RICE UNIVERSITY

Zero-Copy and Zero-Mapping Asynchronous TCP Send

by

Romer Gil

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

[Signatures]

Alan Cox, Associate Professor,
Computer Science

Vijay S. Pai, Assistant Professor
Electrical and Computer Engineering

Willy Zwaenepoel, Professor
Computer Science

Houston, Texas

May, 2003
Zero-Copy and Zero-Mapping Asynchronous TCP Send

Romer Gil

Abstract

Attempts to build zero-copy systems have generally fallen in one of two categories: those supporting zero-copy transparently, using existing (UNIX) interfaces, and those introducing new interfaces. Neither approach has seen widespread acceptance. This thesis explores two ideas. First, it considers an optimization to the operating system that allows it to omit data mappings while performing zero-copy sends through checksum offloading network interfaces. Second, it proposes the design of a zero-copy and zero-mapping TCP send within the framework of an existing asynchronous I/O API. The zero-copy send eliminates all copies of data presented to the kernel for output and takes advantage of zero-mapping to further improve performance. Results on uniprocessor systems show that the proposed asynchronous send outperforms its copying counterpart by up to 40%. Zero-mapping showed to improve performance by up to 3% in the best case.
Acknowledgements

I acknowledge with deep gratitude the guidance and support given to me by my advisor, Prof. Alan Cox, during my stay at Rice University. His patience and direction were critical to the creation of this document and the development of my understanding of operating systems.

I would also like to thank Prof. Willy Zwaenepoel and Prof. Vijay S. Pai for taking the time to be in my committee and for providing me with valuable input.

Special thanks to many of my friends and collaborators during my stay at Rice University - Cheryl McCosh, Santashil PalChaudhuri, Anupam Chanda, Khaled Elmeleegy, Sumit Mittal, Rajnish Kumar, Anshuman Dasgupta, Juan Navarro, Sitaram Iyer to mention some. Their tolerance and moral support was pivotal in my endeavors at Rice.

Last, but certainly the most, I would like to thank my family - my parents, Romer and Esperanza, my brother, Jorge and my sister, Rossana. Their support and motivation has helped me at every single stage of my life.
4.4 Results ................................................................. 25
  4.4.1 Zero-Copy and Zero-Mapping AIO ......................... 26
  4.4.2 Zero-Mapping Benefits ......................................... 28
  4.4.3 Overhead and Performance Penalties ....................... 29
  4.4.4 Copy-on-Write Performance .................................. 32
  4.4.5 IO-Lite Performance ........................................... 34
  4.4.6 Summary ....................................................... 35

5 Related Work ....................................................... 36

6 Conclusions and Future Work .......................... 38
  6.1 Summary of Contributions .................................... 38
  6.2 Future Work ..................................................... 39
    6.2.1 SMP ......................................................... 39
    6.2.2 TCP Receive ............................................... 40
    6.2.3 TLB Avoidance for Disk I/O .............................. 40

Bibliography ......................................................... 41
Illustrations

2.1 Control Flow for Synchronous I/O ........................................ 10
2.2 AIO API ...................................................... 11
2.3 Control Flow for Asynchronous I/O ..................................... 12

3.1 Mbuf Structure .................................................... 17
3.2 Mbuf Chain Example ................................................... 18

4.1 Cumulative Distribution of Requests in terms of File Size ........ 24
4.2 Zero-Copy and Zero-Mapping Asynchronous Send ................ 26
4.3 Zero-Mapping Benefits for Asynchronous Send .................... 29
4.4 AIO Framework Overhead ........................................... 31
4.5 Zero-Copy Overhead ............................................... 32
4.6 Copy-on-Write Performance ......................................... 33
Chapter 1

Introduction

Over the last few years, there has been tremendous growth in the popularity of Internet services, and in particular of applications like video on demand. Today, standard computer systems have sufficient power to handle the load that such applications impose on the client side. On the server side, however, the computing power is often insufficient to handle the high volumes of traffic that the servers support. In heavily loaded servers, the operating system usually represents the performance bottleneck. One of the major problems in operating systems lies in the memory management and is caused by excessive data copying between applications and the kernel. An example of this is the send system call and all its derivatives, which copy data buffers from the user address space to the kernel. While such copying does lead to a simple interface, it also degrades performance because it uses CPU and wastes memory.

