DMA Mechanisms for High Performance Network Interfaces

Zubin D. Dittia zubin@dworkin.wustl.edu
Jerome R. Cox, Jr. jrc@hobbs.wustl.edu
Guru M. Parulkar guru@flora.wustl.edu

In modern high performance network interfaces, the design of the DMA (Direct Memory Access) subsystem is of paramount importance. Indeed, there is no single other aspect of the design that more influences the final throughput and latency that is obtainable by end applications.

Often, NIC (network interface card) designers tend to be hardware experts, with little or no experience in operating systems (OS) software architectures. Since a well designed DMA subsystem MUST closely cooperate with the software running on the host computer, it is little wonder that many so-called "high performance" NICs fall severely short of their targets in terms of performance numbers for real world applications.

Another common mistake made by high performance NIC designers is to focus all their efforts on bandwidth, and to ignore the impact of latency. While it is true that the latency introduced by the NIC hardware and software is unimportant for wide-area network (WAN) applications, the same cannot be said for local-area network (LAN) applications. Since a very large proportion of applications do fall into the latter category, the NIC's DMA subsystem should be designed to cooperate with software in delivering the minimum achievable latency to end applications.

In this article, we will explore various tricks which, when applied to the DMA subsystem of a modern NIC, can result in dramatic performance improvements in terms of both throughput and latency. Some of these tricks are in common use today, while others are relatively new or are still in the research stage. Many of these ideas are presented using an example state-of-the-art NIC chip called the APIC [1, 2]. This chip, which supports link speeds of upto 1.2 Gb/s, was designed and implemented by the authors under a contract from DARPA (Defense Advanced Research Projects Agency). Although the APIC was designed to interface to an ATM network, many of the ideas used in the chip are equally applicable to NICs for other types of networks.

What is a DMA Subsystem?

In simple terms, the DMA subsystem in a NIC is responsible for moving packets to and from the host's main memory.

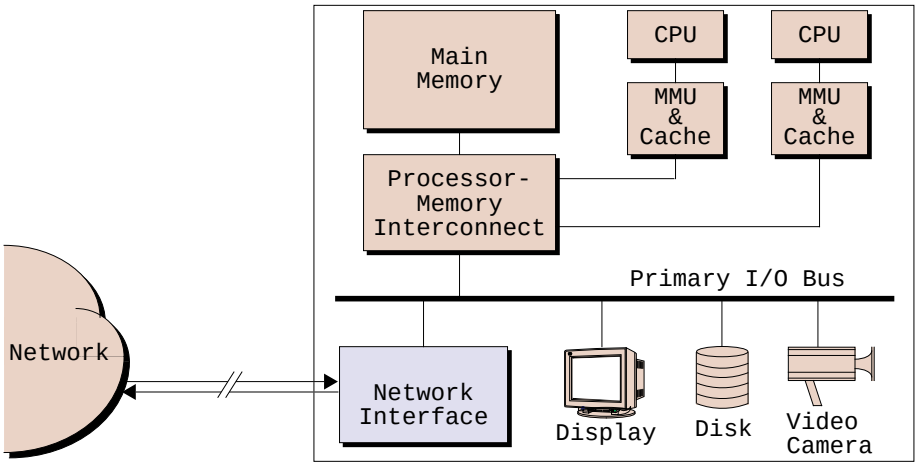

Figure 1: Typical Host Architecture

Figure 1 shows the architecture of a typical modern PC or workstation. The NIC is usually one among a slew of devices which plug into a slot on the machine's primary I/O bus. A good example of a primary I/O bus is the PCI (Peripheral Component Interconnect) bus, which originated as a standard for PCs, but has subsequently gained widespread acceptance in the workstation market too. A special bridge chipset implements the interconnection between processor modules, the system's main memory, and the I/O bus. Because the I/O bus specification remains relatively stable over several generations of computers, this kind of an architecture provides a level of "insulation" that is necessary to protect third party (I/O) device vendors from having to deal with rapid changes in processor-memory subsystem designs.

As shown in the figure, there are two mechanisms by which devices on the I/O bus can communicate with software running on the host processor(s). In the first technique, the software can use processor instructions to read or write data directly from or to the device. This mechanism, called "Programmed I/O", works by requiring the device to make some of its internal memory and registers available to the host processor; usually, this is achieved by mapping the device into an unused portion of the same address space that is used by the processor to access main memory. In other words, by issuing load and store instructions with these special addresses, the processor can read or write to the registers (or memory) resident on the device. This kind of access method is also commonly referred to as "Memory-mapped I/O".

The second method used to communicate information between an I/O device and software running on the host processor is called DMA. Here, all communication passes through special "shared" data structures that are allocated in the system's main memory. What makes these structures shared is the fact that they can be read from or written to by both the processor and the device.

In practice, both methods are used in most devices. Programmed I/O provides a synchronous access

interface to the device, while DMA provides an asynchronous interface. Usually, DMA is the preferred technique when large amounts of data are to be transferred to or from memory, because it does not tie up the processor for the duration of the transfer. Programmed I/O is useful when small amounts of data are to be transferred, or when the interaction needs to be synchronous. Usually, control interactions with the device are implemented using programmed I/O.

