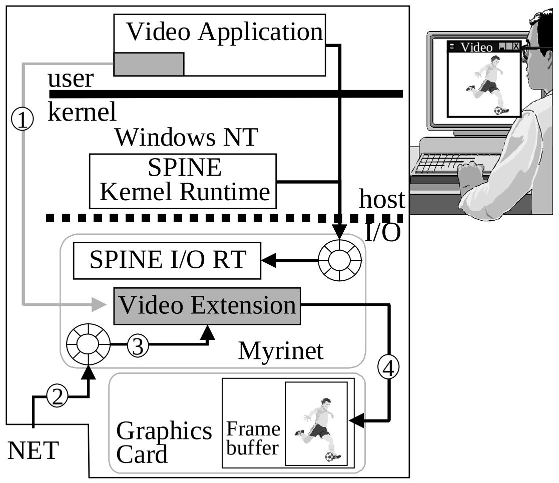

Figure 3. **Network Video.** The video extension transfers data from the network directly to the frame buffer, which reduces I/O channel load, frees host resources, and reduces latency to display video from the time it arrives from the network.

Figure 3 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 LANai's DMA engines can move large quantities of data, its LANai processor is too slow to decode video data on the fly. The LANai is roughly equivalent to a SPARC-1 processor (i.e., it represents roughly 10 year old 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 transferring 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.

A key limitation of SPINE extensions that programmers must be aware of is that *frequent* synchronization between host and device-based components is expensive. In our original implementation of the video client the movement of a window on the host was not synchronized with the transfer of data from the network adapter 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, an ideal solution would require the host and video extension to maintain the same view of new window coordinates at all times.

5.2 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.

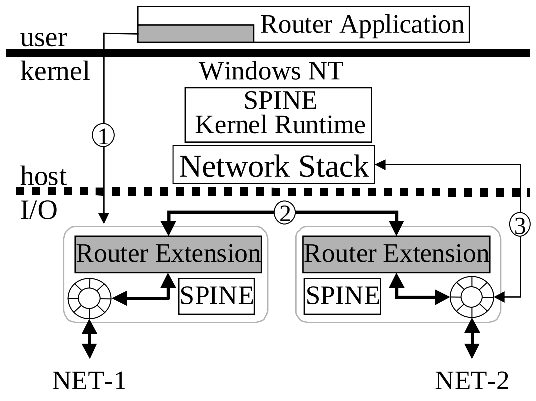

Figure 4. **The SPINE IP Router**. In the common case the Router Extension independently forwards IP packets directly between network adapters. The forwarding path between interfaces, shown in step 2, can be either over the PCI bus or across the Myrinet switch fabric.

Figure 4 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 adapters (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 adapter (label 2), then the router extension can use the peer-to-peer communication support provided by SPINE. In SPINE peer communication is transparently sent over the PCI or the Myrinet switch fabric. If the IP packet is intended for the host, then it is handed to the operating system’s networking stack (label 3). The router demonstrates the distributed systems nature of SPINE. That is, extensions can communicate with the host, peer devices, or via the network.

Figure 5 shows the actual forwarding rate under high load. We used algorithms for forwarding table compression and fast IP lookup described in [23], and inserted one thousand routes into less than 20Kbytes of memory on the adapter. The experiment was to forward two million packets at a maximum rate such that the percentage of dropped packets was less than 0.5%. In the first experiment, labeled “PCI”, the packets were transferred to the egress adapter via the PCI bus. In the second experiment, labeled “Wire”, packets are instead “re-sent” to the egress adapter across the Myrinet switch fabric. Note that the "Wire" experiment serves the sole purpose of verifying the conjecture that our system could perform better with hardware support for messaging. We elaborate on this issue in Section 7.1. Figure 5 shows that forwarding packets over the switch fabric results in substantially higher performance than over the PCI bus.

