An Operating System Architecture for Network Processors

Steve Muir
Princeton University
smuir@cs.princeton.edu

Jonathan Smith *
University of Pennsylvania
jms@cis.upenn.edu

ABSTRACT

Network devices have become significantly more complex in recent years, with the most sophisticated current devices incorporating one or more general-purpose CPUs as part of their hardware. The need for such processing capability is motivated by the desire to move greater amounts of functionality, of ever-increasing complexity, from the host CPU to the network device itself. A significant challenge in doing so is managing the complexity of the software running on the network device.

We believe that the complexity of this software has reached the point where it is now on a par with many general-purpose systems, and thus requires the same management infrastructure—an operating system for network processors.

In this paper we describe an architecture for such an OS, presenting the features most relevant to network processors and describing similarities to and differences from a general-purpose OS. We present a prototype implementation using an SMP system as a virtual network processor, and show how our prototype was used to evaluate a novel user-space interface to a network device.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Design, Experimentation, Performance

Keywords

Operating Systems, Network Processors

*Jonathan Smith is currently on leave from Penn at DARPA. This document is Approved For Public Release, Distribution Unlimited.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ANCS'05, October 26–28, 2005, Princeton, New Jersey, USA.
Copyright 2005 ACM 1-59593-082-5/05/0010 ...$5.00.

1. INTRODUCTION

Network interface cards (NICs) have evolved significantly in recent years, from relatively simple fixed-function devices into sophisticated embedded systems with multiple general processing units, reasonably large amounts of memory, and user-modifiable firmware. Unfortunately, while these devices have increasingly come to resemble general-purpose computer systems in terms of hardware scale, flexibility and complexity, the software environment has not kept pace with that evolution.

We believe that a significant step forward in the reliability and usability aspects of complex, advanced NICs could be accomplished by adopting a standard NIC operating system that would provide researchers and engineers with a device-independent software environment for developing and deploying new NIC software. Such an environment would promote many of the beneficial practices applied to general-purpose systems, such as code portability and reuse. Additionally, we expect that many of the gory details of implementing software for particular devices would be hidden from developers, thus enabling faster prototyping of new services and applications.

1.1 Advanced NIC Architectures

Although the majority of NICs used in commodity general-purpose systems are still relatively simple devices, advanced NICs with sophisticated processing capabilities have become popular in a number of specialised areas.

High-speed NICs, such as Gigabit adapters (e.g., the 3Com *3C2000-T* Gigabit NIC), typically include on-board processing capabilities in order to handle very high packet rates without imposing undue load upon the host processor. In most cases though the processing performed by the NIC CPU is relatively simple, such as offloading of checksum calculation and prioritisation of outgoing traffic. Nevertheless, an underlying OS would allow such features to be easily added to new or existing devices without software having to be reimplemented for each specific device.

Another area where sophisticated NICs have become commonplace is in router line cards. A good example is the Intel *IXP1200* line of network processors, which combine an ARM controller with a number of special-purpose packet engines, processors with an instruction set specifically designed for high-speed packet forwarding. The complexity of this system makes development of software for the device complicated, and requires sophisticated scheduling to extract maximum efficiency from the device [28]. An OS that

hides the hardware complexities of such a device from software developers would presumably be of significant value.

A third class of advanced NIC is the remote system management card, sometimes known as a *remote integrated lights-out (RILO)* adapter. These devices permit system administrators to connect to this dedicated NIC and perform various tasks e.g., access the system console, obtain hardware monitoring information, or reboot the system, even when the host system is overloaded and/or malfunctioning. Furthermore, such capabilities are also being integrated into other NICs, where presumably such integration would be straightforward if both devices provided a common software environment.

These examples show how sophisticated NICs are becoming more commonplace, a trend that we believe is likely to continue as higher-speed network connectivity is pushed out closer to the network edge. Furthermore, as discussed in Section 6.2, the increasing availability of multiple processing elements—multi-threaded and/or multi-core CPUs—in commodity systems raises the possibility of *virtual network processors*.

1.2 NIC Services/Applications

In order to better understand the requirements of an OS architecture for advanced NICs, it is useful to consider the types of functionality that is already implemented on such devices, or may be in the future. Although the basic function of any advanced NIC is the same as that of a cheap, simple Ethernet card—manipulation of network packets—the variety of functions that have been added to these advanced NICs is quite broad. However, the basic goal of all of these designs is to reduce the load imposed upon the host system, both CPU processing and data transfers.

One seemingly obvious application of an advanced NIC is to offload some or all of the network protocol stack from the host CPU—this has been tried by various groups, but the results have been mixed and so it remains unclear as to the merits of such an approach [21, 11].

A second use of an advanced NIC is to filter outbound and/or inbound network traffic. Filtering and/or traffic shaping of outbound traffic can be used to enforce certain network policies, while filtering of incoming traffic can limit the load imposed upon the host system e.g., reducing the number of interrupt requests to eliminate the threat of receive livelock. When combined with higher-level functionality a NIC supporting filtering in this manner may provide an attractive platform for a network firewall device.

A related function of an advanced NIC is to monitor and log network traffic. Software running directly on a NIC may be better capable of analysing packets at high-speed than software running on the host CPU that would require every packet be transferred over the system bus to the host's memory. One such example, implemented in the host OS, is the *ksniffer* [24] kernel traffic monitor that gathers server-side statistics on client-perceived response times for web pages.