Attempts to build zero-copy systems have generally fallen in one of two categories: those supporting zero-copy transparently, using existing (UNIX) interfaces, and those introducing new interfaces. Neither approach has seen widespread acceptance. Getting programmers to adopt new interfaces is difficult, which has slowed the acceptance of newly proposed interfaces. Transparent approaches have traditionally been based on copy-on-write: the pages containing the I/O buffers are write-protected until the I/O is completed. This approach has problems as well. The VM manipulations are expensive, especially on an SMP. Copies may still occur, if the application writes to the pages containing the I/O buffers before the I/O completes. Alignment issues lead to further complications and occasionally additional copies.

Much of the earlier work on zero-copy I/O has been done in the context of syn-
chronous I/O. Modern high-performance applications, however, often use the recently introduced asynchronous I/O interfaces. These I/O interfaces provide a way to transparently implement zero-copy without copy-on-write. In particular, they contain primitives for notifying an application that the I/O is complete and the I/O buffers can be reused. Hence, the application must avoid modifying the buffers between the time the I/O starts and the time that it receives notification that the I/O is completed. Granted that the application obeys this protocol (which is inherent in asynchronous I/O), then no copy-on-write is necessary to implement zero-copy I/O.

Furthermore, conventional implementations of zero-copy I/O map the buffers into the kernel address space before the physical I/O operation is initiated. With conventional network interfaces, the mapping is necessary to allow the kernel to perform operations such as checksumming the data. With modern interfaces that offload such functions, there is seldom if ever any need for the kernel to touch or modify the data, and thus there is seldom any need to map the data in the kernel’s address space. In those cases, the kernel can perform I/O straight from user space, without an intermediate mapping in the kernel. Although the pages still need to be pinned in memory for the duration of the I/O, this operation does not involve the TLB and the page tables, and is therefore much less costly.

My thesis is that asynchronous APIs provide a good framework for efficient and transparent implementations of zero-copy and zero-mapping TCP sends. I show this by designing and implementing various zero-copy TCP send schemes and comparing them with a scheme based on the AIO API.

The contributions of this thesis are twofold. First, it proposes a modification to the operating system that allows it to omit data mapping while performing zero-copy sends through checksum offloading network interfaces, thus avoiding the high and often unnecessary cost of TLB and page table operations. Secondly, it proposes the design and implementation of a zero-copy and zero-mapping I/O system within the framework of an existing asynchronous I/O API. The zero-copy I/O eliminates
all copies of data presented to the kernel for output and takes advantage of the offloading capabilities of modern networking hardware to further improve performance, while remaining compatible with legacy hardware. Furthermore, the implementation integrates well with the operating system, without major modifications to the kernel or existing applications.

A prototype implementation of the asynchronous zero-copy and zero-mapping TCP send was implemented in FreeBSD, along with a few other zero-copy schemes with mapping and zero-mapping variations. In keeping with the goal of improving network server performance, all the evaluations were carried in the context of a web server. Results on uniprocessor (UP) systems show that the asynchronous zero-copy and zero-mapping send outperforms its copying counterpart by up to 40% under the most favorable circumstances, while it decreases performance by 10% in the worst case of small files. Zero-mapping showed to improve performance by 2% in most cases and up to 3% in the best case.

The rest of this thesis is organized as follows. In chapter 2, I present all relevant background information useful for later sections. In this chapter, I also discuss the AIO standard and present the API as a viable medium to implement a transparent zero-copy and zero-mapping I/O system. Chapter 3 explains the details of the design and implementation of the zero-copy asynchronous TCP send in the AIO API. In chapter 4, I present the experiments and results for the evaluation of the zero-copy asynchronous send as well as for the proposed zero-mapping component. Chapter 5 discusses the previous work performed in the area. Finally, in chapter 6 I present the conclusions and some ideas for future work.
Chapter 2

Background

Internet servers are based on what is commonly known as the client-server model. In this model, clients are the consumers and are those systems that make requests for data or services to the server. The servers are the providers, whose responsibility is to respond to those requests made by the clients. Typically, as the number of clients in the entire system increases, the servers become the bottleneck.