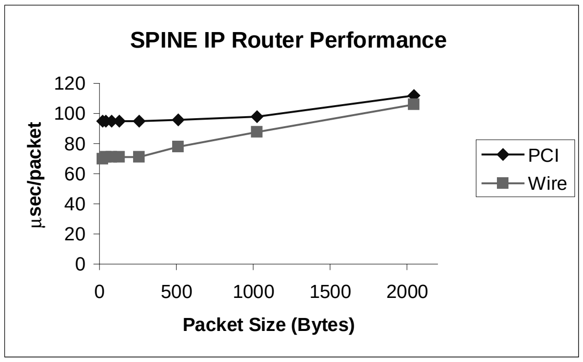

Figure 5. **Performance of the SPINE IP Router**. The average time per packet is shown under peak load for two forwarding schemes. In the first scheme, packets are sent from one LANai to another via the PCI bus. In the second scheme, packets are routed via the Myrinet switch fabric.

In our experimental setup each network adapter using the SPINE router extension can forward 11,300 packets per second over the PCI bus, and 14,100 packets per second over the Myrinet switch fabric. The router places zero load on the host CPU and memory subsystem because neither control nor data needs to be transferred to the host. In comparison, a host based IP forwarding system using identical hardware (i.e., multiple LANai adapters plugged into a 200MHz Pentium Pro PC) built at USC/ISI achieves 12,000 packets per second over PCI while utilizing 100% of the host CPU [6]. The USC/ISI host based IP router implementation optimizes the data path and only the IP packet header is copied into the host system, while the remaining IP packet is transferred directly using device-to-device DMA between the source and destination LANai adapters.

The SPINE based implementation transferring IP packets over the PCI bus is only 6% slower using a slow 33MHz embedded processor compared to the host based forwarding implementation that uses a 200MHz Pentium Pro processor. Further, our approach can achieve higher aggregate throughput as it scales with the number of network adapters in the system, even when using a slow embedded processor. This illustrates that the combination of device-to-device transfers and servicing the IP route lookup on the adapter leads to better overall performance. Additionally, as a result of SPINE’s system structure, zero load is placed on the host CPU, thereby leaving plenty of processing cycles available to handle routing updates or more complex protocol processing while the intelligent adapters independently forward IP packets.

