Performance Tradeoffs for Static Allocation of Zero-Copy Buffers

Pål Halvorsen, Espen Jorde[†], Karl-André Skevik, Vera Goebel, and Thomas Plagemann
IFI, University of Oslo, P.O.Box 1080 Blindern, N-0316 OSLO, Norway

E-mail: {paalh, karlas, goebel, plageman}@ifi.uio.no, [†]espejor@online.no

Abstract

Internet services like the world-wide web and multimedia applications like News- and Video-on-Demand have become very popular over the last years. Due to the large number of users that retrieve multimedia data with high data rates concurrently, the data servers represent a severe bottleneck in this context. Traditional time and resource consuming operations like memory copy operations limit the number of streams that can concurrently be transmitted from the server. To avoid this bottleneck and make memory and CPU resources available for other tasks, i.e., more concurrent clients, we have implemented a specialized zero-copy data path from the storage system through the communication system out to the network. In order to perform better than existing related approaches, we remove overhead of memory related operations by statically allocating the necessary buffers. In this paper, we describe the design, implementation, and evaluation of the zero-copy data path. Our results show the potential for substantial performance improvement when moving data through the communication system without any copy operations and using statically allocated buffers.

1 Introduction

There has been a tremendous growth in the use of multimedia Internet services, and in particular, applications like News-on-Demand and Video-on-Demand have become very popular. Thus, the number of users, as well as the amount of data each user downloads from servers on the Internet, is rapidly increasing. Current mid-priced personal computers are capable of handling the load that such multimedia applications impose on the client system, but in the Media-on-Demand (MoD) servers, the potentially (very) high number of concurrent users retrieving data represents a problem. In MoD servers in general, commodity operating systems represent the major performance bottleneck, because *operating systems are not getting faster as fast as hardware* [12]. There are two basic orthogonal approaches for this problem: (1) development of an architecture for a single server that makes optimal use of a given set of resources, i.e., maximize the number of concurrent clients a single server can support; and (2) combining servers, e.g., in a server farm or cluster, to scale up the number of concurrent users.

In the INSTANCE project, we concentrate on the first approach of developing a new architecture for single servers. Our target application domain are MoD applications, and we want to support multiple concurrent users each retrieving a high data rate multimedia stream. The operating system and server architecture must therefore be improved and optimized. Crucial issues of server design and implementation include copy operations and multiple copies of the same data element in main memory [13]. Major bottlenecks in high throughput systems are therefore the *read()* and *sendto()* system calls (and their equivalents), which copy data between the kernel memory region and an application buffer in user space. This is expensive for various reasons [3]:

- The bandwidth of main memory is limited, and every copy operation is effected by this.
- A lot of CPU cycles are consumed for every copy operation. Often, the CPU must move the data word-by-word from the source buffer to the destination. This means that the CPU will be unavailable during the copy operation.
- Data copy operations affect the cache. Since the CPU accesses main memory through the cache, useful information resident in the cache before the copy operation is flushed out and replaced by the data being copied.

To avoid the memory copy bottleneck, several mechanisms have been proposed (for a detailed state-of-the-art overview, see [13]), using mechanisms like page remapping and shared memory. In INSTANCE, we aim at a highly specialized mechanism to move data through the server as fast as possible and with minimal resource consumption. Most of the existing mechanisms support all general operating

system operations. However, in INSTANCE, we can make some simplifications and optimizations. Our specialized zero-copy data path can therefore be tuned to optimize performance for the main task in INSTANCE, i.e., transmitting data from the storage system through the communication system out to the network. This means that only streaming mechanisms like *splice()* [4], *sendfile()* [7] and *MMBUF* [2] are of interest in our server system. In order to minimize the kernel operations of such streaming mechanisms, we have designed and implemented an even more specialized mechanism that removes overhead like per-operation allocation of memory and variables by statically allocating the necessary buffers.

In the past, zero-copy solutions have been extensively studied before in various contexts. Our memory usage does not outperform similar zero-copy data paths, but in addition to removing copy operations, we do save CPU cycles compared to other approaches. It is the goal of this work to illustrate some of the compromises and tradeoffs using static versus dynamic buffer allocation, and we present in this paper the design and implementation of our zero-copy data path. Additionally, we give a short evaluation of the system and present measurement results. The remainder of this paper is organized as follows: Section 2 presents our design. In Section 3, we describe our implementation based on NetBSD. In Section 4, we present the results from our performance experiments. We summarize and conclude the paper in Section 5.