From a high level, a server works by receiving requests through its network interface, passing them up to the server application through the kernel, and then sending the response generated by the application out through the kernel and the network adapter. The key software components that determine the server’s performance are the server application and the networking subsystem within the kernel. Both of these are described in the following sections.

2.1 Server Application

The server application is the part of the server responsible for interpreting the request and generating responses sent to the network via the network interface. Typically, these applications feature multiprocess or multi-threaded architectures in order to allow for concurrency and some scalability. In such architectures, the application creates threads (or processes) either ahead of time or on-demand to handle the requests. Clients communicate with the server by sending requests to a predefined port on the server. The server picks up these requests as it polls a pre-established listen socket associated with its public port. As requests are taken off the listen socket, a new socket is opened and associated with the connection. This socket and the request are
then handed to the worker thread for processing and response to the client.

The processing at the thread level usually entails constructing protocol specific headers or some metadata and producing a response. This response can be in the form of dynamically generated content or more frequently static content in the form of files obtained from a cache or directly from the file system. After the application specific processing is done, the worker thread sends the response back to the network via a write system call, which is then taken by the kernel and handled by the network subsystem.

2.2 Network Subsystem

As the part of the kernel that manages the networking in the local system, its responsibilities range from managing the network interface and providing an interface to the physical device, to allocating and managing related data structures, and providing/receiving data from the applications. A high-level overview of the typical operation of the network subsystem during output follows.

Applications make network output (send) requests to the operating system by calling the write system call with a socket as the target file descriptor. Once the kernel traps the system call, it converts it to sosend, which is the output function for socket file types. Sosend initially performs some error checking on the request and then copies the data from the user supplied buffer into the network buffers contained in the kernel’s address space. Once this is completed, the kernel invokes the protocol layer corresponding to the socket descriptor (TCP socket). The first routine in the protocol layer, tcp_output, adds all the TCP headers and as many IP headers as it can by filling a new buffer and prepending it to the previously allocated buffers. In this step, the TCP checksum is calculated and stored in the TCP header. Tcp_output also invokes the next protocol routine, ip_output, which computes the IP checksum and fills the remaining fields in the IP header, determines the outgoing interface to which the datagram should be given, fragments the IP datagram if necessary, and
calls the interface output function. The interface routine does more processing and then adds the previously filled network buffers to the end of the output queue of the network card. If the device is not busy at that time, the interface routine invokes the network adapter’s start output (e.g. fxp_start, xl_start or em_start). If the device happens to be busy, its output routine processes the new data after it finishes with the buffers already on the queue.

In the typical sequence, the kernel incurs a data copy and two checksum computations, all of which traverse the MMU and the cache. Copying the user buffer into kernel space guarantees that the network subsystem attempts to send the correct version of the data, disregarding any modifications that may have been performed after the send system call is made. All these copies are useful for maintaining the correctness of the operations, however, copying often is harmful to performance for several reasons. As was noted earlier, data copying consumes CPU cycles that can be used for other processing. Add to this the fact that data copying itself suffers in performance because of limitations in memory bandwidth. Finally, since the copies happen through the CPU, they tend to pollute the cache with useless data, thus reducing the benefits of locality.

2.3 Zero-Copy Sends

All these inefficiencies discussed above have been addressed in previous research. The goal in these works has been to minimize the role of the CPU in I/O operations by avoiding data copies between the kernel and user space. Most of the research in zero-copy has gone in one of two directions: adapting zero-copy semantics to existing interfaces, therefore providing zero-copy transparently to user applications, or creating new interfaces that support zero-copy semantics. Each research direction is discussed below.
2.3.1 Transparent Approach

Most of the work in this direction has been based on copy-on-write (COW) techniques. Examples include Chu's [5] and Chase's [4] work on TCP zero-copy and Brustolini's Genie system [2, 3]. Usually, these schemes work by implementing an alternative version of the kernel copy routine uimove which is used to copy data from the process to kernel space. The modified uimove instead remaps the data transferred between a network socket and a page-aligned user buffer. This new mechanism is analogous to passing buffers by reference instead of by value, which is the default behavior. This way the API exposed to applications can be left unchanged since only the underlying mechanisms are modified.

