ZeroCopy: Techniques, Benefits and Pitfalls

Eduard Bröse

washuu@cs.tu-berlin.de

Abstract: We discuss various approaches intended to improve the data processing performance in OS kernels/drivers commonly described as ZeroCopy techniques. The main principle of ZeroCopy is to avoid completely or at least minimize unnecessary data copy operations by the CPU while processing I/O data in kernel drivers such as networking stacks and disk storage drivers. Modern CPU and memory architectures provide many interesting features to implement such techniques efficiently, however their advantages are often overestimated. The complexity of the memory architecture as well as necessary data processing in networking protocols may impose many problems and sometimes nullify the advantages of a promising ZeroCopy technique. We will try to take a closer look at various existing and proposed implementation and analyze their weak points. This paper will mostly concentrate on the modern UNIX-like operating systems such as Solaris and Linux with an emphasis on ZeroCopy performance and implementation issues in networking stacks.

1 Introduction

While demand for computing performance grows, and current CPU designs have almost reached the edge of possible miniaturization, switch to parallelization with introduction of multicore CPUs is often considered the only feasible option. However, what is often forgotten, is that doubling of the CPU performance (which is also in no way guaranteed for parallel systems and requires redesign of software) doesn't mean an adequate increment in overall system performance, especially when it comes to processing large amounts of data. The performance of system memory, as well as the bus interface to it were not developing as fast as the CPU performance, with sustainable memory bandwidth attaining an average 35% increment per year, while CPU processing power increasing about 50% in the same time. Memory and bus bandwidth are thus the major limiting factors for the data-intensive applications, most notable of them is the networking.

The fundamentals of many modern operating systems including memory management and networking subsystems were created quite a long time ago, when network speeds were quite low and the CPU-to-memory performance ratio was still a relatively minor issue. The steady increase of network speeds, with gigabit Ethernet already being widely adopted in consumer sector calls for revision of the traditional data processing design in operating systems with the goal of eliminating unnecessary memory access overhead [1]. Even a brief analysis of the data processing paths in kernel reveals many such issues, where data is unnecessarily moved multiple times from one buffer to another.

Measurements of processing cost in networking stacks have shown that memory operations are responsible for a significant amount of processing costs, as compared to the per-packet processing which involves interrupt handling, context switches and buffer management. Processing of a single packet may cause four memory bus accesses and per-byte cost starts to dominate processing cost with packets longer than 128 bytes, linearly increasing with packet size. Estimations for different networking protocols in regard to their maximal packet length (Maximum Transmission Unit, MTU) show that about 18-25% of the performance were attributed to per-byte operations for Ethernet (MTU=1500), 35-50% for FDDI (MTU=4352) and 55-65% for IP over ATM (MTU=9180). Since data from network packets not only needs to be copied, but also checksummed, these operations are usually combined, with checksumming being accountable for about 20-30% of total per-byte cost. Benchmarking tests showed for example that a typical Athlon 1.2GHz system would barely handle just I/O related processing when exposed to 1Gbit/s Ethernet traffic on an unoptimized BSD TCP/IP stack [2].

ZeroCopy is a common name for various techniques and design improvements aimed at reduction of unnecessary memory accesses, usually involving avoidance of data copying. In the following chapters we present various ZeroCopy concepts and analyze their benefits but also their implementation and performance problems, often not realized through the general concept description.

1.1 Example of a Typical Data Transfer with read/write

First we will take a look at the typical file transfer operation as performed by some kind of server. Normally, the user application would allocate a buffer of suitable size to hold the data to transfer, read the data from a file and transfer it over the network connection to the client. From the application's point of view, the transfer is accomplished using two system calls, *read* and *write* (or *sendto*) and doesn't require any copying. However, if we look at the inner workings of the kernel parts involved in such transmission, we will see that such approach is quite inefficient, even with hardware support for DMA transfers.

First, the kernel would load the data from the disk into its own buffer using DMA, unless this data is still cached in a kernel buffer after a previous access to the same file. Such transfer doesn't require much CPU effort except for the buffer management and DMA setup and handling. After the file contents were fetched, data needs to be transferred into location in the user address space specified in the *read* call, which is done by CPU copy in a kernel function such as *copyout* or *copy_to_user*.

In the next step, the data again need to be transferred using CPU by user-to-kernel copy function such as *copyin* or *copy_from_user*, from the user buffer to the kernel buffer associated with the network stack. After the copy is completed, the network stack can begin packetizing it and sending to the network interface card (NIC), using DMA. The application can however already return from the system call and continue its operation while data is still being sent. The *write* call semantics imply that the contents of the user buffer may be safely discarded or modified immediately afterwards, since the kernel has its own copy which will be only discarded after successful transmission to the hardware, and if the used protocol requires it, after confirming successful arrival on the destination peer [3].