Finally, as described above, specialised NICs may be used to support remote management of systems. Many such devices are now offering greater functionality, such as a remote console, that require more sophisticated software on the management NIC, and additional functionality e.g., sophisticated user authentication, system event logging, etc., is likely to be added in the future.

1.3 Motivation for a NIC OS

The increased complexity of network devices, in particular the presence of on-board general-purpose CPUs and the increasingly sophisticated tasks performed by such devices, leads us to conclude that such devices have reached a point where an operating system is not only required to manage complexity, but can also provide the same benefits derived from the presence of an OS in a general-purpose computing environment. We consider the most important benefits, at least in the NIC environment, to be:

- **Hardware Abstraction** – OS exports an API that hides the low-level details of various operations from higher-level software.
- **Multitasking** – OS supports multiple applications running concurrently.
- **Isolation** – OS isolates applications from each other and from the OS itself, protecting against malicious or buggy code.
- **Common Application Services** – OS provides a set of commonly-used application services e.g., dynamic loading, logging, debugging.

One possibility for providing this feature set in the NIC software environment is to run a standard general-purpose OS, such as Linux or a BSD variant, on the NIC processor(s). We considered this approach but opted against it for reasons described in the following section.

The remainder of this paper describes a generic (device-independent) OS architecture, the *Lightweight NIC Kernel (LiNK)* that provides these capabilities to software running on a typical advanced NIC. After presenting the LiNK architecture we show how we implemented a prototype system on commodity PC hardware, and used that system to evaluate an API for direct communication between the NIC and user-space applications.

2. LINK: A LIGHTWEIGHT NIC KERNEL

We begin by presenting a high-level OS architecture that we believe to be particularly well-suited for use on network processors. Since this OS is focused specifically on the NIC environment, and is intended to be relatively simple and low-overhead, we coin the term *Lightweight NIC Kernel (LiNK)* for this architecture.

Although we considered the possibility of running an existing general-purpose OS on the NIC processor(s), we opted not to pursue such a path because of the relative complexity and size of such systems. Another possibility that may deserve further consideration is the use of an special-purpose embedded or real-time OS, such as VxWorks or QNX. However, we believe that even in such an environment it may be desirable to model the LiNK architecture as a thin portability layer atop the base OS, in much the same way that various *portable runtime libraries* provide OS-independent software environments for general-purpose systems.

2.1 Features of LiNK

Many of the goals of our NIC kernel are similar to those of a general-purpose OS: providing a hardware abstraction

layer, supporting multiple concurrent applications, and facilitating easier development and debugging of software. However, we are able to take advantage of the specialised environment in which our kernel will be used to refine this basic set of requirements somewhat. For example, the applications to be run on top of the kernel all fit into a simple programming model (see Section 2.2), thus simplifying the memory management and scheduling requirements of the kernel.

The basic features of LiNK are as follows:

- Simple kernel structure: the kernel is organised as a non-preemptible core, where events are queued up for processing in response to external requests. Hardware interrupts are treated as just another type of request i.e., processing is queued with other events, although the kernel may support some notion of higher priority events. Since each event typically is processed to completion, the need to protect critical code sections from other events is eliminated in most cases, the exception being high latency events that explicitly manage their own scheduling.

- Cooperative multithreading: this structure supports multiple applications running concurrently on the NIC processor(s), where each application consists of a sequence of non-preemptible events. Complex applications with long-running functions must be explicitly structured to yield the processor to the main event loop frequently. While such cooperative multithreading is unpopular in a general-purpose system, we believe it to be an acceptable compromise (between the complexity of preemptive multithreading and the inflexibility of a single-threaded system) in a special-purpose environment.

- Simple memory protection: the value of memory protection between components in a complex system is beyond question, but LiNK need not offer a full-featured virtual memory system as is typically found in a general-purpose OS. In particular, we need not support paging of memory, nor do we believe it necessary to separate code into distinct privilege levels e.g., user/kernel. Instead, we propose a simple protection domain scheme, similar to that in the *Nooks* [30] system for Linux device drivers, whereby each kernel component runs in a separate memory region, thus providing protection against memory access bugs.

- Software development support: since a primary goal of LiNK is to simplify the software development task for NIC applications, the kernel should provide those features that assist engineers and researchers in developing new software. This includes a flexible logging facility with the ability to modify logging options (verbosity level) while the system is running, and a dynamic loading scheme so that individual components can be updated without having to disrupt the basic operation of the NIC.

In addition to these high-level design features, LiNK also defines two programming abstractions that are integrated into the kernel structure and provide the basic programming model for implementing new NIC software components and host applications. Components are implemented according to the *network service component* model, which provides a more structured unit of processing than the standard process or thread models, and *posted service requests* are defined as the basic mechanism by which the LiNK communicates with external clients, such as host applications or other devices.

2.2 Network Service Component Model

Network service components assume a role in LiNK analogous to that of processes in general-purpose UNIX systems i.e., they represent the fundamental unit of processing, and have an associated set of properties. However, the type of functionality embodied by a service component means that we can define them in a more structured manner than processes. LiNK defines three different functions that are provided by a service component, although in some cases one or more may be unused, and executes each function in certain circumstances:

1. An *Rx handler* is executed by LiNK whenever a network packet is received by the NIC that is determined to be associated with the service component. LiNK uses standard packet filtering techniques [18, 10, 2] to demultiplex incoming packets according to filters registered by each service.

2. A *Tx handler* is executed by a LiNK whenever an outgoing network packet that is associated with the service component is received from the host system (or possibly generated by another service). Tx handlers are explicitly associated with network flows, which are created by the host or other services when they wish to send packets of a particular type, and may be composed into linear sequences.