Most modern NICs use DMA as the preferred technique for moving packets to or from main memory. The DMA subsystem is the part of the NIC that is responsible for all DMA related actions.

One of the required features of the DMA subsystem in a NIC is "scatter-gather DMA". This refers to the ability to be able to handle packets which are fragmented in memory. Such fragmentation occurs because packets are constructed by network protocols so that they usually reside in different regions of the memory. Additionally, a single packet may be broken up into smaller pieces that reside in different memory locations. This happens, for example, if the protocol which constructed the packet used separate buffers for the packet header and packet data.

Why is the DMA Subsystem so important?

The design of the DMA subsystem in a NIC is made complicated by the fact that there are several trade-offs to consider. One of these has to do with choosing between better memory utilization and improved performance. Another, which is more of a software issue but has a major impact on the DMA subsystem, has to do with the selection of an appropriate API (application programming interface) for applications: usually, APIs that are more convenient and easier to use result in worse performance. A poorly designed DMA subsystem which does not take into account these trade-offs and software interactions can result in phenomenally bad overall performance.

One of the most important issues to consider in the design of a DMA subsystem is the number of "data touch" operations. Any time that packet data is read from or written to main memory, it is considered to have been "touched". A design should try to minimize data touches, because of the large negative impact that they can have on performance. The reason for this is that main memory bandwidth has not kept pace with increases in processor performance, so that any reduction in the number of times memory is accessed for a given piece of data can result in large performance gains.

Consider, for example, a typical PC with a main memory bandwidth of about 1.5 Gb/s for sequential reads, and 0.7 Gb/s for sequential writes. If we assume that on the average there are three reads for every two writes, the resulting memory bandwidth is ~1.2 Gb/s. If our DMA subsystem requires five data touch operations for every word in every packet, then clearly the theoritical maximum throughput that we can

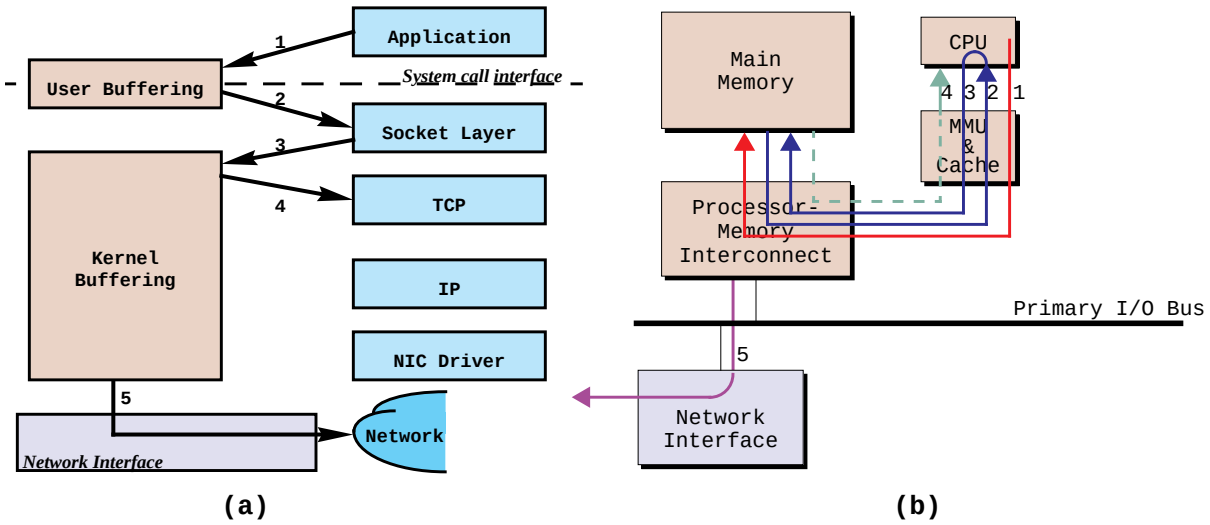

Figure 2: Data Touch Overhead in a Typical Protocol Stack

expect from such a system would be only a fifth of the memory bandwidth, or 0.24 Gb/s. Clearly, every data touch that we can save would provide for significant improvements in throughput.

The number of data touches that we used in the last example (five) is not an unrealistic number; in fact, many modern TCP/IP protocol stacks incur at least that many data touches. To see why this is so, consider Figure 2(a), which shows the data touch operations involved in transmitting a packet using a TCP/IP stack implemented in a typical monolithic Unix kernel. Following the numbered events in the figure, we have:

1. The application generates the data to be sent and writes it to its user-space buffer, following which it makes a system call to the socket layer to transmit the data;

2, 3. The socket layer copies the data from the user buffer into a set of kernel buffers that are used to hold packets.

4. TCP reads the data so that it can compute the checksum which has to be inserted in the packet header.

5. The network interface reads the data from the kernel buffer and transmits it.

Figure 2(b) shows what happens in hardware for these five data touch operations. Some of the lines are dashed to indicate that the corresponding read operation may be satisfied from the cache rather than the main memory. In the best case, there are three data touches (to memory) for any given piece of data; in the worst case, there are five.

Figure 2 only showed the data touches for an outgoing packet; a similar but reversed sequence

applies on the incoming side.