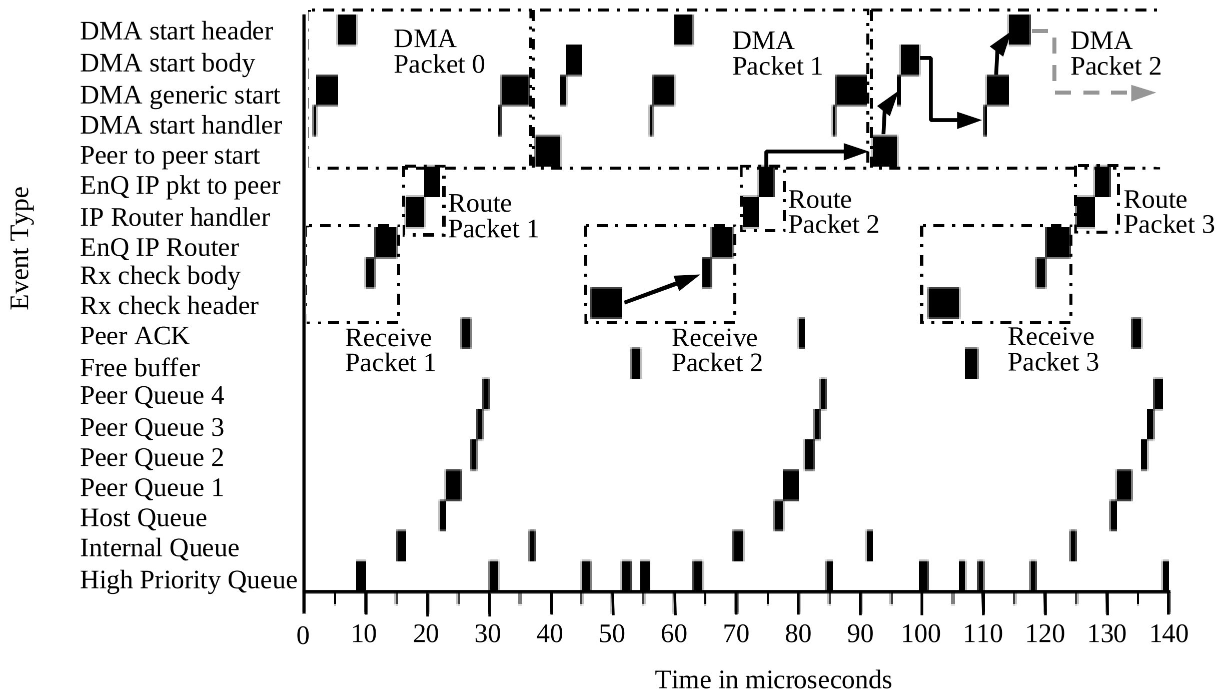

Figure 6. **SPINE IP Router Event Graph**. The figure plots events as they unfold in the SPINE IP Router. Time is shown on the x-axis and event types along the y-axis. A box is plotted at the point at the time of each event occurrence. The width of the box corresponds to the length of the event. The dashed rectangles correspond to higher-level packet semantics: receiving a packet, routing it, and forwarding it over the PCI bus. The arrows trace the causal relationships of a single packet as it moves through the system.

6 SPINE Event Processing

In this section, we show how SPINE event processing translates into overall performance. The current SPINE runtime optimizes for high throughput of small messages. Part of this motivation comes from recent work demonstrating that with low software overhead, many applications have a high degree of latency tolerance [24]. At the same time however, it must be general enough to support user extensions. The SPINE I/O runtime is constructed as a generic event handler.

Recall from Section 3.2 that all SPINE processing is ultimately decomposed into internal Active Message handlers. For example, when the IP router calls the procedure to forward a message to a peer device, the runtime implements this functionality by issuing a series of active messages to itself. Thus, event and handler invocation becomes synonymous. This approach not only simplifies the dispatcher, but it also exposes the natural parallelism inherent in the operation.

Figure 6 shows a real snapshot of the event processing from the SPINE IP router when routing to a peer node over the PCI bus. The x-axis represents time. The y-axis is an enumeration of event types. The dark boxes represent time spent in event processing. The position of the boxes on the x-axis shows the time a particular event took place. The position on the y-axis shows what event was processed at that time. The width of the dark box shows the length of the event. For example, during the period between 16 and 20 microseconds, the IP routing handler was running.

Because SPINE breaks message processing into many tiny events, the event graph at first appears to be a jumbled maze with little discernable structure. In reality however, Figure 6 shows a regular, periodic pattern. The key to understanding the event graph is to recognize these high level patterns emerging from the thicket of small events. From these patterns we can deduce both the rate at which packets are processed, as well as the latency of a packet as it passes through the system.

The dashed rectangles in Figure 6 outline higher-level packet-processing steps. Recall that to route an IP packet requires 3 steps: receiving the packet, determining the destination (routing), and forwarding the packet to an egress adapter.

The bandwidth, in terms of packets per second, is easily discernable via the period of new processing steps in the event graph. The time between receiving packets 1, 2 and 3 in Figure 6 is roughly 55 microseconds. The period for other kinds of processing, such as routing and DMA, is similar. Thus, we can conclude that we can route a new IP packet once every 55 microseconds.

The latency, or time it takes a single packet to move through the system, is observable by tracing the path of a single packet. The arrows in Figure 6 trace the processing path of packet 2 as it is received, routed, and transferred to the egress adapter. From the graph we can see that the latency of a packet through the router is roughly 100 microseconds. Note that the bandwidth and latency are distinct due to overlap (as shown in the figure).

