Appearing in: *Proceedings of the 2003 IASTED International Conference on Communications, Internet, & Information Technology (CIIT)*, Scottsdale, Arizona, November, 2003.

Toward a Common Host Interface for Network Processors[*]

Eric Hawkins
Department of Computer Science
California Polytechnic State University
San Luis Obispo, CA 93407
ehawkins@calpoly.edu

Phillip L. Nico
Department of Computer Science
California Polytechnic State University
San Luis Obispo, CA 93407
pnico@acm.org

Hugh Smith
Department of Computer Science
California Polytechnic State University
San Luis Obispo, CA 93407
husmith@calpoly.edu

Abstract

Since their invention, network interfaces have generally been treated as special by operating systems because of their complexity and unique control requirements. In this paper we present a generic host interface through which an intelligent network interface or network processor can be managed as a simple networking device. To accomplish this, we push the complex network connection management and protocol processing code down onto the network interface. This new network processing platform is treated as a simple device by the host operating system. This model of host operating system interconnection provides for various network processor architectures to be handled identically using a well-defined kernel interface. Selection of the exact location for the kernel interface to the network processor was based on our goals to maximize the utility of the network processing platform, require no changes to existing network applications, and provide interoperability with existing network protocols (e.g. TCP, UDP). This paper documents the criteria and methodology used in developing such a kernel interface and discusses our prototype implementation using Linux kernel modules and our own ASIC-based intelligent network interface card.

Keywords: Information Systems and the Internet, Operating Systems Support, intelligent NIC, network processors

1 Introduction

The concept of offloading network processing from the host processor to a separate communication processor is not a new one. It has been discussed in the literature for some time, and several vendors have emerged to fill the newly created market niches for such devices. In order for such devices to be accepted into mainstream computing, however, a general interface is needed by which operating systems can offload network processing tasks to the coprocessor without requiring a device-specific application programing interface (API) or other support mechanisms.

Traditional network interfaces have relied upon relatively dumb network adapters that simply connect the networking medium to the host I/O bus. These adapters are driven by networking code in the host operating system. To the host operating system, the network adapter looks like a small buffer into which packets can be written for transmission by the network interface card (NIC) onto the network. The Berkeley socket programming interface[8] is predominantly used by applications to send and receive data via the various network protocols supported by the operating system.

As networks have grown in size and complexity the network protocols have evolved to support these networks. Modern network protocol code is quite complex, especially protocols such as TCP/IP. Due to this growth in complexity, the processing required for transmitting and receiving data over the network has grown to a point where it is easy to justify the need for a device to offload these processing duties.[6][5] Much like the evolution of separate graphics processors was encouraged by increasing demands on graphics processing capabilities, the processing requirements of modern networking tasks are pushing the development of separate network processors for even the common network communication tasks.

Network processors provide many benefits beyond relieving host processors of common networking duties. Either through the use of general purpose CPUs or custom hardware, network processors can support auxiliary services right at the network edge. Security services such as intrusion detection or firewalling are enhanced by the physical separation of such devices from the host machine's software environment. Support for encryption can be incorporated into these devices either in hardware or software. Network quality of service mechanisms can be incorporated as well to enable multimedia applications.

Different approaches to the host-coprocessor interface have been proposed, but the most popular solution has been to use custom APIs and function libraries. The

[*]The work described in this paper was partially supported by the Department of the Navy, Office of Naval Research.

Trapeze/Myrinet project[2] has shown impressive throughput across an intelligent network adapter, but it relies upon the Trapeze API to access the network adapter. Likewise, the Nectar Communication Processor offloads protocol processing as well as application specific tasks but does so through the use of the Nectarine programming interface which provides access to the Nectar message passing facilities[1]. Network processors based on the Intel I2O specifications which utilize a split-driver model to isolate host functionality from network interface functionality are also bound to a custom API for the host-coprocessor interface[3]. Since nearly all host-network processor interfaces rely on custom APIs, the benefits of network processors have not been realized on a broad scale. Incompatibility with existing network software is a major impediment to the incorporation of these technologies.