As we see, at least 4 copies are necessary for this simple operation, even when using DMA to communicate with the hardware, CPU is still accessing the data two times. If we go deeper in the inner workings of kernel parts involved, even more data processing by CPU may be revealed. In the read step, the data typically doesn't come directly from the disk, but must pass the filesystem layer, eventually resulting in data aligning and concatenating procedures. In the write step of our file transmit operation, the data, the future packet payload must be split into fragments suitable for packet size, headers must be prepended and checksum over payload must be calculated.

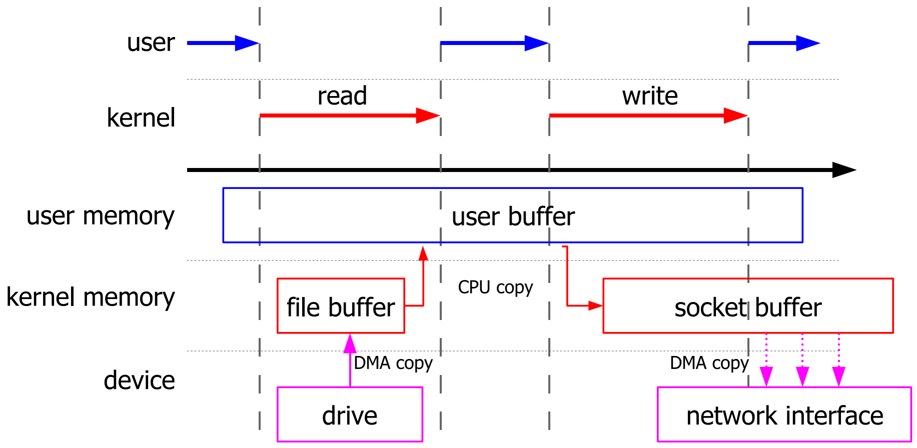

Figure 1: Typical file transfer with *read* and *write*

Modern filesystems and networking stacks already try to optimize these operations. Normally, the files stored in the file system are aligned to some block size (typically 1 kilobyte) and the filesystem tries to avoid file fragmentation. In most cases, file contents may be read into file cache buffers directly without additional alignment corrections.

In case of the networking stacks, appending headers and trailers to the payload by layers of the stack would require copying of the payload multiple times. Instead, these operations are replaced by handling a structure representing a complex packet, *skbuf* in BSD and Linux, or a more generic *mblk* in the STREAMS framework [4]. Headers can be appended in a linked list fashion by allocating an *mblk* structure that uses the aforementioned kernel buffer for storage, with pointers "cutting out" the needed portion of the payload, and the headers being smaller buffers allocated from a pool appended or prepended to it. As the message passes through the network layers, another blocks can be appended without copying contents. This concept also allows the reference of the same storage buffer by multiple packet messages and employs a simple reference counting mechanism, allowing freeing of the storage buffer as soon as all messages referencing it were discarded after being successfully sent.

The actual CPU copy also doesn't go to waste, since the most performance loss occurs due to the memory access and not CRC calculations on the CPU [2, 5], modern stacks use the opportunity of touching data by the CPU to compute the payload checksum at the same time. This technique is sometimes described as "SingleCopy" [5].

As such fragmented message reaches the destination driver, another copy might be needed to flatten it, so the DMA engine of the networking interface can fetch it from the host's memory in one turn. Some of the modern hardware supports so-called DMA Gather Copy that allows reading of multiple data fragments in one DMA session. This feature is further discussed in the "sendfile with DMA Gather Copy" chapter.

Even with such optimizations, our example will still use 2 CPU copies, which lead to a total 4 memory bus transfers. In general, almost every traditional system call involving data transmission would lead to at least one CPU copy between user and kernel space. Further, a simple file transfer like in our example doesn't require the data from the file to be transfered to the user application at all. In the following sections we will examine various techniques that can avoid or at least optimize such operations.

1.2 ZeroCopy Technique Classes

Traditionally, kernel serves as an abstraction layer between user application and hardware, brokering the data transfers between them. This basically leads to the need of two data transfers - from application to kernel and from kernel to hardware, compared to one that would be needed if the application would access hardware directly. Further, communication between hardware and software allows usage of DMA that relieves the CPU, but between two pieces of software, the application and the kernel, there's no such supporting feature. Various ZeroCopy techniques approach this problem from different angles.

ZeroCopy techniques can be roughly separated into three classes :

1. Avoidance and optimization of in-kernel data copying - as mentioned in our file transfer example, in some cases the application doesn't actually require an access to the data. This class of techniques aims at implementing new system calls or optimizing the traditional ones in order to achive more performance in such specialized cases, where data may be processed completely within kernel.

2. Bypass of the kernel on the main data processing path - analogous to the above, sometimes kernel doesn't need to deal with data directly and might be avoided, allowing direct data transfers between user space memory and hardware, with kernel only managing and aiding such transfers.

3. Optimization of data transfer between user application and kernel - finally the third class of techniques concentrates on optimization of the CPU copies between kernel and user space, which maintains the traditional way of arranging communication and is more flexible.