Data touches are not bad just for throughput performance; they also adversely affect packet latency, because of the extra time the processor spends copying data. Clearly, it is important to be able to minimize data touch operations.

NIC design choices that affect the DMA Subsystem

In this section, we look at some architectural choices available to NIC designers, and how they affect the DMA subsystem.

Cut-through vs. Store-and-forward

Network adapters fall into two main categories: cut-through, and store- and-forward. In a cut-through adapter, it is possible that the transmission of a frame can begin even before the entire frame has been read out of main memory. On the receiving end, a cut-through adapter can store part of a frame in main memory before the entire frame has been received. In a store-and-forward adapter, an entire frame needs to be read from main memory before the adapter will begin transmitting it. Also, a store-and- forward adapter will not write a frame into main memory until it has finished receiving the entire frame. Cut-through adapters have the advantage of lower delay, and of not requiring local memory on the network interface card (NIC) to store frames. They have the disadvantage that they might end up transmitting partial frames if there is an error, and of reporting receipt of partial or corrupted frames (which leaves the job of cleaning up to the software).

One of the possible techniques that is often employed to reduce the number of data touches is to move the transport layer checksum computation function into hardware on the NIC. This is easy to do on a store-and-forward NIC (the checksum can be computed while moving data between the NIC and main memory). But in cut-through adapters, it is not possible to compute and insert a checksum in the header of an outgoing packet, because the header may already have been transmitted by the time we finish computing the checksum. This means that either we would be forced to use a transport protocol with trailer checksumming, or incur the overhead of computing the checksum in software. Since TCP, the most popular transport protocol, uses a header checksum, we may have to live with the latter option if a cut-through adapter is used. However, there is a trick which is often quoted (but seldom implemented) that can be used to allow the checksum computation for an outgoing packet to proceed without the overhead of a data touch. Referring to Figure 2, if the processor were to compute the checksum while copying data from the user buffer to the kernel buffer (steps 2 and 3), then no extra data touches would be involved. This scheme cannot easily be used on the receiving end, because the outcome of the checksum verification for an incoming packet needs to be known well before the copy from kernel to user space takes place. So for incoming

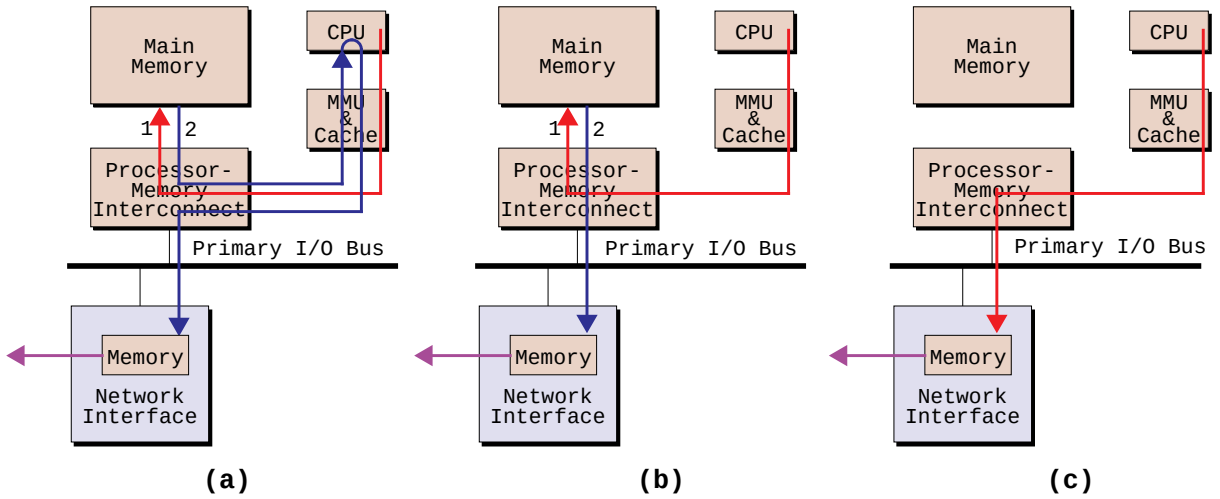

Figure 3: Impact of On-board Memory on Data Touches

packets, the only way to avoid a data touch for checksum computation is to implement it on the NIC in hardware. Note that in this case, header checksums are not a problem, because although a cut-through adapter cannot usually verify the correctness of such a frame by itself, it can provide the computed checksum over the frame to the software and leave the job of verification of the checksum (i.e., comparing it to the value in the packet header) to the software.

On-board memory or not?

Another design choice that affects the DMA subsystem is whether or not the NIC has an on-board memory that can be used as a staging buffer for packets. Figure 3 shows how the number of data touches can be reduced if a NIC contains on-board memory. In Figure 3(a), the kernel buffers (see Figure 2) are allocated from memory on the NIC, and programmed I/O is used to move data from the user buffer to these on-board kernel buffers. The checksum can be computed during this copy loop. Figure 3(b) shows the same layout, but with DMA being used in place of programmed I/O. In this case, the checksum computation cannot be done by the processor, so it would have to be implemented in hardware on the NIC. Each of these schemes involves only two data touches. Figure 3(c) shows a third alternative which involves no main memory accesses at all (zero data touches): the user buffers are allocated from the NIC's on-board memory. This approach is not usually used because a very large amount of memory may be needed on the NIC, which drives up its cost, and because the API that would be presented to applications in this kind of framework would be incompatible with the existing socket based API used with most current applications.