2 Design Considerations

In our scenario, the application process itself in user space does not need to access the data. We can therefore make some simplifications and optimizations, because there are no updates on the data being sent, and data will be sent periodically consuming in each period a given set of resources.

Furthermore, the introduction of dynamic resource management is motivated by the need for efficient use of scarce resources on a machine. The compromise is that dynamic allocation often results in less predictable performance and makes it harder to make reservations. In a MoD application delivering timely data, this is an important issue. We have chosen to design and implement a static resource allocation to both enable an efficient reservation scheme and reduce overall processor cycle requirement per stream. Using a static buffer also gives a reduction in allocation and deallocation cost, because it is performed only at the beginning and end of the life of a stream and not for each packet. Thus, buffer allocation is done with static buffers pinned in main memory, i.e., static `mbuf`s and `buf`s, which are the buffer structures used by the networking and file subsystems, and static data buffers.

In summary, the basic idea of our concept is to create a fast in-kernel data path, i.e., removing the copy operations between the kernel and the application running in user space, and to remove as many instructions per read and send operation as possible. The I/O operations in the storage and the communication system can use different sizes; allowing them to be optimized separately. This makes supporting buffer-streams with variable bit rate easier. The storage system can be optimized for reading as fast as possible, while send parameters can be adjusted to achieve constant or variable bit rate. The scheme is based on the assumption of a memory area shared between the file and networking systems. It does not preserve copy semantics as opposed to systems such as Genie [1]. Changes made by the file system code to a buffer will be visible to the networking code.

3 Implementation

3.1 Data Path

The basic idea of our in-kernel data path is depicted in Figure 1, where both file system and communication system share a memory region for data. The application is removed from the data path and will not have access to the data. If this is required, the existing mechanisms in NetBSD should be used.

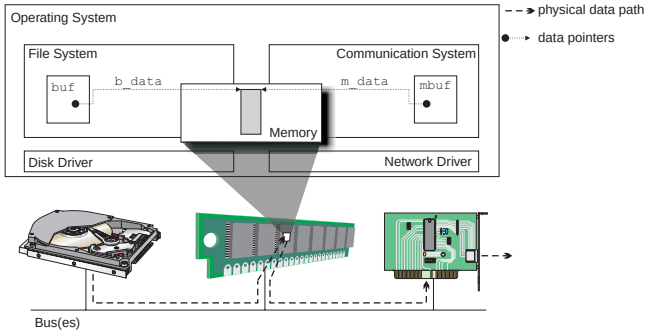

Figure 1. Basic idea.

When a user requests data from the disk, the file system sets the `b_data` pointer in the `buf` structure to the shared memory region. This structure is sent to the disk driver, and the data is transferred into the `buf.b_data` memory area. Furthermore, the communication system also sets the `m_data` pointer in the `mbuf` structure to the shared memory region. When the `mbuf` is transferred to the network driver, it copies the data from the `mbuf.m_data` address to the transmit queue of the network card. Thus, the physical data path is reduced to a data transfer from disk to a shared memory area and a transfer from memory to the network card. This means that we have no physical in-memory

copy operations, not even between the kernel sub-systems. The logical data path (or control path), however, goes from disk to the file system, and from the file system to application process. From the application process it continues to the communication system, and then to the network card. This is identical to the traditional data path using the *read()* and *sendto()* system calls.

The second idea of our design is to reduce the number of instructions for each *read* and *send* operation. This is done by statically allocating the shared memory region and all the `buf` and `mbuf` structures, in an INSTANCE-buffer, as shown in Figure 2. This means that the time normally spent on allocating memory and variables is saved in our system, and the transfer of the data pointer between the file system and communication system can be done during the stream set up. All the `b_data` and `m_data` pointers are set statically into the memory region. The size of the data region for each `buf` and `mbuf` depends on the requested disk read and network packet size, because each `buf` is used for one disk request, and each `mbuf` is sent as one packet. To ensure that we do not have any fragmentation in the data area, which might result in empty packets, each disk request reads one disk block. To optimize read performance in an application performing mostly sequential read operations, we use a file system with a 64 KB block size. To save time for disk requests and to be able to send data while we also read new data from the disk, we will have several alternating INSTANCE-buffers in one stream. In the current implementation, we have two buffers alternating between read and send operations.