A large discrepancy exists between the measured time from Figure 5 (95 microseconds/packet) and the observed times in Figure 6 (55 microseconds/packet). The key to understanding the missing 40 microseconds is that Figure 6 does not show a typical event ordering; rather it is a best case ordering.

We have discovered by observing many event plots that the slightly lower priority of PCI events in SPINE results in oscillations of event processing. Several incoming packets are serviced before any outgoing packets are sent over the PCI. These oscillations break an otherwise smooth packet pipeline. The system oscillates between the "fast" state of "one-for-one" receive-and-send, and the "slow" state of receiving a few packets and draining them over PCI. The net result is that the average forwarding time is reduced from 55 to 95 microseconds. We are currently investigating ways to improve the priority system of the SPINE dispatcher to eliminate this effect.

A closer look at Figure 6 shows a key limitation of the SPINE architecture. SPINE was constructed to be general enough so that the natural parallelism of packet processing would automatically be interleaved with user extensions. However, precise scheduling is possible if we *a-priori* knew what sequence of events need to be processed, and thereby achieve better overall performance. Indeed, many other projects (e.g., [25-29]) have exploited this fact and developed firmware with a fixed event-processing schedule that is specialized for a specific message abstraction or application. The lack of an efficient, static processing schedule appears to be an inherent limitation of any general-purpose system.

From the event graph, we have seen that the periodic rate as well as the latency of packet processing is limited in three ways: (1) the number of events dispatched per message, (2) the speed of each event, and (3) the ordering of events. Many of these limitations in our implementation are due to the fact that SPINE has to emulate various abstractions in software for the LANai adapter. The next section describes several hardware structures that reduce the cost of event processing.

7 Hardware Enhancements

In this section, we estimate the potential performance improvement of three hardware features that our network adapter lacks: hardware FIFOs as messaging units, support for chained and strided DMA, and a faster processor.

7.1 Hardware FIFOs

A hardware FIFO is a memory abstraction that implements a buffer with two interfaces. One that fills the buffer and the other that drains it. The drain removes the oldest entry first.

SPINE incurs a substantial performance penalty, as it has to emulate these FIFOs in software for the LANai adapter. Using hardware FIFOs for synchronized messaging, much like those found in [30, 31], would substantially reduce communication costs over the shared bus. It would improve communication performance with peer devices as well as communication with the host. In particular, a hardware based implementation of these FIFOs would reduce the number of events needed in SPINE to process a message and substantially cut down on adapter memory resources devoted to enable unsynchronized messaging among a set of peer devices.

Recall from Figure 5 that the performance of the IP router was substantially better when using the Myrinet switch fabric, which uses hardware support similar to hardware FIFOs for synchronized messaging among network adapters. We expect that the "PCI"-based IP forwarding performance, as show in Figure 5, would approach the "Wire"-based IP forwarding performance if the LANai adapter had hardware FIFOs to support messaging over the PCI bus. That is, nearly a 25% performance improvement could be achieved with a simple piece of hardware.

7.2 Chained and Strided DMA

A chained DMA controller simply has support to follow a list of pointer/length pairs in adapter SRAM. (Chained DMA is also often referred in the literature as scatter/gather DMA). Unfortunately, current LANai adapters do not have this type of support. Therefore SPINE has to emulate chained DMA in software by scheduling a handler that determines when a particular DMA operation completed in order to start the next DMA operation. This type of support would be useful for both the IP router and the video client in reducing the number of events to process.

We estimate for the IP router, that chained DMA could eliminate several events and save twelve microseconds per packet. Similarly, chained DMA is useful for the video client as an image in frame buffer memory is not contiguous, and therefore each scan line of an image must be DMA transferred separately.

Strided DMA is a more unusual hardware abstraction. Such a DMA engine would place each word of data to addresses with a constant stride, much like a strided vector unit. In our work with the video client, we have found several frame buffers that use non-contiguous, but constant strided addresses for successive pixels. A strided DMA unit would provide an ideal match for such frame buffers.