On-board processor or not?

A number of NICs have on-board processors and firmware that is used to perform various NIC related tasks. This has both advantages and drawbacks; often, on-board processors drive up the cost of the

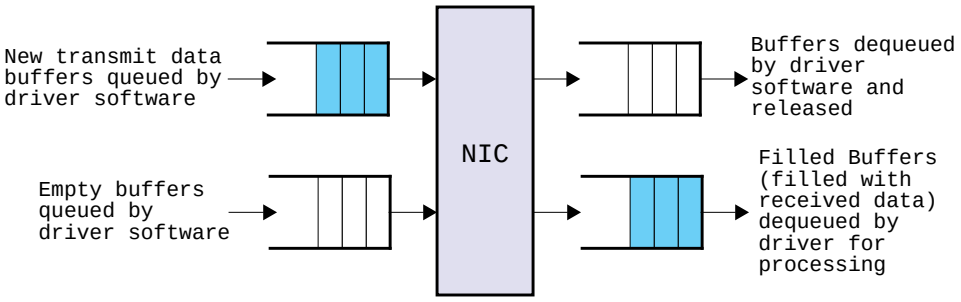

Figure 4: FIFO Queues Used in Simple DMA

NIC, but they provide more flexibility in the sense that more features can be added as necessary without a lot of work. Usually with such NICs, the method by which the driver for the NIC interfaces to the rest of the OS remains the same; however, some researchers have argued for moving portions of the protocol stack onto the NIC. Increasingly, this approach has gained disfavor because it requires very close interaction between the OS on the host and the software running on the NIC's processor. We will not discuss this approach any further.

Standard Techniques Used in DMA Subsystems

Simple DMA

The simplest scatter/gather DMA subsystem is based on a pair of input FIFO queues and a pair of output FIFO queues, as shown in Figure 4. One queue from each pair is used for transmit and the other for receive. The transmit input queue is used by the driver software to pass to the NIC buffers that contain packet data which is to be transmitted; once the NIC has finished transmitting data from a buffer, it deposits the buffer into the transmit output queue. The driver can later read from this queue and free the buffers. On the receive side, the driver provides empty buffers to the NIC on its receive input queue; these buffers are filled in with data from received packets by the NIC, and deposited into the receive output queue. At some later time, the driver can dequeue buffers from this queue and pass the packets contained therein to higher layer protocols.

Figure 5 shows how this simple form of DMA is implemented in the APIC network interface for the receive side only. Each buffer is associated with a descriptor, which is a small fixed size memory data structure that contains a pointer to a buffer, its length, a pointer (called the link) to another descriptor, and some flags. The link pointers are used to chain descriptors together to form a queue. A single chain of descriptors (and buffers) can serve the purpose of one input and one output queue. The APIC reads in a descriptor, reads or writes packet data to or from the corresponding buffer, and once it has finished process-

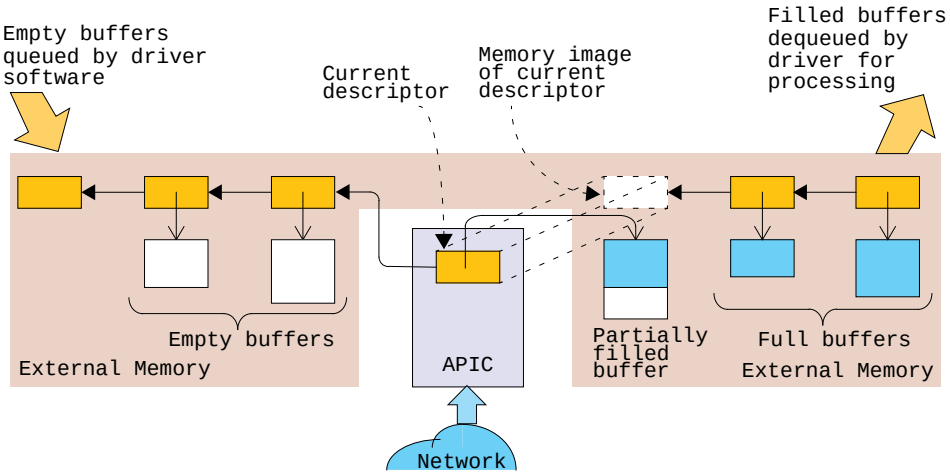

Figure 5: Implementation of Simple DMA in APIC (Receive channel only)

ing the buffer, writes the descriptor back and fetches the next one in the chain. In the figure, everything to the left of the APIC can be thought of as the receive input queue, and everything to the right as the receive output queue.

Note that linked lists of descriptors are only one way of implementing FIFO queues; another popular technique is to use circular arrays of descriptors. The linked list approach is more flexible because it allows descriptors to be added and removed from the list on demand, while an array of descriptors has to be statically allocated.