Since all the resources are statically allocated, we must take special care when using these buffers in the kernel subsystems. For example, when a packet is sent, the disk driver calls `m_freem()`, which frees the transmitted `mbuf` structure and inserts it into the pool of free `mbuf`s. To avoid mixing our buffers with the buffers exclusively managed by the sub-systems, we use special flags in the `buf` and `mbuf` structures and check for these flags in the traditional kernel functions.

3.2 Application Programming Interface (API)

The zero-copy data path is available to user processes through a new stream API:

***instance_alloc()*:** This system call opens the requested file, opens a connected socket, and allocates all necessary variables, structures, and data areas. If the requested memory resources cannot be allocated during the opening of the stream, the request is aborted as there are no available resources. Otherwise, all data pointers in the `buf` and `mbuf` structures are set to the shared data area, i.e., each `buf.b_data` pointer is increased with 64 KB, and each `mbuf.m_data` pointer is increased with the size of the MTU[1] (subtracted the size of the packet headers). This means that the buffer structures for file system I/O are set to read one disk block (64 KB which is the maximum read size), and the buffer structures for communication system I/O is optimized to the maximum packet size in the underlying network, i.e., the amount of data in each packet is pre-fabricated and the packet size is optimized for performance where no IP fragmentation appears. Thus, in addition to allocating the necessary kernel resources, this system call performs operations equivalent to calling the *open()*, *socket()*, and *connect()* system calls.

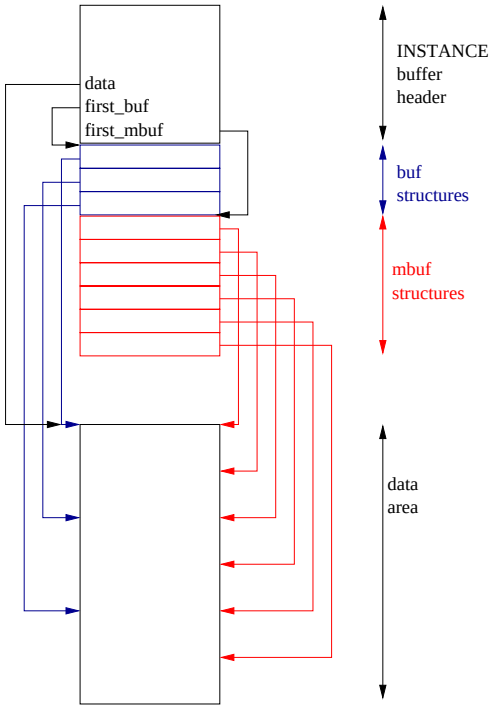

Figure 2. Illustration of an INSTANCE-buffer.

***instance_free()*:** To close the stream and free all the used stream resources, this system call replaces the traditional *close()* call.

***instance_read()*:** To read data from the disk into the alternating stream buffers, this stream version of the *read()* system call retrieves data from the disk. First, all empty buffers are filled. In later operations, the whole buffer is filled in the first read operation to the selected buffer regardless of the requested read size. Thus, we use read-ahead prefetching of data by sending all the

[1] We have used the NIC MTU to at least avoid fragmentation on the first link, but one can send an "MTU request" to get the maximum path MTU to avoid any fragmentation.

`buf` structures to the disk driver which may then optimize the read requests, e.g., reduce seek time on the disk. The read offset in the buffer is set to the requested size, but the file offset in the process' `file` structure is set according to the number of bytes read. Consecutive read requests just check if the requested data is present in the buffer, i.e., the requested read offset versus the offset in the `file` structure. If the requested data is not present in the current buffer, the alternation mechanism is used, and the other buffer is used.

***instance_send()*:** This system call sends data from a selected buffer to the communication system and corresponds to a stream version of the *send()* system call. Unlike *instance_read()*, which always reads data to the whole buffer, *instance_send()* sends the amount of data requested by the user, i.e., as the packet size is prefabricated, the amount of data to send is rounded up to the nearest MTU size. The *send offset* in the buffer is set according to the amount of data sent.