2 In-Kernel Data Copy

2.1 Using mmap

One simple way to force direct copy within kernel is to memory-map the file being transmitted instead of reading it into the user buffer, then specify the mapped buffer in the write call. This way, the user application not only saves memory, it avoids data transfer to and from usermode, as the write call may now directly copy data from one kernel buffer to another. While being completely POSIX-compatible, this method does not achive ideal performance. One CPU copy is still required in order to maintain the write semantics, and the establishment of the mapping is also a costly VM operation that requires page table modifications and TLB flushes in order to maintain memory coherence, however since mapping is usually done for a relatively large area (many kilobytes), these costs would be easily outweighted by CPU copy over the same length. Further, it is subject to exceptional situations that need proper handling, for example, truncation of the file while it is being transmitted will cause signalling of the application that must be caught and handled properly [3].

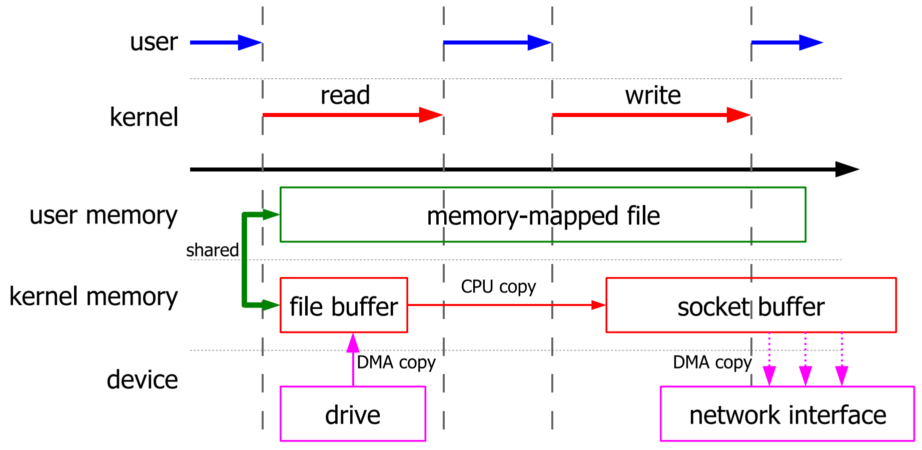

Figure 2: Using *mmap* instead of *read*

2.2 sendfile

In order to simplify the user interface, while exploiting the CPU copy reduction advantage of mmap/write technique, the *sendfile* system call was introduced on some operating systems, including BSD, Solaris, AIX and Linux, first implemented in the kernel 2.1. Windows NT also has a similar API function TransmitFile.

sendfile interface requires two file descriptors, with the first one being a readable memorymappable file such as regular file or a block device, the second one can be a writable file or a network socket. The semantics of the *sendfile* is to transmit data of the specified length, or completely, from the first descriptor to another without copying or mapping any of it to the user address space, which makes it only usable in situations where the user application is only interested in data copy but not in any processing. Since the transmitted data should never cross the user/kernel boundary, *sendfile* greatly reduces costs of memory management, compared to the mmap solution, it also requires only one system call. *sendfile* returns once all data was successfully copied from disk buffers to socket buffers while data may still be transmitted via network, the semantics are therefore similar to that of the write call. *sendfile* implementation also fits well in the kernel framework, as it doesn't require any changes in the memory and buffer management, this way it is also able to transparently utilize the page cache (also known as disk cache) parallel to generic file read functions.

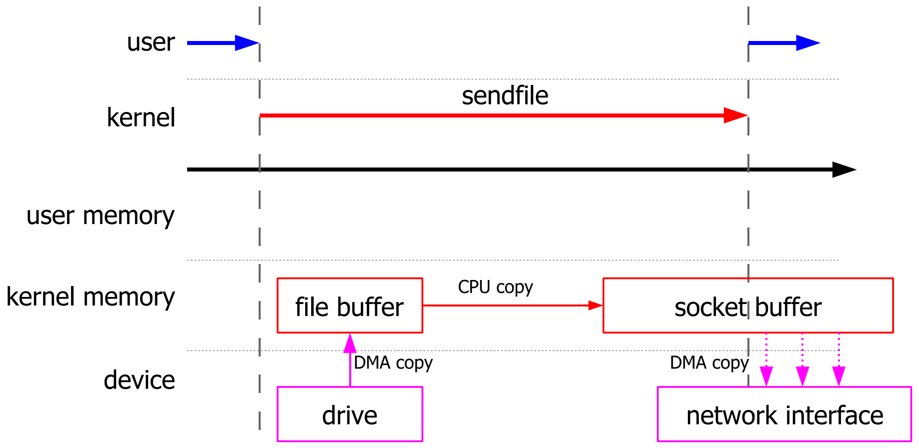

Figure 3: In-kernel file transmission with *sendfile*

While being a good replacement to the mmap technique, *sendfile* is very limited in its functionality and is typically only used in file-serving network applications like web servers. It was stated [6] that the decision to include *sendfile* in Linux kernel was only made on demand from Apache project that used it on other platforms, and because of simplicity of the implementation and good integration with the rest of the kernel. The main problem is that *sendfile* requires explicit usage by the applications, is not standardized and has different interfaces on other UNIXes. Due to asynchrony of the network transfers it is also more difficult to implement the receiving counterpart to *sendfile* call, leaving the receiver without such technique. Regarding performance, *sendfile* is still requiring one CPU copy from file to socket buffers which also pollutes cache with contents of the data transmitted.