7.3 Faster Processor

Another method to shorten the period of the event processing in SPINE is to shorten the length of each event. Given that the most commonly executed SPINE code easily fits in a 32K cache (and thus instruction cache misses are unlikely), a faster processor would translate directly into faster event processing. As represented in Figure 6, this would cut down the width of each box. With a high-performance embedded processor, such as a 200MHz StrongARM, we conservatively estimate the average time per event at roughly 0.5 microseconds (or roughly a 5x performance improvement).

7.4 Hardware Summary

Using hardware FIFOs and chained DMA in the IP router, the number of events processed per packet would shrink, thereby reducing processing time to 33 microseconds per packet on the LANai. This kind of performance would place such a system well ahead of current host-based routers. Our conservative estimate shows that, by using all three hardware enhancements, current embedded processor technology can obtain a throughput of one packet per ten microseconds, or a rate of 100,000 packets per second. Although this rate does not approach specialized forwarding ASICs (such as those used in Packet Engines routing switches [32]), it is much faster than a host-based router and would allow routers to be constructed with cheaper, slower, primary processors. However, unlike the ASIC-based approach, SPINE gives the programmer nearly the same flexibility as a traditional host-based system. For example, it would be straightforward to add firewall or quality of service capabilities to the router.

8 Related Work

There has been substantial work in offloading functionality onto intelligent network adapters. The core ideas in this area can be traced back to the first super-computers and high-performance I/O systems. In the 1960s, the CDC 6600 and IBM 360 systems used I/O processors extensively. However, these processors were used to manage many low-level details of I/O processing, rather than application code or network protocols. Today most of these functions are realized in modern controller cards.

In the context of network adapters, there has been a decade long ebb and flow of the amount of network protocol placed onto the adapter. However, very little has been discussed about *application functionally*, as opposed to protocol support (e.g. a video client vs. placing TCP into the adapter). Most of the debate centers on how much protocol to offload, ranging from no support, to placing pieces of the protocol, to offloading the entire protocol.

In the early 1980's many commercial network adapter designs incorporated the entire protocol stack into the adapter. There were two reasons for such an approach, which, due to its complexity, required an I/O processor. First, many host operating systems did not support the range of protocols that existed at the time (e.g., TCP/IP, telnet, and rlogin) [33]. Writing these protocols once for an intelligent network adapter was an effective method of quickly incorporating protocols into a variety of operating systems. Second, the host processors of the time were not powerful enough to run both multi-tasking jobs and network protocol stacks efficiently [34-36].

The logical apex of the greater and greater inclusion of protocol into the adapter is best exemplified by the Auspex NFS server [37]. Each component of the Auspex server (disk, network, NFS cache) has an I/O processor that is specialized for NFS services. A lightweight kernel running on each I/O processor controls the communication among the functional I/O components of the NFS server. This separation of NFS services from the host operating system enables higher performance, scalability, and resilience to host failure [38].

By the late 1980's however, the tide had turned, with only support for very common protocol operations included in the adapter. The migration of common protocols into commodity operating systems and the exponential growth of processor speed eliminated the original motivations for intelligent network adapters at the time. There has been a great deal of work, however, in offloading pieces of network protocols. For example, there has been work to offload Internet checksum calculations [30, 39, 40], link layer processing [41-45], and packet filtering [8, 46]. Interestingly, the tide might be turning again: recently published results show that offloading TCP/IP to intelligent network adapters yields better performance for an enterprise-wide server system running Windows NT [47].

In recent years, researchers from the parallel processing community have taken a different strategy all together: eliminate protocols as much as possible. The motivation behind this work was an attempt to replicate the communication performance of Massively Parallel Processors (MPPs) on stock hardware. A common method of eliminating the operating system protocols was to use a programmable network adapter. This strategy allows the networking layers to expose the adapter directly to applications [26, 27, 48]. The elimination of the operating system yields a factor of 10 improvement in software overhead over conventional protocol stacks, and peak bandwidths thus become achievable with small packets [49]. Our work was originally motivated by these results of offloading overhead from the host system.