Pool DMA

Often, it may make sense for the NIC to support multiple transmit and/or receive DMA channels. For example, an ATM NIC may implement a separate channel for each ATM virtual circuit (VC). As another example, modern network interfaces may support different traffic classes that have different QoS requirements; in this case, outgoing packets can be queued on different channels depending on their traffic class, and the NIC can service these channels based on some predefined policy (e.g., for weighted fair queueing).

The simple DMA scheme we described above can be used quite effectively for multiple transmit channels by simply having one input-output queue pair per channel. However, in the case of multiple receive channels, using this simple scheme may result in memory wastage. To see this, note that the software would have to provide each channel with its own input queue containing empty buffers that are to be filled with received data. Since there is no way to predict in advance exactly when packets will arrive on a given channel, the driver would have to dedicate a large number of empty buffers for every channel. A better solution would be to have a single pool of empty buffers which can be shared by all the receive channels; whenever a channel needs a new empty buffer, it can draw from this pool. Thus, a single input queue

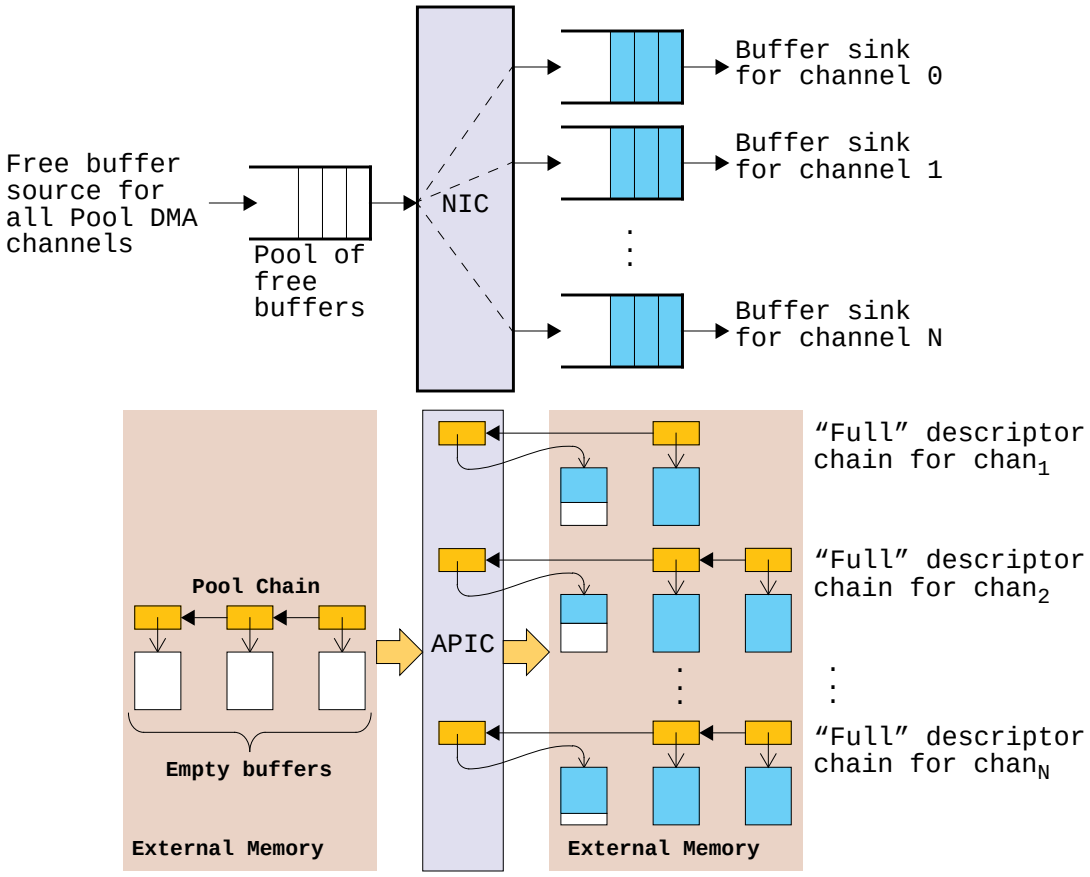

Figure 6: FIFO Queue Model for Pool DMA, and its Implementation in the APIC

suffices for all channels. The output queues for the channels can, of course, be different for different channels. This scheme is called Pool DMA. Figure 6 shows both the queue model for pool DMA, and its implementation in the APIC network interface.

Newer Techniques in DMA Subsystem Design

Page Remapping

A technique that has been proposed several times (but seldom implemented) allows the host computer's virtual memory (VM) system to be used to remap pages and thereby avoid copying data between user space and kernel. This can save upto two data touch operations. On the transmit side, no special support is needed from the NIC to implement this scheme; when the application makes a system call to send the data, all that needs to be done is to wire the application buffer in main memory (i.e., prevent it from being paged out to disk), and change the virtual memory mappings in the system's page table so that the pages comprising the buffer are mapped into the kernel. Subsequently, the application buffer can be used