3. A *timer callback* is scheduled by LiNK, according to the service's needs, in order to perform periodic tasks e.g., purging of expired ARP cache entries, state timeouts.

Although the network service component model is simple, and appears fairly restrictive, we have found that it fits very well into the programming model when building new NIC functionality. Simple examples include an ARP service that uses the Rx handler to respond to ARP requests, Tx handler to add MAC-level headers to outgoing packets, and timer callbacks to flush the ARP cache, and a traffic shaper service that uses the Tx handler to enqueue outgoing packets in the appropriate traffic queue, and the timer callback to implement transmit scheduling.

The primary advantage of using a simple service component model as a framework for encapsulating LiNK components is that each function—Rx, Tx or timer—is only called when a well-defined event occurs, thus making the kernel's task of scheduling very simple, and each function runs to completion (or an explicit yield point), eliminating the need to asynchronously save and restore context. In this regard service components are similar to cooperatively-scheduled threads.

2.3 Posted Service Requests

The second abstraction that is part of the LiNK architecture is use of *posted service requests (PSRs)* as the communication API between the NIC and other system components, either the host system and its applications, or other hardware devices. The PSR mechanism is designed to be simple

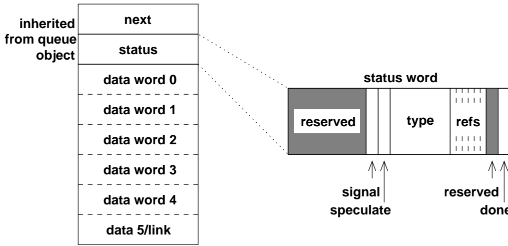

Figure 1: Structure of the LiNK Posted Service Request

and low-overhead while providing a greater degree of flexibility than similar protocols.

The basic structure of a PSR is shown in Figure 1. Clients send requests to the LiNK by enqueuing a sequence of PSRs onto a request queue; request queues may be per-client or shared by many clients, and the exact details of how the queue is maintained are implementation-specific. To give a concrete example of how this may be done, in our initial prototype each client was provided with an array of PSRs in a shared memory region that could be chained together then appended to a request queue using a multiple-writer/single-reader non-blocking operation—in this way we eliminated the need for synchronisation operations between the client and LiNK.

Each PSR is designed to be compact enough to fit into a single cache line. The first two words of the structure are inherited from the shared queue structure, although only the *done* bit is reserved by the shared queue protocol. The details of the shared queue protocol are omitted from this paper but can be thought of as a singly-linked list with a *done* bit used to indicate when the reader has processed an element, implying that the writer can advance the tail pointer.

The remaining six words in the PSR are used to pass data between the client and LiNK, with the last word also being used in some circumstances as a link pointer. The functions of the various unreserved bits in the status word are as follows:

- *Reference bits (6)* are used to indicate whether the content of the corresponding data word is a value or reference. References are used by a client when constructing sequences of PSRs, their use is described in more detail later.
- *Type field* is used to specify the operation being requested by the client.
- *Speculate bit* specifies whether the processing of this PSR is conditional upon the success of a preceding PSR operation. If so then the *link* pointer (word 5) points to the preceding PSR.
- *Signal bit* is set when the LiNK should send a signal to the client upon completion of this operation.

The simplest examples of a PSR are created by setting the type field to indicate the operation being invoked and passing the values of the operation's parameters in the data words. After processing the PSR the kernel stores a return code and other operation-specific data in the data words for subsequent retrieval by the application.

2.3.1 Asynchronicity Exploits Concurrency

While the use of PSRs appears similar to a function-call interface, the biggest difference arises from the nature of the shared queue object to which PSRs are posted. Because the queue is merely a container for elements the act of posting does not immediately initiate LiNK processing of the request. If the client knows that the LiNK will automatically check the queue for new requests within an acceptable time period e.g., when the LiNK is operating in a polling mode, then no further action is necessary. Otherwise it may send an explicit signal to request that the queue be processed.

An asynchronous programming model has both advantages and disadvantages when compared to a conventional synchronous interface. The primary advantage is that the decoupling provides implicit concurrency between the execution of the application and the kernel. However, the application programmer is faced with an unfamiliar programming model and must restructure their application to obtain maximum benefit from this concurrency.

We observe that although the asynchronous model is unfamiliar to application programmers used to RPC types of communication, it is exactly analogous to the method of communication used by device drivers to interact with hardware, and so is not difficult for users familiar with such concepts to grasp. Furthermore, as part of our prototype implementation we built a simple library that hides asynchronicity from application programmers in order to provide compatibility with existing applications, albeit with a reduced degree of concurrency between application and the LiNK device.

A practical side-effect of having an asynchronous service invocation mechanism is that notification of the success or failure of an operation is no longer a simple matter of returning the appropriate code since the application is unlikely to be waiting for that return value. While LiNK does update each PSR with a completion code which the client can poll to determine when the operation has completed and its outcome i.e., whether an error occured, the LiNK can also be configured to use asynchronous signals, analogous to UNIX signals, to indicate operation completion. Normally a signal is only sent to the client if an operation fails, but the *signal* bit can be set to request a signal even if the operation is successful. This facility can be used by a client to block until the PSR completes if it cannot proceed without the result.

2.3.2 References Arguments and Speculation

This programming model would perform very poorly if a client had to post one PSR and wait for the results it generates before being able to use them in the next PSR. One simple way of increasing the performance of the system is to allow independent requests to be batched together, so that the overhead of signaling between the client and LiNK is amortised over multiple messages; such a system was described by Freimuth et al. [11]. We take a similar but more flexible approach, adding two features to the basic mechanism: reference arguments and speculative processing.