2.3 sendfile with DMA Gather Copy

Further improvement of sendfile technique is possible if the CPU copy between the disk buffer and socket buffer is eliminated. It is possible with some hardware devices that support so called DMA Gather Copy. The main concept behind it, is that given the gather/scatter feature is available, the data for DMA transfers do not need to be in a consecutive memory area, but may be collected from multiple locations. This way, the data read from a file (the payload) does not need to be copied to the socket buffer at all. Instead, only a buffer descriptor is passed to the networking stack, which then constructs packet headers and trailers in its own buffers, and combines everything to a single network packet via DMA gather copy. The DMA engine of the networking card will read the headers and the payload from multiple locations in a single operation.

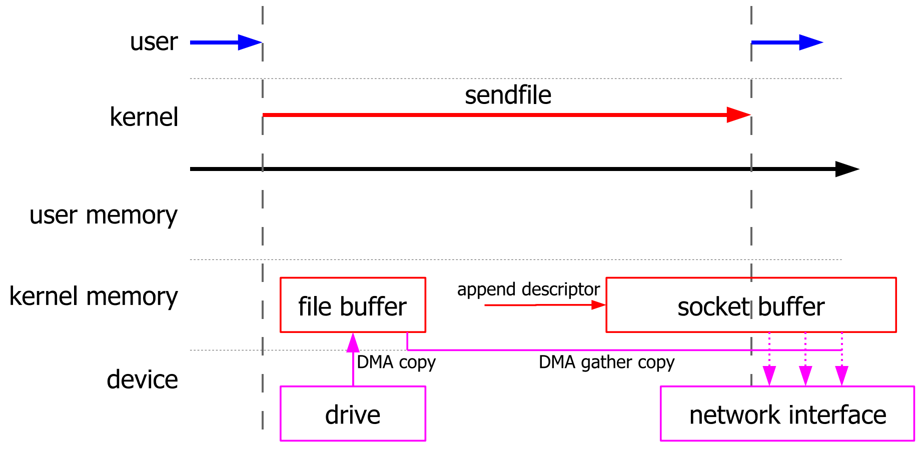

Figure 4: *sendfile* with DMA gather copy

This way, CPU would not only avoid the copy, it will theoretically never come in contact with transmitted data, which can have a positive effect on the CPU performance: first, cache is not polluted with payload data, second, the cache coherency doesn't need to be maintained - caches do not need to be flushed before or after the DMA transfer. In practice however, the latter is rather difficult to implement, considering how *sendfile* is integrated into the memory management framework. As mentioned before, the source buffer may be a part of the page cache, which means it is available for the generic read functionality and may be accessed in traditional way as well. As long as the memory area can be accessed by the CPU, the cache consistency has to be maintained by flushing the caches before DMA transfer from them.

2.4 splice

Another proposal for the fast in-kernel data transfers as a replacement for *read* and *write* system calls that copy data between user and kernel buffers is the introduction of the new system call *splice*. *splice* is meant to be a facility for user applications that allows establishing of data transfer paths, instead of explicit data copying using buffers located in the user space. This way, data, as long as it doesn't require processing by the user application, but merely transfers from one instance to another, can be moved entirely within kernel, eliminating most of the copying. Further, these transfers can be done asynchronously, with user application returning from the system call and continuing its operation while a kernel thread would control the actual transfer. The concept of *splice* could be seen similar to the STREAMS-based pipe implementation that interconnects two file descriptors leading to user, while *splice* interconnects two devices (or protocol stacks) within kernel, controlled by the user process.

splice interface is very similar to the *sendfile*, the user application must already have two file descriptors opened to the input and the output device. Unlike *sendfile*, *splice* allows interconnection of two arbitrary file descriptors, not just file-to-socket transmission. *splice* call takes the two file descriptors and desired transmission length (or a constant that defines transmission to the end of file) as arguments. Depending on file descriptor configuration (which can be done using fcntl), the call can be blocking for duration of data transfer, or asynchronous. In latter case user application will be informed of transfer termination with a signal (SIGIO).

During transfer, *splice* mechanism will alternately issue reads and writes on associated file descriptors, and is able to reuse read buffers for the writes. It also employs a simple flow control by using predefined watermarks for pending write requests.

Performance measurements done by testing file transfers from one disk to another showed about 30-70% improvement in throughput and almost halving of CPU load during transfer [8].