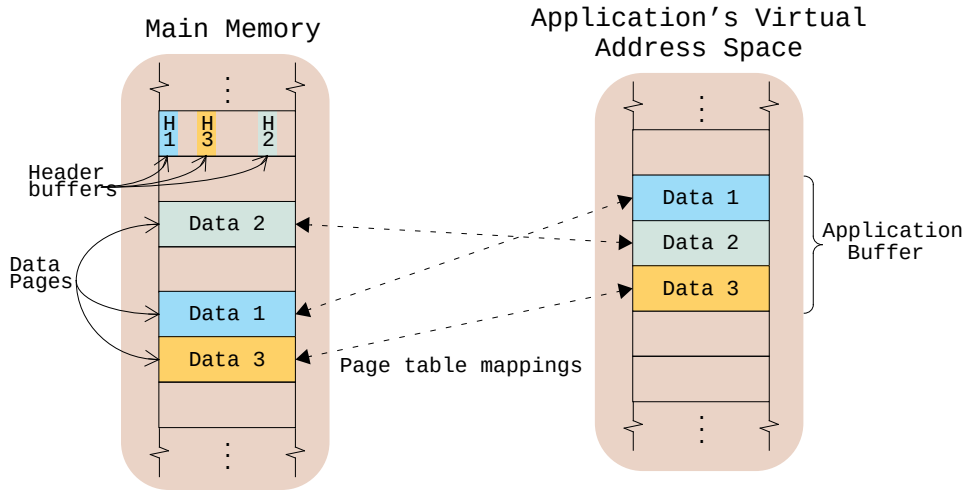

Figure 7: Using Page-Remapping to Avoid Data Copying

like a kernel buffer, thereby avoiding a copy step. On the receive side however, a special technique called "packet splitting" has to be implemented by the NIC's DMA subsystem, if we want to reap the benefits of page remapping.

Packet splitting is usually used in conjunction with pool DMA for receive channels. If a pool DMA channel has packet splitting enabled, the NIC will split all received packets on that channel into a header portion and a data portion. It will then write the header portions into buffers drawn from one shared pool of empty buffers, and the data portions into buffers drawn from a different (also shared) pool. The header buffers are assumed to be small buffers that are mapped into the kernel's address space, while data buffers are assumed to be complete VM pages. We assume that each packet carries enough data to fill a whole VM page. Figure 7 shows what happens after several packets have been received on a given channel. The main memory contains the headers and full pages of data. The protocols can examine the headers in order to process the packets, and based on sequence number information in the headers, they can decide where the data should reside in the application's address space. Once this decision has been made, it is a simple matter of remapping the data pages (which reside at arbitrary locations in main memory) to appropriate locations in the application's virtual address space; this is done simply by twiddling page table entries, thereby avoiding any data touches

Page remapping is generally not used for general purpose protocol implementations, because it places several overly restrictive constraints on the design of protocols and the NIC. For one thing, every packet has to contain exactly enough data as a page on the receiving host. For another, the NIC has to be able to figure out where the header ends and the data begins in a packet; this can be done by either fixing the header length (another protocol constraint), or by incorporating enough intelligence in the NIC to be

able to parse the headers of incoming packets. Another problem is that a socket-like API cannot be provided to applications, because applications do not have write access to a buffer while it is being sent or received into. A scheme like copy-on-write (COW) can be used to overcome this limitation, but only at the cost of more data touches.

Nonetheless, page remapping can be a very useful mechanism in a variety of specialized environments. For example, in cluster computers, it can be used to provide a very efficient implementation of distributed shared memory (DSM).

User-space access to the NIC and Protected DMA

Conventional NICs are usually controlled by a device driver that resides in a trusted context, usually the kernel. We now present a relatively new approach that allows driver code running as part of an untrusted user space process to have limited but direct control over the NIC, in a manner that is commensurate with the operating system's protection policies. This approach was first proposed and independently implemented by three different research groups [1,3,4], including our own. All three approaches try to provide the same basic functionality, but their implementation methods are quite different. The implementation technique used by the APIC has some important advantages, so we will focus on that technique in this section; readers are urged to look at the referenced papers for other methods of implementing direct user space access. It is to be noted that this kind of support in NICs is catching on in the industry too; Compaq, Intel, and Microsoft recently announced a specification [5] for a standard for such NICs, which they call "virtual interfaces".

Figure 8(a) illustrates the differences between the traditional kernel-resident control model, and the user-space control model. The figure distinguishes between two kinds of control operations: those on the data path, and those on the control path. Data path operations are executed every time some data is sent or received, while control path operations are relatively rare, and usually need to be executed only once or a few times in a connection's lifetime. This distinction is important, because it means that data path operations are executed very frequently and therefore should be optimized, whereas control path operations are relatively rare, so there is not much to be gained by optimizing them. Examples of data path operations include: queueing and dequeueing data buffers on DMA channels, informing the NIC that there are new buffers in these queues, etc. Examples of control path operations include: setting up DMA channels, NIC initialization, etc.