To address the issue of binary compatibility we have defined an interface to the network processor that works along with the socket programming interface. We have developed a prototype system that uses a well-defined Linux kernel interface at the top of the protocol stack. Using Linux kernel modules we have integrated support for the Cal Poly Intelligent Network Interface Card (CiNIC)[4] into the native Linux network protocol stack. Operations on CiNIC connections are dispatched through this interface in the host operating system to the CiNIC for protocol processing. Although the initial development has been done in Linux, the requirements and architecture of the interface can be applied to any operating system that supports the socket API.

The rest of this paper is organized as follows: In Section 2 we discuss the requirements of the host-network processor interface. In Section 3 we describe the kernel level interface selection for our prototype implementation. In Section 4 we describe our implementation and prototype platform. In Section 5 we discuss directions for future work. In Section 6 we present conclusions from our work.

2 Interface Requirements

We identified several requirements for the host-network processor interface. These requirements all stem from the primary requirement that our new OS interface be compatible with existing user-level interfaces so that existing programs would not have to be altered.

Use socket API: Since the majority of legacy network application code uses the socket API, the interface to the network processor must exist within the scope of socket calls. Figure 1 shows the traditional socket call interface. The socket API sits at the top of the network subsystem and provides user programs with an interface to the operating system's network subsystem. The great majority of existing network-based programs were written to the socket API, and so to ensure compatibility with existing programs the interface to the network processor must exist within the scope of the socket API. However, this requirement means

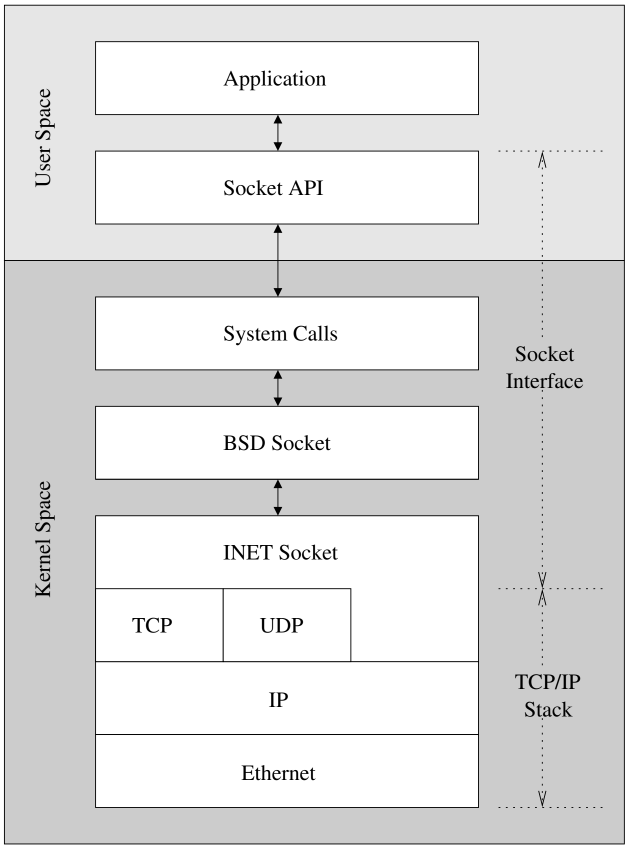

Figure 1. Socket API and OS interface.

that either the user-level socket library must be modified or support must be built into the OS. We chose the second approach since OS modification is the best way to support the existing APIs and applications.

Utilize native device management: The network processor should be managed as a normal device by the host operating system. Custom application-level code should not be required to manage the device. Rather, existing operating system support for synchronization and protection should be used. The justification for this requirement is that production operating systems use proved access management policies and procedures. With a well-designed interface to the network processor, these existing mechanisms can be utilized intact. Requiring network processors to be treated as special devices outside the scope of operating systems' existing device management facilities poses a potential security and robustness risk.