3 Kernel Bypass Techniques

3.1 Direct Hardware Control from User Applications

One of the most performant techniques is to allow direct access to device's memory by application or libraries running in user mode. This way it is possible for main data processing path to bypass the kernel almost completely, with exception of necessary virtual memory configuration tasks. On part of usermode implementation, it is possible to design extremely performant networking stacks optimized for a very specific task, for example for an implementation of the message passing interface (MPI) or remote shared memory in a cluster computing system [9, 10]. Such library could be designed to prepare buffers for send and receive in the most efficient way and implement flow control optimized for the protocol used. On the hardware side, it would be possible to take advantage of recent NIC designs equipped with fully programmable controllers, and use custom firmware to implement not only basic processing tasks like checksumming, but also parts of networking stacks, thus reducing the main CPU load.

Needless to say that using this technique breaks the hardware abstraction, the one of the most important aspects of modern operating systems. Further, the NIC controllers usually utilize a less powerful CPU for example a MIPS-architecture processor with simplified instruction set (without unnecessary features such as floating point calculations etc.) and also do not have much memory to hold a complex software. The implementations are therefore usually restricted to specialized protocols on top of Ethernet with much simplier design than TCP/IP [11, 10], and are mostly used in the LAN environment, where packet losses and corruptions are exceptional, which makes complex packet acknowledgement and flow control unnecessary. Since such an implementation would also require design of custom NIC firmware, they are hardware-dependant. In order to provide a reliable platform for design of such implementations, a standard was proposed by a few network interface manufacturers, the Virtual Interface Architecture (VIA) [12]. VIA defines functions, data structures and semantics for direct device access and is implemented by few companies such as GigaNet, ServerNet and Finisar for their hardware.

Direct hardware access technique respective VIA impose various limitations on the application design compared to the traditional communication design. Since data transfer from and to device is done with DMA, the user pages containing data buffers must be pinned by the operating system (prevented from swapping and changing their physical addresses). The performance tests have shown that pinning the user pages may require about as much effort as copying the data with CPU [10]. In order to avoid frequent calls to the operating system, applications must allocate and register a persistent memory pool to be used for data buffers. This also allows the network adapter to keep track on the receive buffers of a specific application, since the payload of received packet must be directed to correct application's buffer, according to recipient identification in the packet header. From the application's point of view, this allocation represent creation of the “virtual interface”, hence the name of the technology standard.

The buffer management for a single application is implemented in form of a send and receive queue, where it posts descriptors of buffers filled with data ready to send and empty buffers ready to receive, to the virtual interface. VIA requires receive buffers to be present at the time the packet arrives and both receive and send buffers to be of fixed size. These limitations unfortunately may lead to the need of an additional CPU copy in user space in order to transfer data between aligned, fixed-size send or receive buffer and a generic user data buffer unless the messages can be processed in-place. Since receive descriptors are stored in the host's memory, these need to be retrieved by the NIC using DMA accesses which imposes another impact on overall performance.

Performance tests on a 32-node cluster of 450MHz-PentiumII PCs have shown that VIA-enabled gigabit network interfaces are capable to achieve near 100% of line rate when transferring MPI or raw data packets with latencies as low as 20 microseconds for small packets. But it also showed that an additional CPU copy principially needed by applications using VIA may reduce bandwidth by 20-30% and double the latency [10].

The technique of the direct user access to hardware shows high performance, however its area of application is restricted to very specialized cases like node communication in clusters or network storage systems. It requires custom hardware and specifically designed applications, but relatively small changes in the operating system kernel, that can be easily implemented in form of a kernel module or device driver. Direct access to the hardware may impose serious safety issues, as buggy applications may deplete limited hardware resources and indirectly affect other applications using the same device while kernel has no direct control of it. Virtual Interface Architecture provides a more standardized and safer API and helps avoiding firmware redesign, but has its drawbacks in area of buffer management.

3.2 Kernel-Controlled Direct Data Transfers

A more compatible and safer solution that falls in the same class of techniques is the kernel-controlled direct data transfers between user buffers and hardware. Instead of copying data between user and its own buffers, kernel may arrange DMA transfers to and from memory areas of the application buffers. This technique, again implies that the kernel only acts as a broker between application and devices and doesn't partake in actual data processing, which may require massive hardware support in case of networking protocol stacks. An important advantage is that the call interface and semantics should not change for the user application, the actual implementation only affects the kernel itself. Should a requested transfer be impossible due to some implications, the kernel may fall back to the less performant, traditional way using its own buffers to store the data temporarily, transparently to the application and the hardware device.