The *reference bits* in the status word allow the client to designate particular arguments as being references rather than values. A reference argument specifies a data word of another PSR as the target of the reference, which is deref-

erenced by the LiNK when it processes the referring PSR. Reference arguments permit one PSR to use values which were unknown to the application at the time it posted the PSR to the client page. An example of this usage occurs when one PSR allocates a LiNK resource e.g., a network buffer, which is then used by a subsequent PSR—the capability reference for the kernel resource is created during processing of the first PSR so the value is not known to the application at the time that it posts the second.

Speculative processing is used by an application to notify the kernel that one PSR is dependent upon another—if the first was not processed successfully then the second should be ignored. Speculation is useful in two distinct situations: when the second PSR is contingent upon either a state change or output produced by the first, and when the application does not wish to perform the second operation if the first did not succeed, even though it could. This latter case arises when two operations refer to a sequence of data—if the first packet transmit request failed (say) then there is no point transmitting the second packet.

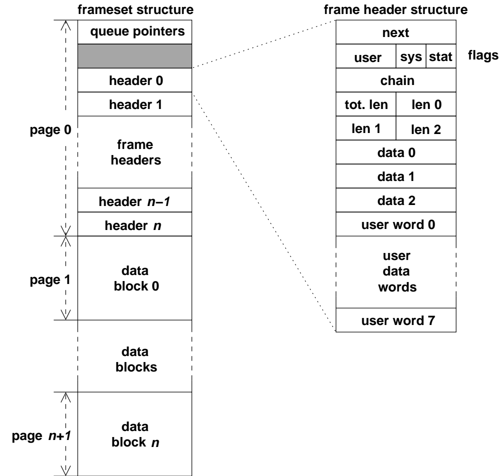

Figure 2: Frameset and user-level frame header structure

3. PROTOTYPE IMPLEMENTATION

We developed a prototype implementation of LiNK that runs atop commodity SMP hardware. Our prototype, which dedicates one CPU in the system to running LiNK, not only serves as a simple, flexible demonstration of the LiNK architecture, but also shows how one might leverage multi-core and/or multi-threaded CPUs to gain increased system performance. We refer to this system architecture as a *virtual NIC*, since the combination of a general-purpose CPU and a standard NIC appears to the host OS as a NIC with sophisticated processing capabilities. Since we originally took this approach only as an expedient way of implementing a LiNK prototype, we defer discussion of the merits of such an architecture for future systems until Section 6.2.

Our prototype represents a full implementation of the LiNK architecture described in the previous section, with the only omission being support for memory protection. It includes network service components for ARP and ICMP, as well as TCP/IP multiplexing and demultiplexing (packet filtering) component to support host protocol stack implementations, and a Virtual Clock-based [33] traffic scheduler.

The implementation was based upon an early Linux SMP kernel (the Linux 2.0.x kernel series). Although the base kernel is obviously far from state-of-the-art, we do not believe that any deficiencies in that base adversely affected the behaviour of the LiNK prototype. LiNK is initialised by loading the LiNK module into an already running kernel, at which point it acquires one of the CPUs in the system for its exclusive use. Subsequent communication with the host Linux kernel, and support for a direct user-space API, described in section 4, are provided using a shared memory interface. Since the LiNK CPU is just one CPU in the SMP system, the high-bandwidth, low-latency memory subsystem enables certain communication mechanisms that may not be suitable for a more conventional device interface, where the LiNK runs on a NIC that communicates with the host system over a slower I/O bus. We explore the details of our shared memory communication later in Section 3.2.

3.1 Polling vs. Interrupts

One implementation decision we made early on in development of our prototype was to take advantage of the dedicated LiNK CPU by making the kernel using polling exclusively rather than interrupts for communication with other CPUs and devices. While this greatly simplified the kernel implementation, and eliminates the possibility of *receive livelock* [22], our subsequent experiences revealed two distinct drawbacks to such an approach. We believe that a hybrid approach, such as *clocked interrupts* [27], where the system switches from its normal interrupt-driven mode into a periodic polling mode when under heavy load, is a more reasonable approach for future implementations.

The first problem we encountered was an engineering challenge—many devices are not intended to be used in a polled manner, but instead are designed with the assumption that the host OS is interrupt-driven. While it was possible to reengineer the driver for the particular NIC used in our test system to support our needs, it has been brought to our attention that this is not possible for all network devices. The second problem became obvious when running simple microbenchmarks on our test system: the cost of polling a device attached to a relatively slow I/O bus is very expensive in terms of CPU cycles, even on our modest test system. For these reasons we plan on adapting our LiNK prototype to be interrupt-driven, but probably in a similar manner to the Nemesis kernel [16] wherein interrupt handlers are scheduled just like other events. A valuable side-effect of doing so is that we should be able to incorporate standard Linux device drivers into LiNK relatively easily.

3.2 Network Packet Framesets

Since packet handling is the primary task of our virtual NIC we extended the LiNK with specialised versions of the PSR queues to handle packet transmission and reception. These packet queues are called *framesets*, each frameset representing separate outgoing and incoming sequences of packets. Framesets have the same basic characteristics as PSR queues but each element—a *frame*—has a structure specifically designed for packet processing.