The throughput that a single stream can achieve on a unloaded system, may be limited by a strict waiting for all packets to be sent on the network. This means that before we can read new data into a buffer, all the data must be sent so that we do not overwrite packets waiting to be copied to the network card transmit ring. This is because we allocate all the resources for a stream statically and reuse the allocated set of resources for each operation. As opposed to the traditional system calls, which return after inserting the packet into the queue in the driver, we therefore may experience a delay waiting for the packets to be sent. We will however save the time to allocate memory and variables used for each call, and reduce the number of executed instructions.

4 Evaluation

Several tests are performed to measure the performance of our zero-copy data path compared to the use of the traditional (*read()* and *sendto()*) system calls. The measurements are made on a *Dell Precision 620*, with a 1 GHz *Intel Pentium III Xeon* processor and 256 MB RAM. The operating system installed on the machine is NetBSD 1.5U extended with the INSTANCE zero-copy data path. The file system is located on a 36 GB SCSI disk connected to a *Ultra 160 SCSI Controller*.

In our experiments we transmitted a test file of 1 GB from a *ffs* partition with a 64 KB block size, which is required for a file system used with our zero-copy data path.

4.1 Transfer Rate

In the first test, we measured the maximum throughput the system can achieve. Two test programs are used; both programs read a 1 GB file from the disk and transmits it via UDP, using our zero-copy data path and *read()*/*sendto()* respectively. The destination address is on the same host, but the packets are discarded during input processing. This allows the output performance to be measured without being overly influenced by input processing of UDP/IP packets, or overhead from the network interface card. The machine is fast enough to easily saturate a 100 Mbps Ethernet, which would reduce the usefulness of our measurements[2].

However, our results also show that our one-disk storage system is a large bottleneck. Tests with the file system benchmarking tool *Iozone* [11] showed that the total throughput performance of the storage system of about 320 Mbps which is approximately equal to the achieved transfer rate using both the zero-copy and the traditional data path. Our networking performance, measured with the *netperf* benchmarking tool [8], show that the these operations clearly are not the bottleneck. Thus, to improve the system I/O performance faster disks are needed[3]. Integrating a chain of high-speed disks and repeating these measurements is planned for the near future. Nevertheless, our experiments with a similar (dynamic) zero-copy data path in [6], show that the removal of copy operations can increase the throughput significantly, and the number of concurrent clients are approximately doubled. Thus, if the amount of memory is not a bottleneck, but CPU cycles spent performing copy operations and introduced latency are, we might expect a similar improvement.

4.2 Static Buffer Allocation

To see the time saved by allocating and freeing memory for data, consider the following example. Each buffer has a 64 KB data area. When transmitting a 1 GB video file, the kernel will perform 16 K *alloc* and *free* operations. Depending on the memory management mechanism, i.e., whether we use traditional *malloc()*/*free()* or a pool mechanism, this will consume a varying amount of CPU resources. Our performance measurements show that using the NetBSD pool mechanism, each of theses operations take 0.15 μs, and using *malloc()* and *free()* these operations spend 5.80 μs and 6.48 μs of the CPU time, respectively. Furthermore, assuming the `buf` and `mbuf` structures use the pool mechanism (`mbuf` management is implemented like this in NetBSD),

[2] In a real deployment situation one would use a Gbps network, e.g., the Intel IXP2800 network processor [9] provides 10 Gbps packet forwarding.

[3] The performance of a chain of high-speed disks or a RAID system using disks like the Seagate Cheetah X15 [14], each capable of achieving a minimum data rate of 299 Mbps, can easily deliver the requested data rate.

we additionally need one `buf` structure to read a 64 KB disk block and two `mbuf` structures for each packet, i.e., one for the header and one for the data. If we assume a 1500 B packet size, we need 89 memory elements for subsystem buffer structures when retrieving a 64 KB data area from disk and sending it to the network. The allocation and deallocation of the data area takes 0.3 μs[4] or 12.3 μs depending on the mechanism used per operation, and pool get and put operations for the storage and networking buffer structures will additionally consume 26.7 μs per operation. In summary, static allocation will in this example at least save 27 μs per sent 64 KB data element or about 0.44 s totally for the entire file. Additionally, we save the time to set the data pointers, flags, etc. in the `buf` and `mbuf` structures (not measured).