There are performance problems with this approach however. The application may modify or even overwrite its send buffers after requesting the send but before the send completes. In order to prevent data corruption in such cases, the kernel must protect the pages with pending transmit COW. As the name implies the pages are copied in case the application tries to write to them. This implies that aside from the page remapping overhead, some COW-related overhead is caused as well.

Most of the software overhead occurs in the virtual memory system and includes looking up pages and updating data structures. Most of the overhead, however, is hardware-related and it includes reprogramming the MMU and flushing various caches to maintain consistency. All this overhead can be very expensive and sometimes comparable to data copy [11]. In the multiprocessor case, these overheads can be even more expensive due to TLB shoot-downs. Additionally, a significant amount of overhead is architecture-specific as it depends on machine configuration, cache architecture and TLB miss handling.

Aside from the overheads mentioned above, COW-based techniques can encounter other performance problems. One can imagine the degenerate case in which the application writes to all protected pages causing them to be copied. This would cause COW-based techniques to perform worse than normal sends because they are
forced to copy all pages after establishing all the protections.

2.3.2 New API Approach

This approach to zero-copy systems has also taken two directions. The main idea, however, is that they modify existing interfaces or provide entirely new ones to the applications thus making them less appealing to programmers. One of the directions on this front has been towards the design of unified buffer systems, while the other has been in the creation of monolithic or do-all system calls.

Unified Buffer Systems

These systems usually are built from scratch and consist of a pool of preallocated buffers that are intended to be shared among certain I/O subsystems in the kernel and the application. The I/O subsystems that have direct access to these buffers determine the entities in the kernel that are allowed to perform zero-copy operations. Examples of work in this area include IO-Lite [10] and Thadani and Khalidi’s zero-copy networking [11].

There is nothing conceptually wrong with these schemes; since they are designed from scratch, they offer good performance without major complications in implementation. IO-Lite for example offers 40% to 60% performance improvement while Thadani and Khalidi offer up to 40% improvements. Their problem lies in the fact that they modify the API exposed to applications. Therefore, they cannot be used without either modifying existing applications or educating programmers to use them in future software.

Monolithic System Calls - Sendfile

Due to the popularity of internet applications and the high demands for static content, there has been a push for specialized system calls to improve system throughput. As a result, most mainstream operating systems have adopted features like the sendfile
system call that concentrate on optimizing the handling of static content. In FreeBSD, sendfile is used to send entire files over network in a zero-copy fashion. The system call takes as arguments a file identifier, a socket, and some headers to be prepended to the file. Sendfile works by processing the file one page at a time to populate the network buffers and then send them. For each page, sendfile first makes sure that the data is in memory, causing a page fault to bring it in from disk if necessary. Once the data is guaranteed to be in memory, sendfile allocates a sendfile buffer (sf_buf) and maps the data into it. It then associates a network buffer with the sf_buf and calls the appropriate protocol send routine on the newly created buffer.

The sf_bufs are buffers, which like the network buffers are created once, ahead of time and are allocated and deallocated upon request by removing them from and inserting them into a kernel pool of available buffers. The only purpose of the sf_bufs in the sendfile implementation is to provide the data being transferred with a kernel virtual address, in case the kernel needs to access it to perform computations. Sendfile does a zero-copy transfer because instead of using uio_move or any other copy routine to pass the data to the network subsystem, it only remaps the data buffers into the sf_bufs, which are then handed to the network buffers as external data.

2.4 Synchronous vs. Asynchronous I/O

The multithreaded/multiprocess server model discussed previously, although easy to implement has a disadvantage. This type of architecture suffers significantly from the overhead caused by cache and TLB misses, scheduling, and lock contention that strain the operating system, and therefore degrades in performance as the number of threads becomes large. To improve scalability and performance, developers often use an event-driven approach to handling concurrency. In this model, a server consists of a small number of threads that loop processing pending events. Events may be generated by the kernel or the application, and generally correspond to network or disk I/O readiness and completion notification, timers or application specific events. This
programming model has been studied extensively and shown to benefit application performance in systems such as Flash [9], SEDA [12] and JAWS [7] among others. One limitation of this model, however, is that it assumes that event-handling threads do not block. Consequently, they must either perform asynchronous I/O or operate on non-blocking file descriptors, dealing with potentially blocking amounts of data in smaller chunks.