Clients obtain framesets from LiNK using the posted service request mechanism, subsequently the frameset itself is used as a shared queue for packet transmission and reception between the client and LiNK. Each frameset con-

sists of internal control fields (not shown), plus two shared memory queues that hold transmitted and received frames. Each frame, comprising a *frame header* and a small number of possibly non-contiguous data buffers, represents a single network packet; LiNK does not perform fragmentation of frames into packets but requires that the client performs both fragmentation and reassembly of packets. The structure of both framesets and frame headers is depicted in Figure 2.

One design decision made early on that we found to have tremendous benefit was incorporating support for multiple non-contiguous data buffers into the frameset structure. This decision was enabled by making an explicit assumption that all current and future network devices will support DMA from a reasonably-sized set of such non-contiguous buffers, thus obviating the need to reconstitute outgoing packets into a single contiguous region. This also simplified the task of managing distinct client and LiNK buffers without incurring additional copying costs.

Only the client-visible part of the frameset structure is shown in Figure 2. The queue write pointers—tail pointer for the transmit queue, head for the receive queue—are stored at the beginning of the structure; the complementary pointers i.e., transmit head and receive tail, are required only by the LiNK, so are not made accessible to the client. Following some miscellaneous frameset state fields, are n frame headers each with the structure shown on the right side of the figure. Finally, an array of n pages is placed after the headers, one page per header.

Each frame header consists of a number of fields. The *next* pointer is inherited from the PSR structure described earlier, but frame headers use a status byte in the flags word rather than a single *done* bit; the function of this status byte is described below. System flags are reserved for LiNK use, while the user flags field is available for application-specific purposes. The chain pointer is used to create linked lists of frame headers—this is a common enough occurrence in protocol stack implementations e.g., a TCP retransmit queue, IP refragmentation queue, etc., to merit a dedicated field. Four half-word sized length fields indicate the total length of the packet and the length of each of up to three data blocks, pointers to which are stored in the final three words. The ability to specify three non-contiguous data buffers is used to perform *scatter-gather DMA* when transferring data from/to the NIC.

As a frame is transmitted or received it undergoes various state changes. In order to simplify the description of the state transitions the remainder of this discussion focuses on packet transmission only, but packet reception is handled similarly. The frame's current state must be exposed to both the LiNK and the associated application so that it can manage its use of the headers. Figure 3 shows the state transition diagram associated with a frame. Frames can be in one of 7 states, represented as circles, encoded as the value x shown within the circle. All valid transitions are shown on the graph: the text label associated with each transition indicates the action which triggers the transition, with *Tx-* and *client-* prefixes indicating actions performed by LiNK and client respectively, and a C-syntax expression indicating how the state variable is atomically modified by that action; *cmpxchg(x, Tx-ready, skipped)* means "if the current value of x is *Tx-ready* then change it to *skipped*, otherwise do nothing", executed atomically. The meaning

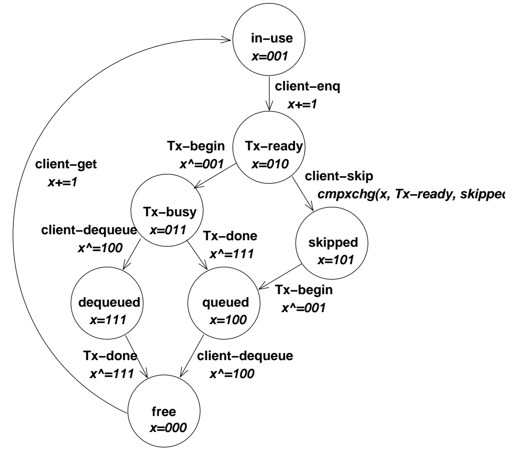

Figure 3: Frame header state transitions

of each action is as follows:

- *client-get:* mark a free frame as being in-use by the application, primarily for the client's own bookkeeping purposes.
- *client-enq:* enqueue a frame which has been prepared by the application, this indicates to the LiNK that the frame is ready for transmission.
- *Tx-begin:* begin transmitting the frame.
- *client-cancel:* skip processing if not already begun, typically used by the client to abort transmission of an enqueued frame.
- *Tx-done:* frame processing complete.
- *client-dequeue:* remove frame header from queue, only applied once at least one other frame has been enqueued.

Most of the states themselves are self-explanatory, however the steps by which a frame moves from *Tx-ready* to *free* are worth explaining. Consider a frame being transmitted by an application: after being enqueued on the transmit queue and subsequently selected by the LiNK for transmission the *Tx-begin* action is used to indicate that the frame is being processed e.g., is being transferred by DMA to the network device. In this state the frame header structure is being used by both the network device, to conduct the DMA transfer, and the queue reader and writer(s), as the head element. An artifact of the particular non-blocking queue algorithm we used is that there must always be at least one element in the queue, which complicates the state diagram as follows.

One of two possible events can occur next: if processing completes before another frame is enqueued then *Tx-done* is used to indicate that the network device no longer needs access to the frame—however the application cannot reuse this frame yet since it still constitutes the head of the queue. Alternatively, another frame could be enqueued, permitting the *client-dequeue* operation to be performed—in this state though the device is still accessing the frame, so again it cannot be reused yet. Only when both *Tx-done* and *client-dequeue* have been performed is the frame available for reuse.

The important point to observe about the state transition diagram is that the actions taken by both the client and LiNK require no knowledge of action about to be taken by the other party—this property is guaranteed by careful selection of the binary representation of the current state and the choice of atomic operation corresponding to each action; note that we assume the client follows the protocol correctly. For example, once the LiNK observes that the head frame is in the *Tx-ready* state it always applies the *Tx-begin* action—if the client should apply *client-skip* in-between the observation and subsequent action the state of the frame is still modified correctly. The LiNK subsequently checks whether the state is now *Tx-busy* and if so begins transmission.