The actual implementation however has to deal with important issues, for example, the pages of the user process involved in the DMA transfers must be pinned (made non-swappable) during the transfer and unpinned after it. The cache lines buffering respective memory locations also must be flushed to ensure the consistency of data before or after the DMA transfer. These obstacles may lead to significantly lower performance then expected, as the *read/write* semantics do not hint the user buffers as memory that can be involved in DMA transfers, unlike kernel buffers. Since pages of the user buffers may be located at arbitrary positions in the physical memory, some poorly implemented DMA engines may have addressing limitations and have trouble accessing those areas. Some technologies like IOMMU in AMD64 architecture may allow workarounds for these limitations by remapping DMA address to the physical address in memory, but in turn raise portability issues, as other architectures, even the Intel's variant of 64-bit x86 architecture EM64T does not posess such a unit [15]. Further, other limitations may exist for DMA transfer alignments, disallowing use of arbitrary buffer address specified by the user application.

The success of such technique also depends on whether the transfers are synchronous or asynchronous. Former case is usually true when accessing disk storage for read or write - the application may block on system call until the transfer is completed, and what's more important, the buffer is definitely specified by the call semantics. In case of network transfers however, while write operation may be blocking as well, the read operation is asynchronous - the packets can be received before the read system call is issued and the kernel doesn't have other choice but to store the payload in its own buffers and copy it later. In chapter "Dynamical Remapping with Copy-on-Write", similar difficulties are also discussed in regard to ZeroCopy technique that uses address remapping.

Many practical difficulties surrounding this ZeroCopy technique unfortunately lead to a very limited adoption in current operating systems. Since implementation of asynchronous transfers and networking seemed infeasible, it is mostly used to implement fast "Raw Disk I/O" transfers [5].

4 Optimizing Data Transfers Between Usermode and Kernel

The goal of the techniques presented until now was to avoid the copying between user and kernel buffers, which is traditionally implemented using CPU copies. As we have seen, their application is usually restricted to specialized cases, where we can do without data processing either in kernel or in user space. The class of ZeroCopy techniques presented in this chapter is aimed to maintain the traditional concept of passing data between user and kernel, while optimizing the transfers themselves.

We've seen that the data between software and hardware can be offloaded to DMA transfers, but in our case where data is transferred between user and kernel software, there is no such copy facility. On the other hand, the virtual memory architeture present on the most modern CPUs and used by operating systems suggests the possibility to virtually copy and share memory contents, although with relatively large granularity of 4 or 8 kilobytes, by remapping pages on different virtual locations, also between user applications and kernel. In this chapter we discuss two such techniques that take advantage of virtual memory remapping.

4.1 Dynamical Remapping with Copy-on-Write

As shown before, the synchronous write and read operations can be achieved by using user buffers directly, in case of the network transfers however, data needs to be copied to the kernel buffers, so it can be processed asynchronously by the networking stack. For this purpose, we can use the virtual memory operations to map the pages of the user buffers into the kernel address space. The main problem with this approach however, is that once the user application returns from the write or sendto system call, it may be able to modify data in its buffers while it is still being accessed by the networking protocol stack. Although no explicit locking mechanism is provided in the traditional write semantics, data protection can be implemented using the Copy-on-Write (COW) technique, which exploits an additional feature of MMU, the access protection of pages.

On the transmitting side, the implementation is relatively simple, pages associated with user buffers are mapped into the kernel space and marked write-only for the user. After the system call returns, both user application and networking stack can read the buffer, while it is shared between kernel and usermode. After the kernel has sent all data required, it can make them writable for user again. Should the application however try to write to its buffers before that, an exception occurs, and the kernel will have to copy the data to its own buffers and restore mapping on the user side. This technique requires that the actual COW events should be rare, as the total cost of a COW event will be significantly higher than a simple CPU copy. It should be also noted that even if the data transmitted from the buffer covers some complete pages, they cannot be simply replaced with free pages on the user side in order to prevent copy-on-write - the semantics of the write call must be preserved and they guarantee that the written data will be unchanged when the call returns. In practice, most applications will usually reuse the same buffer many times, so it is also reasonable not to unmap the pages from the kernel space after they are no longer needed, but keep the mapping to save management effort in the expected event that the same page will be used again. Unfortunately, such persistent mapping won't eliminate the need for costly page table traversal and TLB flush, since the read-only flag of the page will still needs to be changed every time the page is COW-locked and unlocked.

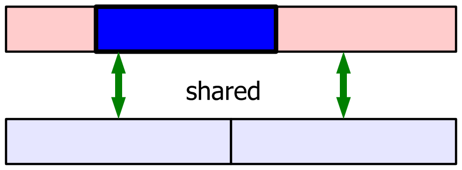

Figure 5: Send via address remapping with Copy-on-Write

The implementation of a similar technique on the receiving side however must deal with much more complicated issues. The semantics of the read call will only allow the kernel to know where the payload of a received packet should be written if the read was issued before the arrival of the packet and the application is blocked. In such case however there's also no need for page remapping, given the adequate support of the network interface card, it will be more reasonable to directly write the payload into the user buffer.