![Diagram](image)

Figure 2.1: Control Flow for Synchronous I/O

2.4.1 Synchronous I/O

Traditional output operations work in the following way from an application program’s perspective. First, and depending on the specifics in user space, the application produces a file descriptor to write to and a buffer to write from. The application then, via the write system call, hands the file descriptor and buffer to the kernel, which in turn blocks the application thread that made the request as shown in Figure 2.1. The kernel then services the request by programming the I/O device and getting data to it. Once the data transfer is completed, control turns back to the application.

The benefit of this mode of operation is that it is straightforward. Once the line of code for the request is executed, the application is guaranteed to have either
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aio_read</td>
<td>Allows the calling process to read from a given descriptor into a buffer.</td>
</tr>
<tr>
<td></td>
<td>The call returns immediately after the read request has been enqueued.</td>
</tr>
<tr>
<td></td>
<td><code>int aio_read(struct aiocb *iocb);</code></td>
</tr>
<tr>
<td>aio_write</td>
<td>Allows the calling process to write from a given buffer into a descriptor.</td>
</tr>
<tr>
<td></td>
<td>The call returns immediately after the write request has been enqueued.</td>
</tr>
<tr>
<td></td>
<td><code>int aio_write(struct aiocb *iocb);</code></td>
</tr>
<tr>
<td>aio_error</td>
<td>Returns the error status of the aio request associated with the iocb</td>
</tr>
<tr>
<td></td>
<td>structure.</td>
</tr>
<tr>
<td></td>
<td><code>int aio_error(const struct aiocb *iocb);</code></td>
</tr>
<tr>
<td>aio_return</td>
<td>Returns the final status of the aio request associated with the iocb</td>
</tr>
<tr>
<td></td>
<td>structure.</td>
</tr>
<tr>
<td></td>
<td><code>int aio_return(struct aiocb *iocb);</code></td>
</tr>
</tbody>
</table>

Figure 2.2: AIO API

an acknowledgement of completion or an error notice. This means that it is trivial to write the support code for the write at the application level. The drawback to this model is that the application thread that made the request gets blocked from execution for the duration of the write, preventing it from performing any other operations.

2.4.2 AIO

To alleviate the problem mentioned above, UNIX adopted the asynchronous I/O standard in the form of the aio API. This API includes two functions, aio_write and aio_read, that mirror their synchronous counterparts write and read, respectively. Additionally, the API contains functions to obtain progress and completion status for the reads and writes and functions to control the I/O operations after they have been requested. Figure 2.2 gives an overview of the main functions in the API.

Figure 2.3a shows the flow of an aio write operation. As in the synchronous
case, and depending on the specifics of the application program, a file descriptor and a buffer are generated in user space. From these two pieces of information and a few others, the application constructs an aio control block (aiocb). The aiocb is then passed to the kernel as an argument to the aio_write function, thus making the formal request for the output to the kernel. The aiocb is the operation identifier and is used to hold all information related to the I/O, such as file offset, number of bytes, type of notification, and priority in addition to the buffer and file descriptor as mentioned above.

![Diagram](image)

(a) request
(b) response

Figure 2.3: Control Flow for Asynchronous I/O

Once the write request is made, the kernel attempts to enqueue the operation in the aio system. The kernel then immediately returns control to the application, which receives the status for the queuing of its request. At this point, the I/O operation is only guaranteed to have been queued on the device and it may or may not have even started.

Figure 2.3b shows the flow of the kernel once the I/O is finished. As seen in the figure, the kernel notifies the application, which is then free to use any data resulting from the I/O and then logically close the whole operation by calling the aio_return on the identifying aiocb. The notification of completion is given to the application
in one of two ways. The application can either poll on the progress status of the I/O by continuously calling aio_error until the operation is marked completed or it can also configure the kernel to raise a signal upon completion of the I/O. Another more scalable form of notification is also available in FreeBSD in the form of the kevent notification mechanism [8]. The type of notification is specified at the time of constructing the aiocb to be given to the aio_write call.