I_2O is a recent technology providing a general architecture for coupling host systems with I/O devices [31]. The primary goals of the I_2O work are the elimination of multitudes of device drivers and improved performance by reducing device related processing and resource requirements from the host. Although I_2O provides a message-passing protocol between the host and devices, its focus and approach are fundamentally different from SPINE. SPINE provides a trusted execution environment for applications that need to be tightly integrated with the underlying device, while I_2O provides an execution environment for trusted device drivers.

Several research projects are investing the applicability and system structure of intelligent I/O. These projects range from restructuring the main operating system, to embedded execution environments, to investigating intelligent disks.

The Piglet OS project suggests an alternative approach to intelligent I/O from SPINE. Piglet partitions the processors of a symmetric multiprocessor (SMP) system into functional groups with the goal of improving system performance for I/O intensive applications [50]. However, such processors are unavailable for user-applications. We believe, for the architectural reasons outlined in Section 2, that using an integrated I/O processor will yield similar or better performance at a much lower cost.

Perhaps the closest work to SPINE is the U-net/SLE (safe language environment) project [51]. U-net/SLE implemented a subset of the Java Virtual Machine (JVM) for the LANai; thereby allowing customized packet processing to occur on the network adapter. Because the LANai processor is slow, the primary focus of their work was on the performance of the JVM. Their preliminary results indicated that the LANai processor lacked sufficient power to support the JVM interpretation efficiently. We used Modula-3, although less popular than Java, because of the availability and portability of open, efficient compilers. In addition, we are able to take advantage of much of the SPIN work. The use of machine binaries enables us to focus on system design issues using a high-level language rather than on interpreter efficiency. The two languages are sufficiently similar that many of our results may be applicable to the Java language as well.

The IDISK [52] and Active Disks [53] projects are investigating how to add application processing to the disk subsystems. Although our work does not address disks directly, the operating system requirements for placing application processing onto an intelligent disk may be quite similar to an intelligent network adapter.

9 Conclusions

Using SPINE, we have demonstrated that intelligent devices make it possible to implement application-specific functionality inside the network adapter. Although hardware designs using “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, I/O interconnects by their very nature must be both open and enduring. These two non-technical forces alone will hinder the growth of I/O performance more than anything else will.

Our two example applications show that many extensions are viable *even with an incredibly slow I/O processor*. A faster CPU, for example, would allow the use of a virtual machine interpreter (e.g., Java), enabling transparent execution of extensions regardless of the instruction set. Using a vector processor, as suggested in [54] 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 adapter.

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 200 MHz and with a cache size of 32KB) could improve performance by a factor of five over our current system.

We hope to explore applications using more powerful network adapters. For example, Alteon’s ACEnic [55], which is equipped with two 100MHz MIPS processors, would make an excellent candidate for future research in placing application-specific functionality onto an adapter.

References

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

[2] 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 Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*. 1995.

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

[4] 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 File-server."* In *Proceedings of the Sixth Network and Operating System Support for Digital Audio Video (NOSSDAV) workshop.* 1996.

[5] T.D. Nguyen and J. Zahorjan. *"Scheduling Policies to Support Distributed 3D Multimedia Applications."* In *SIGMETRICS'98 / PERFORMANCE'98 Joint International Conference on Measurement and Modeling of Computer Systems*. 1998.

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

[7] 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."* Journal of Parallel and Distributed Computing. 1997.

[8] P. Druschel and B. Gaurav. *"Lazy Receive Processing (LRP): A Network Subsystem Architecture for Server Systems."* In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*. 1996.

[9] M.D. Dahlin, R.Y. Wang, T.E. Anderson and D.A. Patterson. *"Cooperative Caching: Using Remote Client Memory to Improve File System Performance."* In *Proceedings of the First USENIX Conference on Operating System Design and Implementation (OSDI)*. 1994.