Look like a regular character device: To the host operating system the network processor should appear to be a simple character device. The device should interact with the host operating system using a minimal number of operations (open, read, write, close, etc.), and it should fundamentally act like a simple device that raw data is written to and read from. This requirement is essential to preserving the simplicity of the host-network processor interface. In implementation this requirement translates to locating the cleanest point at which to terminate host processing and invoke the co-processor. Determination of the point at which this division should be made is driven by two considerations.

First, the number of operations required for device support should be minimized in order to simplify the implementation of the interface. Second, the data structures used by the operating system for network devices must be kept consistent on both the host and the co-processor with minimum synchronization overhead. Many network related data structures are used at various points in the OS networking code and care must be taken to divide processing between the host and co-processor such that minimal replication of data is required. For example, the Linux kernel uses the *socket* data structure for filesystem information such as the inode number while the *sock* data structure is used for socket-specific network connection information. It is necessary to have the *sock* available to the co-processor since it takes care of network processing. This requires the *sock* to either be replicated between the host and co-processor or available only to the co-processor. On the other hand, the *socket* is required by the host who takes care of filesystem management. However, due to interdependencies between the data structures, separation of the two data structures would require synchronization overhead, but replication would require more.

3 The Socket Family Interface

As discussed in the previous section, the requirements for the host-network processor interface drove the design of the interface to be a kernel level modification. Several existing interfaces within the Linux kernel appeared as potential points to make the processing break between the host and co-processor.

The host-network processor interface could be implemented by intercepting all socket system calls destined for network processor connections and redirecting these calls to the co-processor. The host OS's system call table could be modified to redirect processing to functions capable of checking connections and dispatching calls appropriately. System call redirection minimizes the number of data structures requiring synchronization between the host OS and network processing platform. Also, due to the high level at which the processing division is made, system call redirection maximizes the amount of processing offloaded from the host to the co-processor. Unfortunately, the number of system calls and the requirement for catching all system calls makes this approach prohibitive in terms of implementation and execution overhead. The mapping mechanisms required to maintain network connections across multiple processes would also be complex and costly.

Another possible OS interface is the Virtual Filesystem Switch (VFS). The VFS is a software layer within the kernel that handles system calls related to a filesystem. It provides a common interface to various filesystems through specification of the functions and data structures that must be implemented to support a particular filesystem. The VFS seems like the natural spot to break host processing since it would allow network processor support to be implemented as a new filesystem type. Operations destined for the co-processor would be redirected through the VFS interface and handled on the co-processor. However, the implementation of OS socket call handling makes the VFS an inappropriate point for interfacing to the co-processor. The primary reason for this is that not all socket processing proceeds through the VFS. The *socketcall* multiplexer is actually a parallel data path to the VFS through which network operations can alternately be invoked. For example, to receive data from a socket, an application can make a *read* call, which is handled by the VFS implementation of the *read* system call. Alternately, an application can make a *recv* call on a connected socket, which is handled by the *socketcall* multiplexer and does not interact directly with the VFS.

The socket protocol family interface is a well-defined kernel interface just below the VFS layer. All socket processing converges from the VFS and *socketcall* multiplexer at the protocol family interface where it is dispatched to particular socket implementations. In the native networking code, this interface allows for the implementation of different socket types or protocol families. For example, with Internet domain sockets using the Internet Protocol (IP), this interface redirects socket processing to the set of data and operations defined for the Internet protocol family (PF_INET). The protocol family interface provides a narrow point at which to terminate host processing and invoke the co-processor on behalf of socket operations.

Using the socket protocol family interface, we have implemented a new type of socket family to be used with our network processor. We have named our protocol family PF_CINIC, since our prototype implementation utilized the CiNIC as previously mentioned. Figure 2 shows the software architecture of the network processor interface using the PF_CINIC protocol family alongside the host's native protocol families and network stack. Implementation of the PF_CINIC protocol family requires a minimal number of functions to be implemented (17 to be exact) due to the fact that the various possible data paths for socket operations converge at this point into the fundamental socket operations (e.g. *create, release, connect, bind, sendmsg, recvmsg,* etc).