During the time between the queuing of the request and the completion of the I/O, user processes are not allowed to modify the aiocb for the operation. In fact, any changes to the aiocb during this period cause the aio system to produce undefined results. Also, since the aiocb contains the pointer to the data buffer for the I/O, if either the address of the buffer or the address of the aiocb become invalid during the lifetime of the aio, the behavior of the operation becomes undefined.

The main drawback to this standard is that it takes more effort to write the support code for an aio_write that it does for a normal write. Since aio does not block the requesting process, the user cannot be guaranteed to have any data ready by the time the aio_write returns, therefore complicating the flow of control at user level. The user is, therefore, forced to deal with event notifications or signals as well as extra error handling. The benefit, however, is that this standard allows for better performance. If there is any useful work independent from the I/O, the application is free to do it instead of having to wait for an indefinite amount of time. This improves pipelining in the user application, resulting in higher throughput. The other benefit is that the semantics of the asynchronous I/O prevent the application process from touching the data buffer until the I/O is completed, which allows the kernel to make assumptions about the data contained in the buffers. In particular, this restriction enables the operating system to share data buffers read-only with the user application, which is the main idea on which I base the zero-copy scheme I propose.

The alternative to pure asynchronous I/O calls are synchronous I/O operations on non-blocking file descriptors. These are usually carried out by the application
by marking the descriptors as non-blocking and then sending normal write requests. The difference, however, is that once the kernel decides it cannot write anymore without blocking the user application, it returns immediately to avoid blocking the application. When this happens, the write caller is given status in the amount of data written, and is thus responsible for finishing the I/O when it is again possible. This paradigm suffers from most of the same drawbacks as AIO in the sense that it is more challenging to write code in this fashion.
Chapter 3

Design and Implementation

Consider the typical steps taken to handle a file request on a traditional server. First, the server issues a read request for the file. To satisfy this request, the kernel programs the storage device to read the data and copy it into a buffer in the kernel's address space. To finish serving the read, the kernel then copies the data from its internal buffer into the server application buffer. The server then, in order to satisfy the original file request, immediately writes the data into a network socket. To satisfy this send request the kernel is again forced to copy the data, only this time it does so from the user buffer back into a different kernel buffer. At this point, the kernel is ready to program the network interface to read the data directly from the second buffer and begin transmitting it.

The data copies that occur in the standard data path are overhead that degrades server performance considerably, because copying buffers ties up system resources as explained in Section 2.2. Applications that serve data from disk storage over the network such as database servers and web servers are examples of software that is affected by this type of overhead in the kernel. Typically, high performance servers feature optimizations such as data caching and special implementations of I/O operations on raw disk that focus on minimizing the cost of making the data initially available to the server. Zero-copy network I/O targets such applications where the cost of obtaining the data is lower. Therefore, the relative penalty for copying the data in the send (receive) path is a higher fraction of the total cost.

The idea behind zero-copy network I/O is to avoid the copy of the user buffer into kernel space for outgoing data without compromising the integrity of the information
being transferred. The way I satisfy these two constraints is by adding zero-copy to the AIO transfer routines to exploit the way the standard naturally fits into the problem. I modify the AIO API so that calls to the methods result in the kernel performing zero-copy transfers. As was discussed in previous sections, AIO strictly prohibits the user from modifying the data buffer while the I/O is in progress. This restriction allows the kernel to safely share the data buffer with the user process without having to take the extra precautions like protecting pages copy-on-write. This eliminates the copy in the send path and reduces the total overhead per request. To adapt the zero-copy semantics to the AIO interface and eventually produce the zero-copy and zero-mapping TCP send I took three main steps.

- Modify the implementation of the send routines to make the kernel and the application share the buffers in a zero-copy fashion by mapping the data into the kernel.

- Change the notification system to trigger events when the sending system receives acknowledgement for the data transferred, instead of when it queues the data in the socket for transmission, meaning that the buffer sharing can be terminated.