[10] 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 Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*. 1995.

[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 Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1998.

[12] 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 Fifth Hot Interconnects Symposium.* 1997.

[13] M. Welsh, A. Basu and T. vonEicken. *"Incorporating Memory Management into User-Level Network Interfaces."* In *Proceedings of the Fifth Hot Interconnects Symposium*. 1997.

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

[15] G. Nelson, ed. *"System Programming in Modula-3"*. 1991, Prentice Hall.

[16] W.C. Hsieh, M.E. Fiuczynski, P. Pardyak and B.N. Bershad. *"Type-Safe Casting."* Software Practice and Experience. 1998.

[17] W.C. Hsieh, M.E. Fiuczynski, C. Garrett, S. Savage, D. Becker and B.N. Bershad. *"Language Support for Extensible Operating Systems."* In *Proceedings of the First Workshop on Compiler Support for System Software*. 1996.

[18] M.E. Fiuczynski and B.N. Bershad. *"An Extensible Protocol Architecture for Application-Specific Networking."* In *Proceedings of the Winter 1996 USENIX Technical Conference.* San Diego, CA. 1996.

[19] E.G. Sirer, M.E. Fiuczynski, P. Pardyak and B.N. Bershad. *"Safe Dynamic Linking in an Extensible Operating System."* In *Proceedings of the First Workshop on Compiler Support for System Software*. 1996.

[20] Lucent. *"Inferno: la Commedia Interattiva."* Available from www.lucent-inferno.com. 1996.

[21] G.C. Necula and P. Lee. *"Safe Kernel Extensions Without Run-Time Checking."* In *Proceedings of the Second USENIX Conference on Operating System Design and Implementation (OSDI)*. Seattle, WA. 1996.

[22] G.C. Necula and P. Lee. *"The Design and Implementation of a Certifying Compiler."* In *Symposium on Programming Language Design and Implementation (PLDI)*. Montreal, Canada. 1998.

[23] S. Nilsson and G. Karlsson. *"Fast Address Lookup for Internet Routers."* In *Broadband Communications: The future of telecommunications*. 1998.

[24] R. Martin, A. Vahdat, D. Culler and T. Anderson. *"Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture."* In *International Symposium on Computer Architecture (ISCA)*. Denver, CO. 1997.

[25] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich and J. Wilkes. *"An Implementation of the Hamlyn Sender-Managed Interface Architecture."* In *Proceedings of the Second USENIX Conference on Operating System Design and Implementation (OSDI)*. Seattle, WA. 1996.

[26] S. Pakin, M. Lauria and A. Chien. *"High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet."* In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference.* 1995.

[27] D.E. Culler, T.L. Lok, R.P. Martin and C.O. Yoshikawa. *"Assessing Fast Network Interfaces."* IEEE Micro. 16(1): p. 35--43. 1996.

[28] D. Anderson, J. Chase, S. Gadde, A. Gallatin, K. Yocum and M. Feeley. *"Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet."* In *Proceedings of the Usenix Technical Conference*. New Orleans, LA. 1998.

[29] L. Pryllli and B. Tourancheau. *"Bip: a new protocol designed for high performance networking on myrinet."* In *Workshop PC-NOW, IPPS/SPDP*. Orlando, FL. 1998.

[30] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards and J. Lumley. *"Afterburner."* IEEE Network. 7(4): p. 35--43. 1993.

[31] I2O Special Interest Group. *"Intelligent I/O (I_2O) Architecture Specification v1.5."* Available from www.i2osig.org. 1997.

[32] Packet Engines. *"PowerRail Enterprise Routing Switch Architecture."* Available from www.packetengines.com. 1998.

[33] G. Powers. *"A front-end telnet/rlogin server implementation."* In *Proceedings of Uniforum 1986 Conference*. 1986.

[34] S. Bal. *"Networked Systems (the architecture for multi-user computing)."* DEC Professional. 5(2): p. 48--52. 1986.