Should the receive be done asynchronously, there's no user buffer to write to until read call is issued, and the kernel must store the packet in its own buffer. Now there is still a possibility to efficiently move the data into the user space by flipping the mappings of pages of the kernel buffer and the user buffer, but only if the received data covers a complete page (otherwise we would still need to copy the rest) and the data in the kernel buffer is properly aligned to the address specified by the application. Both of these requirements are almost impossible to achieve, unless the networking protocol is somehow forced to transmit packets that match page size exactly [5].

Both the copy-on-write technique for the transmitting and page flipping for receiving side were implemented in Solaris, but due to mentioned restrictions on the receiving side, support for page flipping was eventually removed. Implementation itself required changes in the key networking code components, but were very trivial, basically it was only needed to replace the calls to *copyin* with *cow_copyin* [5].

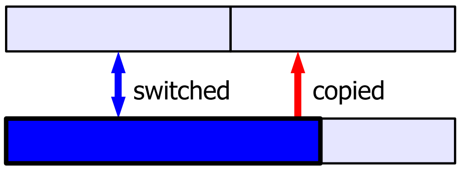

Figure 6: Receive via page flipping

4.2 Buffer Sharing

Another possibility to implement fast data transfers between user and kernel is to use pre-mapped shared buffers. A framework was proposed under the name “fbufs” by Druschel and Peterson [16] and implemented for Solaris by Thadani and Khalidi [17]. The main idea lies in a cardinal change of the API and semantics, where applications and kernel drivers strictly use the fbufs framework to communicate data between user application and kernel and for in-kernel communication.

The fbufs framework uses a per-application pool of buffers mapped in both user and kernel address space or creates them when necessary. With virtual memory operations done only once upon creation of the buffer, the fbufs concept efficiently eliminates most performace impacts of memory coherency maintenance.

In order to use the fbufs framework, both applications and kernel drivers need to use the new API respective DDI functions, that mimic the traditional interface. An application wishing to send data may obtain an fbuf from the pool, fill it with data and send through a file descriptor using the new *uf_write* API method. Since an fbuf is basically just a memory area, they can be used as the storage memory in the conventional mblk_t structures and propagated transparently through the STREAMS framework. STREAMS modules thus would not require modifications, they can duplicate, discard and even chain the fbufs-based mblks with normal mblks when splitting data and appending packet headers using traditional functions. The driver at the end of the stream also frees the fbufs-based mblk or message using traditional freeb respective freemsg functions [4].

Implications with fbufs framework arise on the receiving side: in order to allocate fbufs-based mblks, the device driver must use *uf_allocb* DDI function instead of traditional *allocb*, and what’s more important, it must be able to determine the correct fbufs pool to allocate from, using data from received packet which can be considered a violation of layering concept in networking stacks. The application that receives data in an fbuf also may still need to copy the data from it to another buffer in order to reassembly a data stream that arrives in packets. Received fbufs can be kept by the application for a while, reused to send another data, or simply returned to the pool.

In order to maintain the traditional write semantics, the application must be prevented from writing to fbufs currently processed in the kernel. Since introduction of automatic protection scheme like copy-on-write would basically nullify the advantages of having pre-mapped memory in many cases, the fbufs framework employs mandatory locking of

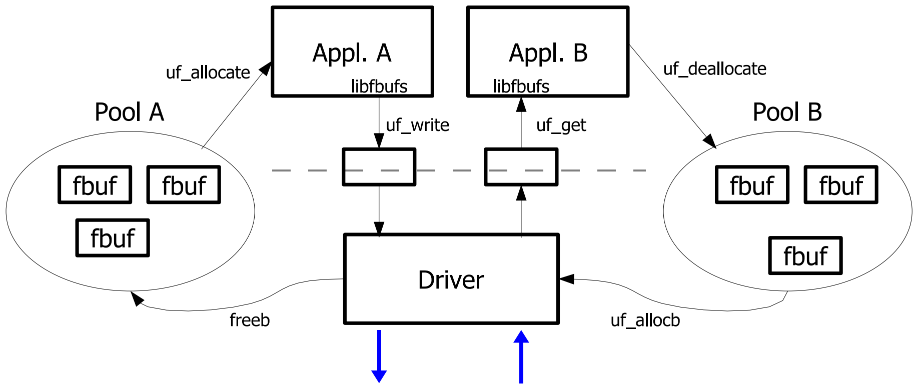

Figure 7: fbufs architecture

the resource instead - the application must be designed in a way that fbufs sent to the kernel are considered to be no longer in posession of the application, although the memory is still mapped in the address space [17].

5 Summary

As we have seen, many ZeroCopy techniques were proposed and implemented, but not many of them have found wide adoption in the actual operating systems. Some of the techniques such as fbufs sound promising in many aspects, but require a radical change in API semantics, and also in drivers, which prevented them, in addition to some unsolved implementation problems, from leaving the experimental stage. Although good performance improvements could be shown on a test system, converting the complete install base to the new architecture seems an impossible task. The strategy of using preallocated shared buffers also suffers from granularity issues - data delivered in such buffers to application might still need another copy to destined location, unless it can be processed in place or reused in communication.