Another advantage of making the break in host processing at the protocol family interface is that it provides a low-level view of the socket based only on the kernel data structures. Integration of the socket into the filesystem is handled by the kernel at a higher level, so all of the filesystem maintenance operations such as allocating and maintaining file descriptors for sockets are automatically taken care of. This allows the network processor to function as a true networking device without the overhead of filesystem operations, which would be required if host processing

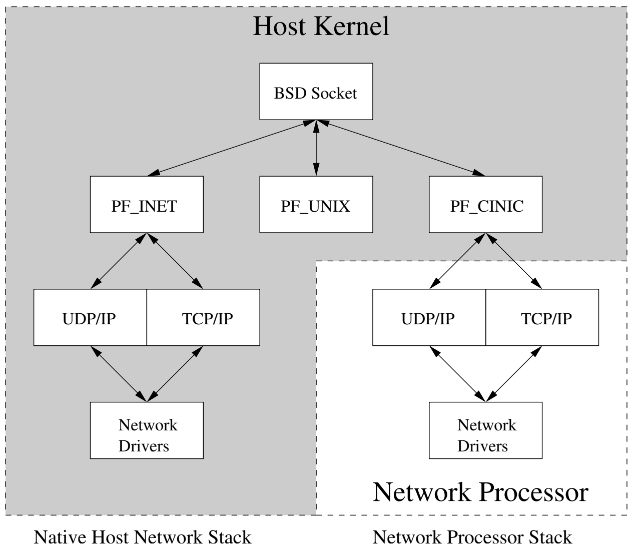

Figure 2. Protocol family interface

was terminated at a higher level. The low-level view of the socket at the protocol family interface also limits the set of data structures affected by kernel operations on both the host and co-processor, providing for a clean separation of data between the host and co-processor with minimal synchronization requirements.

Breaking the host network processing at the protocol family level allows multiple network protocols to be supported by the co-processor. Protocol such as TCP/IP and UDP/IP are implemented at lower levels in the operating system, so processing destined for different types of sockets can proceed through the PF_CINIC interface and be switched based on the type of connection at lower levels. This approach stands in contrast to other so-called *offload engines* that can only support specific protocols.

4 The CiNIC Prototype Implementation

Our prototype implementation utilizes our Strong-ARM processor based CiNIC for the network co-processor. It runs a full Linux 2.4 operating system and is connected to the x86 Linux 2.4 host computer through a PCI-to-PCI bridge. A shared memory interface between the host and co-processor provides communication between the two platforms. Figure 3 shows the development platform.

Socket operations destined for CiNIC connections are intercepted at the protocol family layer, just at the top of the network protocol stack. The implementation of the PF_CINIC interface uses Linux loadable kernel modules, which are loaded prior to network processor usage much like a standard device driver. When host processing reaches the PF_CINIC interface, a communication packet is built with the necessary data and control information to transfer processing to the co-processor. This communication packet is then placed onto shared memory and a data ready

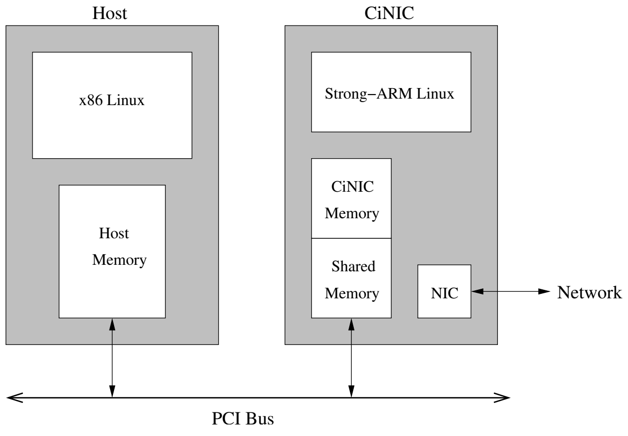

Figure 3. CiNIC development platform

flag is marked. The host process is then put to sleep until the call returns from the co-processor. Figure 4 shows the shared memory communication protocol and threading architecure for the host-network processor interface.