Frames are transmitted by enqueuing them onto the associated frameset's Tx queue, which is frequently checked for updates by the LiNK *frame multiplexer*, either periodically in polling mode or when explicitly requested by an inter-processor interrupt (IPI). When the multiplexer determines that a new frame has been enqueued it creates a shadow frame header by copying the frame header into LiNK memory inaccessible to the client. This step is required to prevent modification of the frame header fields e.g., source IP address, once the frame is passed to the device, and thus is not required when the client is trusted e.g., the host OS system or a LiNK-internal client such as the ICMP service.

After creating the shadow frame header and attaching the data buffers the frame is passed to the service Tx handlers (see Section 2.2) associated with the frameset. These can perform various manipulations upon the frame but at a minimum create the appropriate headers for transmission of the frame e.g., TCP, IP, and link-level. To prevent modification of packet headers by the application program these headers are created in LiNK memory, optionally using parameters passed in the frame headers *user data words*, for example the TCP sequence number and source/destination ports. The LiNK uses gather DMA when transferring the packet to the NIC to avoid having to copy the packet headers and payload into a contiguous physical memory buffer.

Service components also perform the task of packet scheduling. Such blocks attach a transmit timestamp to each frame which is then used by the frame multiplexer to select one frame per physical output device from the set of framesets associated with that device. This frame is finally passed to the device driver which creates the device-specific descriptors used to construct the packet, then initiates DMA of the packet from application memory to the device.

4. EVALUATING A NEW USER-SPACE API

We used our LiNK prototype to implement a new user-space API for a single-copy network protocol stack. The API takes advantage of the fact that the LiNK has direct, high-speed access to the same memory as user-space applications, so we can easily use the frameset abstraction to support communication between the application and LiNK. The goal for supporting a user-space protocol stack with the LiNK frameset abstraction was that performance could be achieved that would equal or surpass a kernel protocol stack. As a simple test of that goal we decided to port an existing high-performance webserver, Vivek Pai's *Flash* [25], to use our new protocol stack.

File	Size	Frequency
`file500.html`	500 bytes	35%
`file5k.html`	5kB	50%
`file50k.html`	50kB	14%
`file500k.html`	500kB	0.9%
`file5m.html`	5MB	0.1%

Table 1: Distribution of file sizes for Webstone benchmark

Test System	Server CPUs	Server Throughput/Mb/s	Connections per second
LiNK	1	71.34	438.2
Linux 2.2.16	1	45.5	281.7
	2	43.5	273.6
	3	42.8	268.4
	4	42.1	262.2

Table 2: Flash performance for the Webstone benchmark

The Linux kernels we planned on comparing our performance against did not support single-copy protocols, so modifying Flash to take advantage of the single-copy nature of our user-space stack would have been a lot of engineering work and also been an unfair comparison. Instead, we opted for a simpler path of implementing a TCP sockets compatibility library that presented the standard sockets API to Flash but utilised the single-copy API to interact with LiNK, an additional copy inside the library being used to preserve socket semantics.

4.1 Application Performance Results

Flash was evaluated by running it on a small testbed configured as follows; although the hardware we used for our measurements is now old, we do not believe this invalidates the results since it is not the absolute performance that is of interest but the relative. The server itself was run on a Hewlett-Packard Netserver LS quad-CPU (166MHz) machine, running under either the Linux 2.0.30-based LiNK prototype, Linux-2.2.16, or FreeBSD 4.2, with default parameters and the number of server processes varying from one up to the maximum number of available application CPUs—3 for LiNK, since 1 is dedicated to the LiNK, 4 for Linux and FreeBSD. Two client machines, were each connected to the server over separate Fast Ethernet (100Mb/s) switches—independent connections allow the server to generate aggregate bandwidth greater than one single link.

Two different programs were used on the client machines to generate requests for the server: the *Webstone 2.5* [20] web benchmark program, and the *curl* [29] utility for automated retrieval of web pages from a server

Webstone allows the user to specify a page-size distribution which the clients use to determine which pages to fetch from the server. The distribution shown in Table 1 is supposedly representative of 'real' web traffic. A coordinating process starts a number of clients on each client machine (5 per machine in this case) and provides them with the necessary parameters to drive the server. After a fixed time interval, one minute for this test, the coordinator queries

Test System	Server CPUs	Server Throughput/Mb/s
LiNK	1	75.0
	2	93.7
	3	98.8
Linux 2.2.16	1	50.83
	2	51.14
	3	50.86
	4	51.38
FreeBSD 4.2	1	51.30
	2	51.24
	3	32.0
	4	24.0

Table 3: Flash performance for the *curl* test

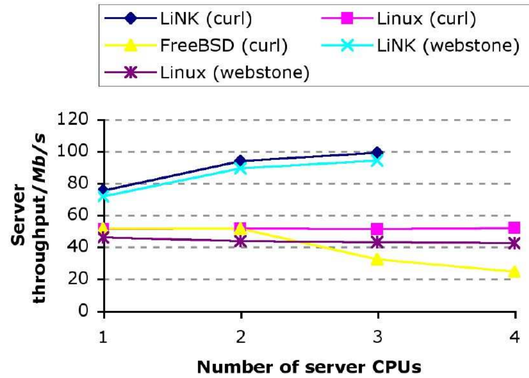

Figure 4: Flash performance for n server processes

the clients for various traffic statistics and produces overall server performance numbers. These results are shown in Table 2.