The tradeoff for saving the time for allocating and freeing memory is that the read operation must block if all data from previous operations is not yet transmitted onto the network from a buffer that is reused, e.g., we are transmitting as fast as possible. This is because unsent data will be overwritten if a new read operation is issued on the same buffer. However, the achieved data rate per stream (limited by the storage system - see previous section) is still far above the requested data rate for video playout, e.g., MPEG-2 DVD requires in average 3.5 Mbps and maximum 10.08 Mbps [10]. Since our mechanism both saves a lot of resources and still fulfills the throughput requirements of a MoD server, we consider the benefits to outweigh the drawbacks.

4.3 No Copy Operations

Our zero-copy data path mechanism does not require that the I/O data is copied between the kernel and the application. It is expected that our zero-copy data path will require fewer system resources because of this. The CPU usage of the test programs is measured to see if this is correct.

Figure 3 shows the total time spent, i.e., both elapsed time user and kernel space, making the server transferring our test file running using the *time* program. It shows that less time is being spent in both kernel and user space for the application using our zero-copy data path. The difference is especially large for kernel time with small buffer sizes. The reason that the overall throughput remains constant is that other factors such as I/O operations (i.e., disk I/O) dominate. Basically, this experiment shows that our zero-copy data path uses fewer CPU resources than the traditional approach.

The final point we wish to examine is the effect of CPU load. The test programs are used on a system with heavy load to determine how this influences performance. The main task of the programs used to generate the background load is to increment a single value, which means that they do not need frequent access to different locations in main memory or on the disk. Four such programs run during the measurements.

[4]Note, however, that the pool mechanism does not support pool items larger than 4 KB making this solution useless unless we implement our own pool.

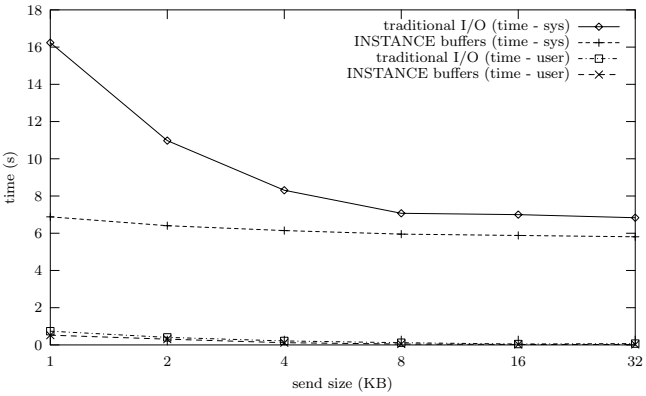

Figure 3. CPU time.

Figure 4 shows a comparison of traditional I/O with our zero-copy data path during high CPU load. Our zero-copy data path is clearly less influenced by CPU load than the traditional mechanism, with roughly half the reduction of the traditional approach, compared to an unloaded system. The traditional mechanism performs especially poor for smaller send sizes.

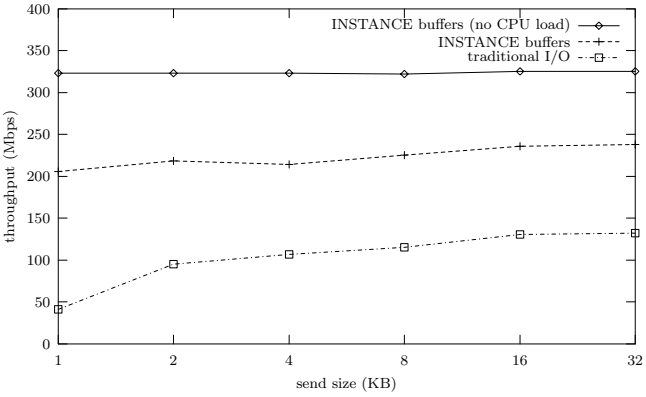

Figure 4. Performance during high CPU load.

In summary, our mechanism outperforms the traditional data path. Other (dynamic) zero-copy streaming mechanisms will probably have a similar overall server throughput, but our mechanism reduces the required number of CPU cycles.