The dynamic address remapping techniques require few changes to the operating system and usually none to user software, but the current virtual memory architectures are not well-suited for frequent remappings of the virtual memory. In order to visualize the performance impact, we must consider that current designs usually use multi-stage address translation, AMD64/EM64T architecture for example uses a 4-stage hierarchy of page tables [18] in 64-bit mode, which would require at least 4 memory accesses to different locations. With cache line size of 64 bytes, such access would equal prefetching of 256 bytes, with necessary modifications in operating system's own tables only adding to this overhead. After remapping, TLB flush is required to ensure correct memory address translation, and virtually-mapped L1 cache must be also flushed to ensure coherency. On SMP systems, the same must be done for other CPUs by issuing interprocessor interrupts that add a considerable overhead.

In the regular network traffic, a wide spectrum of packet sizes can be expected, with largest size of an Ethernet packet being 1500 bytes. In practice, only a little portion of regular traffic would be even feasible for ZeroCopy implementation with address remapping techniques, as virtual memory operations' cost would outweight the cost of the default CPU copy, except for large packets. Separate protocol implementations for different packet sizes would be quite complicated to coordinate and debug, and may even deplete instruction cache sizes [7]. It should be also taken in consideration that CPU copy of small units touches data, forcing its prefetching into caches, which can be very useful if the arrived data is expected to be processed by the application shortly afterwards.

Often, an extensive hardware support is required to fully eliminate memory accesses by CPU, and since not every commodity hardware offers them, or even implements all features correctly, kernel developers are seldom motivated to implement additional processing paths in central drivers and modules, unless performance improvements are convincing.

Unfortunately, the adoption of ZeroCopy in regard to networking is encumbered by many architectural limitations, such as the architecture of virtual memory and network protocols, and is still limited to very specialized cases like file serving and high-bandwidth communication with specialized protocols. The adoption of ZeroCopy in disk operations seems to be a lot more feasible, mostly due to their synchronous nature and usually large, page-granular transfer units, as opposed to typical network traffic.

References

[1] Christian A. Kurmann, *Zero Copy Strategies for Distributed CORBA Objects in Clusters of PCs*, Hartung-Gorre Verlag Konstanz (2003).

[2] A. Romanow, J. Mogul, T. Talpey, S. Bailey, *Request For Comments 4297 : Remote Direct Memory Access (RDMA) over IP Problem Statement*, IETF (2005)

[3] Dragan Stancevic, *Zero Copy I: User-Mode Perspective*, Linux Journal (2003), http://www.linuxjournal.com/node/6345

[4] *STREAMS Programming Guide*, Sun Microsystems (1997)

[5] H.K. Jerry Chu, *Zero-Copy TCP in Solaris*, SunSoft Inc. (1996)

[6] jamal, Linus Torvalds, *Is sendfile all that sexy? (Discussion thread at the Linux-Kernel Mailing List)*, Linux Kernel Mailing List (2001), http://www.cs.helsinki.fi/linux/linux-kernel/2001-02/0106.html

[7] Ingo Molnar, Alan Cox, Jeff V. Merkey, Linus Torvalds, *zero-copy TCP (Discussion thread at Linux Kernel Archive)*, Linux Kernel Archive (2000), http://www.ussg.iu.edu/hypermail/linux/kernel/0009.0/0210.html

[8] Kevin Fall, Joseph Pasquale, *Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability*, USENIX Winter (1993)

[9] Piyush Shivam, Pete Wyckoff, Dhabaleswar Panda, *EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing*, (2001)

[10] X. Liu, *Performance evaluation of a hardware implementation of VIA*, Tech. rep., U. of California in San Diego (1999)

[11] Mohammad Banikazemi, Bulent Abali, Lorraine Herger, Dhabaleswar K. Panda, *Design Alternatives for Virtual Interface Architecture and an Implementation on IBM Netfinity NT Cluster*, Journal of Parallel and Distributed Computing vol. 61 (2001)

[12] *Virtual Interface Architecture*, http://www.viarch.com/

[13] Thorsten von Eicken, Werner Vogels, *Evolution of the Virtual Interface Architecture*, IEEE Computer 31 (1998)

[14] Ian Pratt, Keir Fraser, *Arsenic: A User-Accessible Gigabit Ethernet Interface*, INFOCOM (2001)

[15] corbet, *DMA issues, part 2*, LWN.net (2004), http://lwn.net/Articles/91870/

[16] P. Druschel, L. Peterson, *Fbufs: A High-Bandwidth Cross-Domain Transfer Facility*, Proceedings of the 14th ACM Symposium on Operating Systems Principles (1993)

[17] Moti N. Thadani, Yousef A. Khalidi, *An Efficient Zero-Copy I/O Framework for UNIX*, Sun Microsystems Inc. (1995)

[18] *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*, Advanced Micro Devices (2005)