Returning to the figure, we see that in the traditional model, when a user- space process wants to use a network interface device, both control and data path operations need to pass through (and be blessed by) the OS kernel. In the user-space control model, only control path operations need to pass through (and be

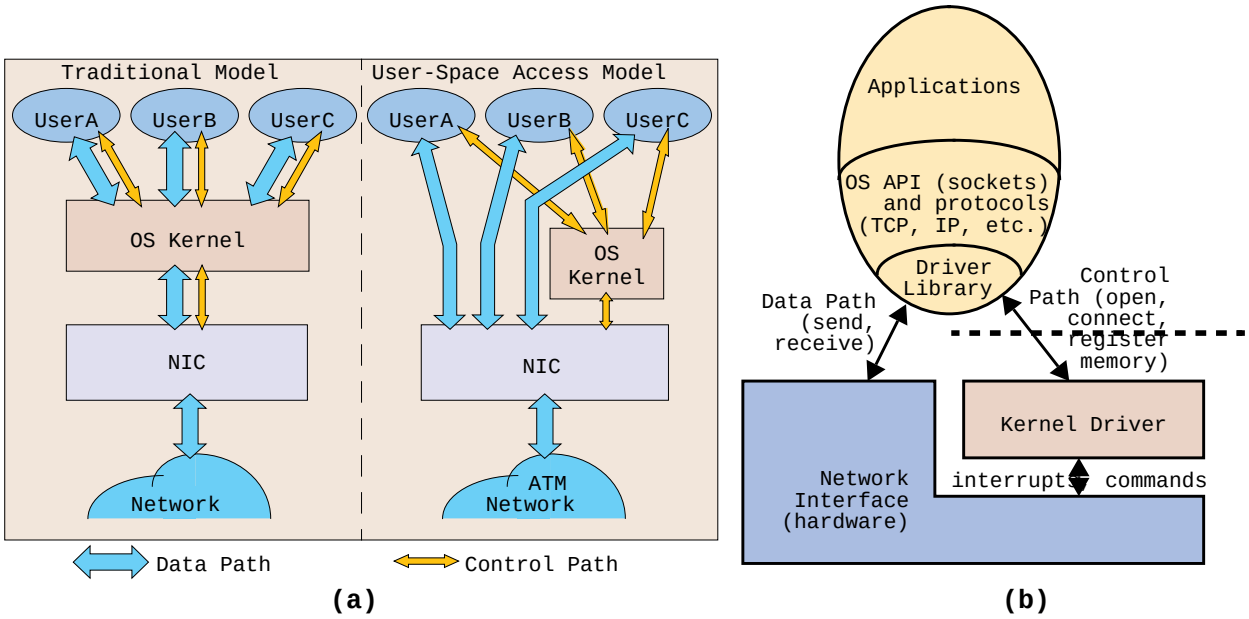

Figure 8: Traditional vs. User-Space Access Model

blessed by) the OS kernel; the frequently occuring data path operations do not need any kernel intervention (and therefore are more efficient).

Since we have removed the kernel from the data path, the NIC has to take over the job of "blessing" all data path operations. Here, "blessing" means ensuring that the protection policies imposed by the kernel are not violated. This requires the NIC to be able to distinguish between packets belonging to different application flows. In case of a network like ATM, this can be done based on the VC that is associated with a packet. For other types of networks (e.g., Ethernet), it may be necessary to first establish the equivalent of a flow, and to stamp each packet with the flow identifier.

The protection policies imposed by the kernel manifest themselves as sets of operations that are considered to be "legal" for the "owner" of a flow. A user process is said to be the owner of a flow if it holds an OS granted capability for that flow. Usually, the process that is responsible for opening a flow becomes the owner of the flow. Referring to Figure 8, it should be illegal for user process B to send data on the flow which is owned by user process A. Similarly, it should be illegal for user process B to be able to receive data arriving on a flow owned by user process C.

In the traditional model, such checks were performed in software by the kernel. With the user-space model, they become the NIC's responsibility. It is important to note that the user-space control model implies a different software structure, which is shown in Figure 8(b). Notice that a portion of the NIC device driver has been migrated to user-space. We will henceforth refer to this part of the driver, which runs in an untrusted context and is typically implemented as a library that can be linked with the applica-

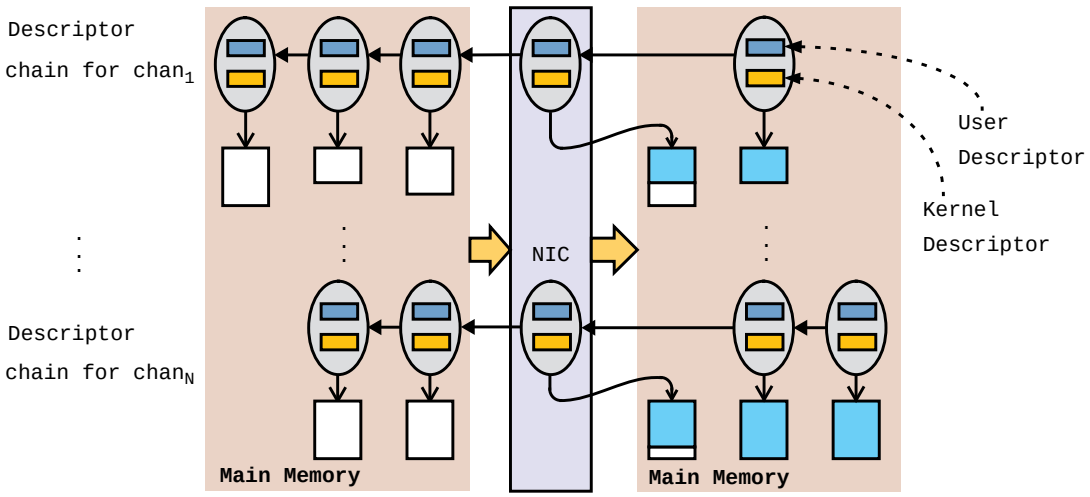

Figure 9: Protected DMA as an Implementation Technique for User-Access Model

tion, as the user-space driver. The trusted portion of the driver typically resides in the kernel, and is responsible for all control operations, for dictating protection policies that will apply to user-space drivers, and for fielding device interrupts - we will henceforth refer to this part of the driver as the kernel driver. Note that the protocol stack is implemented as a library that sits on top of the user-space driver.