Unfortunately our prototype LiNK implementation was not stable enough to support Flash running with more than a single server process for long enough to complete the Webstone test. In order to provide a comparison with Linux and FreeBSD, *curl* was used to generate upper-bounds for server performance by repeatedly downloading the largest file in the distribution (5MB) and measuring the average throughput. These measurements are shown in Table 3; as an aside, these numbers are essentially the same as those we obtained in microbenchmarks measuring raw TCP throughput.

These throughput figures were used to calculate projected values for the Webstone benchmark on LiNK by multiplying the *curl* throughput for 2 or 3 server CPUs by the ratio between the Webstone and *curl* throughput for a single CPU—95%. Both sets of throughput measurements are displayed in Figure 4.

Similarly, when FreeBSD was used as the test platform for the WebStone test all web connections would cease transmitting after a few seconds, regardless of the number of server processes. Fortunately it proved possible to at least complete the *curl* measurements on this platform. However, the results show unexpectedly poor performance for more than two server processes; one might surmise that perhaps FreeBSD is optimised for the common case of dual-processor SMP systems in a way which penalises systems with a higher degree of multiprocessing.

What we see from this graph is that the LiNK prototype offers better bandwidth scalability, in terms of the number of CPUs utilised by the server, than either Linux or FreeBSD; we hypothesise that the failure of Linux and FreeBSD to take advantage of increased number of server CPUs is due to concurrency problems caused by locking in the kernel protocol stacks. The unexpectedly small increase for LiNK's throughput when the server is given 3 processors rather than 2 was determined to be caused by a combination of overutilisation of the LiNK CPU due to the overhead of polling and scheduling bottlenecks in the base Linux kernel.

However, this result should obviously be taken with a large pinch of salt, since both kernels have undoubtedly made significant progress from the tested versions in their support for SMP systems; we are in the process of upgrading our test environment to repeat these measurements on contemporary hardware and OS kernels. Nonetheless, the results do support the basic principle that a user-space protocol stack can be made to perform as efficiently as an in-kernel stack, without significant modifications to the server applications, as long as the appropriate abstractions are made available by the host OS and/or network devices.

5. RELATED WORK

A large amount of work has been conducted in the design and implementation of advanced NICs. The topic was particularly popular in the early 1990s [4, 5, 6, 8, 12, 15, 26, 32], but attempts to commercialise the technology, such as the *Virtual Interface Architecture* [3], based upon the Cornell *U-Net* project [32] were mostly unsuccessful. After a period of relative quiet, there appears to be a resurgence in interest in the area, perhaps spurred by the beginnings of commercial adoption of advanced NIC designs.

A number of groups have recently presented their experiences attempting to construct advanced NICs for modern systems: Haas et al. [14] discussed their work with the IBM *PowerNP* system, while Hady et al., [13] presented their *'Twin cities'* prototype, which uses a special hardware platform to tightly couple an IXP1240 with a general-purpose CPU. Freimuth et al. [11] described how they built a software prototype in order to model and measure the behaviour of a new network interface design.

The *NEPAL* framework, designed and implemented by Memik and Mangione-Smith [19], is complementary to our work on LiNK as it aims to provide network processor application developers with an automatic parallelisation tool to take advantage of the multiple processing elements available in many network processors. One possible scenario for use of NEPAL alongside LiNK is LiNK running on one processing element while a parallelised application takes advantage of the remaining elements. A similar structure may be appropriate for systems such as Intel's IXP network processor, where a full-featured CPU acts as a controller for a number of simple, special-purpose packet processors—in this case LiNK would be used on the control processor.

One of the original motivating factors for LiNK was as a vehicle for investigating user-space protocol implementations, of which a number have been described in earlier work [7, 9, 31]. Another motivation was to attempt to demon-

strate how an SMP system might be structured to provide maximum network performance; earlier work [1, 23] on parallelising protocol stacks had independently shown that contention for shared resources is the primary limiting factor for simple protocols such as UDP, but synchronisation dominates for more complex protocols. This suggests that an approach which minimises the amount of state shared between network flows, such as the user-space protocol implementations in LiNK, is an important step in maximising the benefit of multiple processors. This conclusion is echoed by Nahum et al. [23], who state that intra-flow parallelism is severely limited by the locking which is required, while inter-flow parallelism is reasonably scalable.

6. CONCLUSIONS AND FUTURE WORK

Network processors are experiencing a resurgence in popularity, both commercially, where they are being deployed in various scenarios, and also in the research community, where the availability of significant computing power on the network device offers a great deal of potential for developing new ideas. However, the software environment provided by these advanced NICs is still relatively primitive.

We believe that a *lightweight NIC kernel*, designed specifically for the advanced NIC environment can offer many of the same benefits to NIC software developers that general-purpose operating systems provide, while still remaining simple and streamlined. Our prototype implementation of this architecture runs on commodity hardware and has been used to demonstrate the practicality and performance of a novel user-space API.

The results obtained from our LiNK prototype are extremely promising, but there are a number of areas still to be explored. For practical reasons the prototype needs to be moved forward to current hardware and OS kernels, and we also imagine a modified version of our current implementation that does not require multiple processors but still presents the same API to the host OS and user-space applications, thus permitting software to be developed on a wider range of systems.