5 Conclusions

In this paper, we have examined our implementation of a new data movement mechanism. It is designed to make more efficient use of the resources in a MoD system. It does this by eliminating the movement of data between the kernel and application; data is transmitted directly from disk onto the network. By using static buffers which are allocated only before the transmission starts, the per-packet cost is reduced.

The main task of a MoD system is to transmit the contents of a media object onto the network. We have compared our approach with the traditional mechanism using the *read()* and *sendto()* system calls to transmit the data over UDP, to determine how it performs in terms of resource usage and efficiency. We showed that our zero-copy data path requires fewer CPU resources and performs better when the system is under a high CPU load. We measured the maximum achievable throughput, but found no significant difference. This was due to the limited data rate of the disk in the test system. The test system only used a single disk, while a more realistic MoD system would probably store the files on a RAID system with a higher effective data rate. In conclusion, we have achieved the goals of the design, but some work still remains to be done on testing the system.

There are several issues which need to be examined in the future. Our zero-copy data path should be tested in an environment more similar to an actual production system, and it would also be useful to examine how it performs in comparison to I/O mechanisms which use similar techniques to reduce resource usage. Furthermore, our static allocation of `mbuf`s allows further improvements in the communication system in case of using header templates, e.g., as suggested in [6], where each static `mbuf` can contain the header template removing the block-copy operation of the template.

Finally, the question whether static or dynamic allocation of zero-copy buffers is preferable is dependent of the application scenario and is a trade-off between flexibility and CPU resources. A static allocation scheme should be considered if we perform CPU intensive operations in addition to streaming high-rate data. However, such a scheme may be inflexible, and in most cases, the performance gain might be traded for flexibility.

References

[1] Brustoloni, J. C., Steenkiste, P.:*"Effects of buffering semantics on I/O performance."*, Proceedings of the Second Symposium on Operating Systems Design and Implementation, Seattle, Washington USA, October 1996

[2] Buddhikot, M. M: *"Project MARS: Scalable, High Performance, Web Based Multimedia-on-Demand (MOD) Services and Servers"*, PhD Thesis, Sever Institute of Technology, Department of Computer Science, Washington University, St. Louis, MO, USA, August 1998

[3] Cranor, C. D.: *"The Design and Implementation of the UVM Virtual Memory System"*, PhD Thesis, Sever Institute of Technology, Department of Computer Science, Washington University, St. Louis, MO, USA, August 1998

[4] Fall, K., Pasquale, J.: *"Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability"*, Proceedings of the 1993 USENIX Winter Technical Conference, San Diego, CA, USA, January 1993, pp. 327–333

[5] Fujitsu Computer Products of America, INC.:*"MAJ Series Disk Drive Specification"*, http://www.fcpa.com

[6] Halvorsen, P.: *"Improving I/O Performance of Multimedia Servers"*, PhD Thesis, Department of Informatics, University of Oslo, Norway, August 2001

[7] Hewlett-Packard Company: *"Linux Programmer's Manual - sendfile"*, http://devresource.hp.com/-STKL/man/RH6.1/sendfile_2.html, May 2001

[8] Hewlett-Packard Company (Information Networks Division): *"Netperf: A Network Performance Benchmark, Revision 2.0"*, http://www.netperf.org

[9] Intel Corporation:*"The Intel IXP2800 Network Processor"*, http://www.intel.com/design/network/-products/npfamily/ixp2800.htm, June 2002

[10] MPEG.org: *"DVD Technical Notes - Bitstream breakdown"*, http://mpeg.org/MPEG/DVD, March 2000

[11] Norcott, W. D., Capps, D.: *"Iozone Filesystem Benchmark"*, http://iozone.org

[12] Ousterhout, J. K.: *"Why Aren't Operating Systems Getting Faster As Fast As Hardware?"*, Proceedings of the 1990 USENIX Summer Conference, Anaheim, CA, USA, June 1990, pp. 247–256

[13] Plagemann, T., Goebel, V., Halvorsen, P., Anshus, O.: *"Operating System Support for Multimedia Systems"*, The Computer Communications Journal, Elsevier, Vol. 23, No. 3, February 2000, pp. 267–289

[14] Seagate, Disk Products by Product Number, http://www.seagate.com/cda/products/discsales/-index, May 2001