A kernel thread on the co-processor is responsible for retrieving the communication packet from shared memory. The current implementation uses a polling protocol in which a kernel thread constantly checks the data ready flag in shared memory to see if a communication packet is ready for handling. When a communication packet is ready, this thread moves it out of shared memory and clears the flag. The communication packet is then placed on a wait queue to be handled by another thread responsible for maintaining socket connections and dispatching processing on each socket to worker threads. This *handler thread* allocates and deallocates socket data structures. It maintains a mapping of host sockets to co-processor sockets so that subsequent socket calls on a pre-existing socket proceed through the proper connection. The *handler thread* also manages a pool of *worker threads*, which are responsible for all socket processing other than creation and destruction. These *worker threads* pick up where processing left off on the host, calling functions within the native network family implementation.

When the co-processor completes its work on behalf of a socket call, all return data and control information are in the associated communication packet, which is placed onto shared memory for the host to pick up. The host utilizes a kernel thread that polls shared memory for returning packets. When a communication packet arrives, the host thread pulls the packet out of shared memory and wakes up the associated sleeping process. The native host process resumes where it left off and the socket call returns.

5 Future Work

Several optimizations are possible for the host-network processor interface as well as for our prototype platform. We are currently working to reduce the number of data

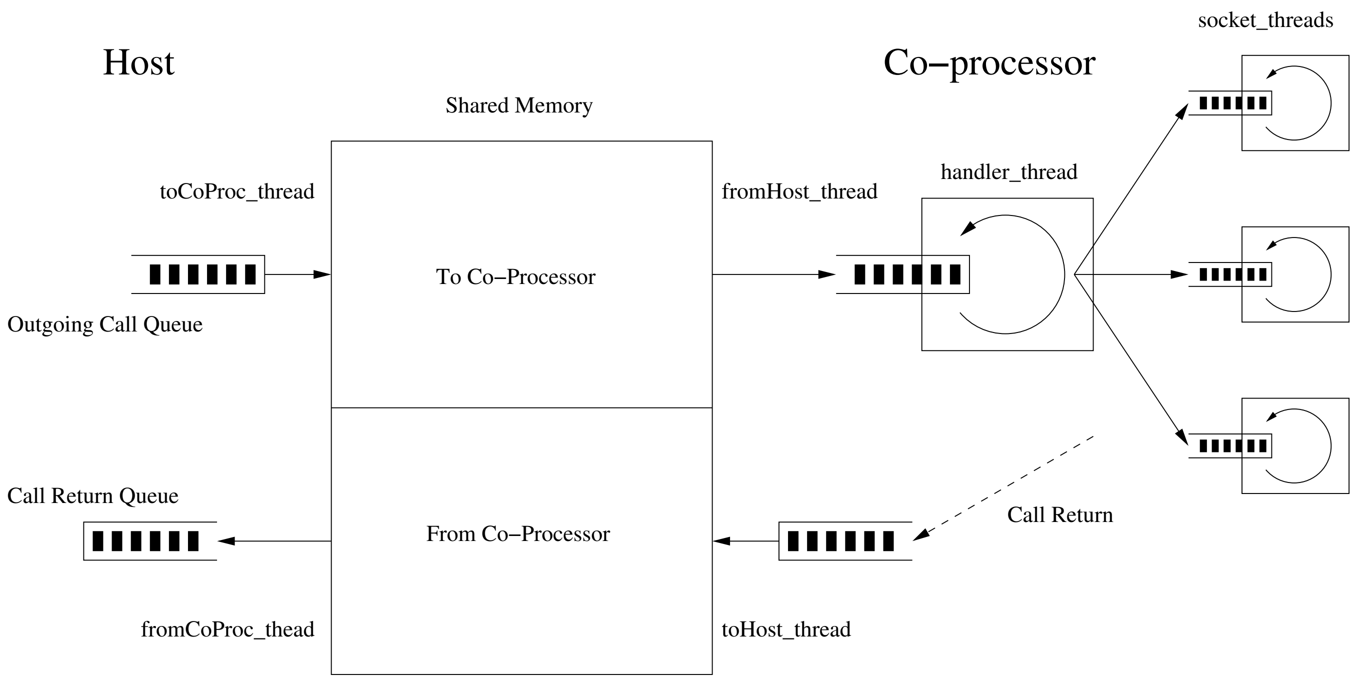

Figure 4. Shared memory communication architecture.

copies required for communication between the host and co-processor in our development platform. The current implementation requires data to be copied from the host to shared memory and from shared memory to the co-processor (and vice versa). Since data copy operations are a major bottleneck in network processing[7], we need to reduce the number of copy operations in order to get reasonable performance. However, unlike other performance-oriented research on network processors, our goal is not to enhance overall network performance, rather to provide a standard interface to the network processor through which the network processor can provide various processing tasks.

Eventually, we plan to investigate how our Linux implementation of the host-network processor interface ties into the structure of networking code in other operating systems. We expect that other Unix implementations should coincide fairly well with the Linux implementation. The correlation to other proprietary operating systems may not be so close.