6.1 NIC Prototype and Debugging Environment

One of the uses envisioned for LiNK was an environment for developing and testing new NIC features. Here the LiNK prototype is valuable, providing a convenient platform for researchers and engineers to develop new software on, particularly when hardware is in the design and testing phases. But the value of such a prototype is greatly increased if the same APIs are available in the production system, so we intend to explore the ease with which LiNK can be implemented on real devices.

6.2 Applicability to Chip Multiprocessors

The final, and perhaps most promising area of future work, is to investigate the feasibility of the *virtual network processor* model for use in production systems. McAuley and Neugebauer [17] support such a case, arguing that the increasing number of special-purpose processors in a system, combined with the emergence of multi-threaded and multi-core CPUs, require new OS architectures in order to efficiently harness the power of all these processors. We believe that a simple software architecture such as LiNK has an important part to play in such systems.

7. REFERENCES

[1] M. Björkman and P. Gunningberg. Locking effects in multiprocessor implementation of protocols. In *Proc. of ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 74–83, Sept. 1993.

[2] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proc. of the 6th Symp. on Operating Systems Design and Implementation*, pages 347–362, San Francisco, CA, Dec. 2004.

[3] Compaq Computer Corp., Intel Corporation, Microsoft Corporation. *Virtual Interface Architecture Specification Version 1.0*, 1997. `www.viarch.org`.

[4] B. S. Davie. The architecture and implementation of a high-speed host interface. *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)*, 11(2):228–239, February 1993.

[5] B. S. Davie, J. M. Smith, and C. B. S. Traw. Host interfaces for atm networks. In A. N. Tantawy, editor, *High Performance Communications*, pages 195–224. Kluwer Academic Publishers, 1993.

[6] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *1994 ACM SIGCOMM Conference*, pages 2–13, Aug.-Sep 1994.

[7] A. Edwards and S. Muir. Experiences implementing a high-performance TCP in user-space. In *Proc. of ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 196–205, Sept. 1995.

[8] A. Edwards, G. Watson, J. Lumley, C. Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost gb/s lan. In *1994 ACM SIGCOMM Conference*, pages 2–13, Aug.-Sep 1994.

[9] D. Ely, S. Savage, and D. Wetherall. Alpine: A User-Level Infrastructure for Network Protocol Development. In *USENIX Symposium on Internet Technology*, pages 171–184, San Francisco, CA, Mar. 2001.

[10] D. R. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proc. of ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 53–59, Aug. 1996.

[11] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracy. Server Network Scalability and TCP Offload. In *Proc. of the 2005 USENIX Annual Technical Conference*, pages 209–222, Anaheim, CA, Apr. 2005.

[12] D. J. Greaves, D. McAuley, and L. J. French. Protocol and interface for atm lans. In *5th IEEE Workshop on Metropolitan Area Networks*, May 1992.

[13] F. T. Hady et al. Platform level support for high throughput edge applications: The twin cities prototype. *IEEE Network*, 17(4):22–27, July 2003.

[14] R. Hass et al. Creating advanced functions on network processors: Experience and perspectives. *IEEE Network*, 17(4):46–54, July 2003.

[15] H. Kanakia and D. R. Cheriton. The vmp network adapter board (nab): High performance network communication for multiprocessors. In *ACM SIGCOMM '88*, pages 175–187, August 1988.

[16] I. Leslie et al. The design and implementation of an operating system to support distributed multimedia applications. *IEEE/ACM Journal on Selected Areas in Communications*, 14(7):1280–1297, Sept. 1996.

[17] D. McAuley and R. Neugebauer. A case for Virtual Channel Processors. In *Proc. of the 1st Workshop on Network-I/O Convergence: Experience. Lessons, Implications*, Karlsruhe, Germany, Aug. 2003.

[18] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. of the Winter USENIX Conference*, pages 259–269, Jan. 1993.

[19] G. Memik and W. H. Mangione-Smith. NEPAL: A Framework for Efficiently Structuring Applications for Network Processors. In *Proc. of 2nd Workshop of Network Processors*, Anaheim, CA, February 2003.

[20] I. Mindcraft. WebStone: The benchmark for web servers, 2000. `www.mindcraft.com/webstone/`.

[21] J. C. Mogul. TCP Offload is a dumb idea whose time has come. In *Proc. of USENIX HotOS Workshop*, Hawaii, May 2003.

[22] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, Aug. 1997.

[23] E. M. Nahum et al. Performance issues in parallelized network protocols. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, pages 125–137, Nov. 1994.

[24] D. P. Olshefski, J. Nieh, and E. Nahum. ksniffer: Determining the Remote Client Perceived Response Time from Live Packet Streams. In *Proc. of the 6th Symp. on Operating Systems Design and Implementation*, pages 333–346, San Francisco, CA, Dec. 2004.

[25] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proc. of the Annual USENIX Technical Conference*, June 1999.

[26] K. K. Ramakrishnan. Performance considerations in designing network interfaces. *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)*, 11(2):203–219, February 1993.

[27] J. M. Smith and C. B. S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.

[28] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pages 216–229, Banff, Alberta, Canada, Oct. 2001.

[29] D. Stenberg et al. cURL: a client that groks the URLs, Mar. 2001. `curl.haxx.se`.

[30] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 22(4), Nov. 2004.

[31] C. A. Thekkath et al. Implementing network protocols at user level. In *Proc. of ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 64–73, Sept. 1993.

[32] T. von Eicken et al. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 40–53, Dec. 1995.

[33] L. Zhang. Virtual Clock: A new traffic control algorithm for packet switching networks. In *Proc. of ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 19–29, Sept. 1990.