The APIC implements the user-space model using a mechanism called Protected DMA. Protected DMA is in many ways similar to Simple DMA, but it enables certain protection checks to be carried out by the APIC on buffers and descriptors that are enqueued by untrusted code running in user- space. To enable this kind of checking, Protected DMA defines two types of descriptors: kernel descriptors and user-descriptors. Instead of having a single descriptor referencing each buffer, Protected DMA requires a pair of descriptors to refer to each buffer. One descriptor in the pair is a kernel descriptor, while another is a user descriptor. This is illustrated in Figure 9. Before the APIC can use a buffer from a Protected DMA channel, it always reads both descriptors for the buffer. The user descriptor contains values supplied by the user-space driver, and therefore cannot be trusted. The kernel descriptor on the other hand is initialized by the kernel driver and is not accessible to the user-space driver, so the values in a kernel descriptor are trustworthy. The APIC performs its protection checks by comparing values in the kernel descriptor against corresponding values in the user descriptor. The kernel descriptor is therefore used to certify the values supplied by the user- space driver in the user descriptor. Only if this validation succeeds can the APIC safely DMA data to or from the associated buffer. It is important to recognize that the FIFO queue model implied by Protected DMA is identical to that for Simple DMA (see Figure 4). Both associate a single dedicated descriptor chain for each channel. The difference is only in way the FIFO queues are implemented: in the Simple DMA case, there is a single descriptor per buffer and no protection checks made by the APIC,

while for Protected DMA there is a pair of descriptors for each buffer, and implied protection checks.

To use Protected DMA, the user-space driver is required to allocate special communication buffers (using malloc or a similar utility), to wire these in memory (using a system call like BSD mlock), and to make a system call to the kernel driver to associate descriptor pair(s) with the buffer and initialize the kernel descriptor(s). These are all assumed to be control path operations, and should not need to be done very frequently. In fact, in most cases it may be sufficient to perform these actions for a set of communication buffers just once when the program is started. Note that a single communication buffer may span multiple non-contiguous physical memory pages, so it may be necessary to associate multiple descriptor pairs with the buffer (one pair for each page in the buffer).

Once the afore-mentioned control path operations are completed, the user-space driver is free to create chains of descriptor pairs pointing to the communication buffers by modifying fields in user descriptors only. It can enqueue portions of these buffers in any desired order onto the protected DMA channel's descriptor chain. No system calls are needed for these data path operations. The protection checks performed by the APIC ensure that accesses to illegal descriptors or buffers will be rejected. For each descriptor pair, two such checks are needed. One check is needed to ensure that the buffer specified in the user descriptor does not exceed the bounds of the (previously registered) buffer pointed to by the kernel descriptor. The second check is used to ensure that the next descriptor pair which is referenced through the link pointer in the user descriptor lies within a valid range of descriptors (which have been dedicated to the flow). Both of these checks are simple inequality tests, and are easily performed in hardware.

The user-space access model for NICs that is enabled by technologies such as Protected DMA allow for extremely efficient networking. There need not be any data copying involved, which improves throughput, and the elimination of system calls and kernel intervention on the data path allows for very low end-to-end latency. For this reason, such schemes are especially attractive for cluster computing in a LAN.

Conclusions

We presented several different DMA mechanisms in this article. The DMA subsystems in many modern NICs include support for some or all of these mechanisms. But we would like to emphasize here that the DMA subsystem is not the only sybsystem in a NIC that needs to be optimized, albeit an important one. Another subsystem to which careful attention should be paid in modern NICs is the Interrupt subsystem; a problem known as interrupt livelock often plagues high bandwidth NICs, and there are several different techniques to address this problem. We plan to make this the topic of a future article.

References

[1] Z.D.Dittia, G.M.Parulkar and J.R.Cox, “The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques,” *Proc. IEEE INFOCOM 1997,* Kobe, Japan.

[2] Z.D.Dittia, J.R.Cox and G.M.Parulkar, “Design of the APIC: A High Performance ATM Host-Network Interface Chip,” *Proc. IEEE INFOCOM 97,* Boston.

[3] P.Druschel, L.Peterson and B.S.Davie, “Experiences with a High-Speed Network Adaptor: A Software Perspective,” *Proc. ACM SIGCOMM 94,* London, U.K.

[4] T. von Eicken, A.Basu, V.Buch and W.Vogels, “U-Net: A User-Level Network Interface for Parallel and Distributed Computing,” *Proc. 15th ACM Symposium on Operating System Principles,* Dec. 1995.

[5] “Virtual Interface Architecture Specification,” (Compaq, Intel, Microsoft): http://www.viarch.com.