We are also developing a next-generation CiNIC using a FPGA design with an embedded soft-core processor running Linux. This future platform will provide us with many hardware capabilities beyond that of the current Strong-ARM platform such as the ability to create auxiliary processing blocks for special purposes. Along with development of our next-generation hardware, we plan to move from a polling communication protocol between the host and co-processor to an interrupt-driven communication protocol. This approach will be facilitated by the new hardware and will relieve both the host and co-processor from supporting the busy-waiting kernel threads used in the polling protocol.

6 Conclusions

We have described the implementation of a host-network processor interface that relies upon the traditional socket programming API. We implemented the interface in kernel space using loadable Linux kernel modules. The selection of the network protocol family fulfilled our design requirements for the host-network processor interface by providing a narrow point at which to terminate host processing. This interface allows network processing to proceed on the outboard platform with minimal synchronization overhead, and allows the network processor to look like a simple device to the host operating system.

References

[1] COOPER, E. C., STEENKISTE, P. A., SANSOM, R. D., AND ZILL, B. D. Protocol Implementation on the Nectar Communication Processor. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols* (1990), ACM, pp. 135–144.

[2] GALLATIN, A., CHASE, J., AND YOCUM, K. Trapeze/IP: TCP/IP at near-gigabit speeds. In *Proceedings of the USENIX '99 Technical Conference* (June 1999), pp. 109–120.

[3] GUTLEBER, J., AND ORSINI, L. Architectural software support for processing clusters. In *Proceedings*

of the IEEE Int'l Conference on Cluster Computing (2000), pp. 153–161.

[4] HATASHITA, J., HARRIS, J., SMITH, H., AND NICO, P. An evaluation architecture for a network coprocessor. In *Proceedings of the 2002 IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)* (November 2002).

[5] KAISERSWERTH, M. The Parallel Protocol Engine. *IEEE/ACM Transactions on Networking 1*, 6 (1993), 650–663.

[6] KANAKIA, H., AND CHERITON, D. R. The VMP network adapter board (NAB): High-performance network communication for multiprocessors. In *Proceedings of Sigcomm-88* (1988), pp. 175–187.

[7] KLEINPASTE, K., STEENKISTE, P., AND ZILL, B. Software support for outboard buffering and checksumming. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (1995), pp. 87–98.

[8] STEVENS, R. W. *UNIX Network Programming*. Prentice Hall, Englewood Cliffs, NJ, 1990.