[35] D. Way. *"Front-end processors smooth local network-computer integration."* Electronics. 57(3): p. 135-9. 1984.

[36] L. Gross. *"Architecture of an Ethernet Interface."* Elektronik. 34(7): p. 124-6. 1984.

[37] D. Hitz, G. Harris, J.K. Lay and A.M. Schwartz. *"Using Unix as One Component of a Lightweight Distributed Kernel for a Multiprocessor File Server."* In *Proceedings of the Winter 1990 USENIX Conference*. 1990.

[38] P. Trautman, B. Nelson and Auspex Engineering. *"The Myth of MIPS for I/O."* Available from www.auspex.com/tr10.pdf. 1997.

[39] J. Touch and B. Parham. *"Implementing the Internet checksum in hardware."* Internet RFC 1936. 1996.

[40] V. Jacobson. *"Efficient protocol implementation."* ACM SIGCOMM'90 Tutorial. 1990.

[41] P. Druschel, L.L. Peterson and B.S. Davie. *"Experience with a high-speed Network Adapter: A Software Perspective."* In *Proceedings of the ACM SIGCOMM '94 Symposium on Communication Architectures and Protocols*. 1994.

[42] E.C. Cooper, P.A. Steenkiste, R.D. Sansom and B.D. Zill. *"Protocol Implementation on the Nectar Communication Processor."* In *Proceedings of the ACM SIGCOMM '90 Symposium on Communication Architectures and Protocols*. 1994.

[43] B.S. Davie. *"A Host/Network Interface Architecture for ATM."* In *Proceedings of the ACM SIGCOMM '91 Symposium on Communication Architectures and Protocols*. 1991.

[44] H. Kanakia and D.R. Cheriton. *"The VMP Network Adapter Board (NAB): High-performance Network Communication for Multiprocessors."* In *Proceedings of the ACM SIGCOMM '88 Symposium on Communication Architectures and Protocols*. 1988.

[45] C.B. Traw and J.M. Smith. *"A High-performance Host Interface for ATM Networks."* In *Proceedings of the ACM SIGCOMM '91 Symposium on Communication Architectures and Protocols*. 1991.

[46] M.L. Bailey, M.A. Pagels and L.L. Peterson. *"The x-chip; An experiment in hardware demultiplexing."* In *Proceedings of the IEEE Workshop on High Performance communications Subsystems*. 1991.

[47] T. Matters. *"Offloading TCP/IP to Intelligent Adapters."* Slides from PC Developers Conference. 1998.

[48] T. von Eicken, A. Basu, V. Buch and W. Vogels. *"U-Net: A User-Level Network Interface for Parallel and Distributed Computing."* In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*. 1995.

[49] K.K. Ramakrishnan. *"Performance Considerations in Designing Network Interfaces."* IEEE Journal on Selected Areas in Communications. 11(2): p. 2030--219. 1993.

[50] S. Muir and J. Smith. *"Functional divisions in the Piglet multiprocessor operating system."* In *Proceedings of the ACM SIGOPS European Workshop*. 1998.

[51] D. Oppenheimer and M. Welsh. *"User Customization of Virtual Network Interfaces with U-Net/SLE."* TR CSD-98-995, Department of Electrical Engineering and Computer Science, University of California, Berkeley. 1998.

[52] D. Patterson and K. Keeton. *"Hardware Technology Trends and Database Opportunities."* Slides from SIGMOD'98 Keynote Address. 1998.

[53] A. Acharya, M. Uysal and J. Saltz. *"Active Disks."* TR CS98-06, Computer Science Department, University of California, Santa Barbara. 1998.

[54] K. Keeton, R. Apraci-Dusseau and D.A. Patterson. *"IRAM and SmartSIMM: Overcoming the I/O Bus Bottleneck."* In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*. 1997.

[55] Alteon Networks. *"Gigabit Ethernet Technology Brief."* Available from www.alteon.com. 1998.