- Under the assumption that the kernel would not need access to the transmitted data for the duration of the transfer, eliminate the kernel mappings that were generated for the buffer sharing.

Most of the modifications not in the AIO system were localized to the sosend routine, which is at a relatively high level in the send path. This method is effectively the first one called on socket writes once the operating system determines that the target file descriptor corresponds to a network connection. Sosend is responsible among other things for calling the lower level protocol specific functions that do the actual transfers. Some modifications were necessary on the event notification system as well as the network interface driver.
3.1 Zero-copy Mapping TCP Send

In this section, I will revisit the network subsystem for background purposes although this time at a bit of a lower level than it was done previously.

Mbufs - Network Subsystem Revisited

One of the main concepts in the design of the networking subsystem in FreeBSD is the memory buffer (mbuf) used throughout the implementation to hold connection information and user data. Following is a discussion of the mbuf as a cornerstone idea to the networking subsystem, as well as a description of the dataflow for typical network output operations.

![Mbuf Structure](image)

Figure 3.1 : Mbuf Structure

As the main data structure in the networking subsystem, mbufs provide an abstraction that allows the implementation of typical network operations like packet fragmentation and reassembly and header and trailer manipulation, while minimizing the amount of data copied around. The main use of the mbufs is to hold data that travels from the user application to the NIC and vice-versa. Mbufs, however, are also used to carry protocol headers, socket options and source and destination addresses. Figure 3.1 shows the structure of an mbuf. In recent versions of FreeBSD,
the size of the mbuf structure is 256 bytes of which 20 contain control information and 236 contain actual data. The m_next and m_nextpkt are pointers to other mbufs and are used to create lists that represent network packets. The m_data field points to the beginning of the data within the mbuf, while m_len holds the amount of data contained in the buffer. The operating system uses m_type and m_flags to define the mbuf type and other options.

![Diagram of mbuf structure](image)

Figure 3.2: Mbuf Chain Example

Consider a situation in which the kernel wants to buffer 200 bytes of user data into mbufs. To handle this, the kernel copies the 200 bytes of the data in the data portion of the mbuf. It also fills the control and option fields and adds a little extra information to the data portion of the mbuf to mark it as the first buffer. What happens if the kernel wanted to buffer 2100 bytes of data? In this case, since the data to buffer exceeds 236 bytes, which is the maximum amount of data that can be embedded in an mbuf, the kernel uses an mbuf cluster. An mbuf cluster is another type of buffer that provides 2KB of space and much like the mbufs are managed by the kernel. To do the buffering, the kernel uses the m_flags field to mark the mbuf as a buffer pointing to external data, and stores a pointer to the cluster in the data portion of the mbuf. The kernel then allocates a new mbuf and links it to the first
mbuf by using the m_next pointer, creating an mbuf chain. The remaining 100 bytes are then copied to the newly allocated mbuf. Figure 3.2 shows the sample mbuf chain discussed. We can conclude from this that there are 2 types of mbufs: those embedding data and those pointing to external buffers.

In the implementation of the basic sendfile, the sf.bufs take the place of the mbuf clusters, allowing the file pages to be treated just like regular data.

In order to produce the zero-copy mapping send, sosend was modified to adapt the sf.buf mechanism. The modifications mentioned in this section were performed to the sf.buf data structures to adapt them to normal socket writes and to the sosend routine itself. As a result, sosend avoids the data copy in a similar fashion to sendfile by using the sf.bufs to fill the network buffers. However, this happens only when mbuf clusters are necessary because there is enough data to send out. In the case that mbuf clusters are not used and the data gets embedded into the mbuf, the data is copied into the buffer like in a regular send. The reason for this is to establish a threshold for the amount of data so that the zero-copy path is used on large enough chunks to guarantee that the benefits of zero-copy outweigh the costs.

### 3.2 Ack Activated Notification

An added benefit of adapting the sf.buf mechanism to sosend is that since the sf.bufs are associated with the network mbufs, they get destroyed whenever the mbufs do. This means that the sf.bufs are freed either when the send completes because of the sender receiving an acknowledgement or when a transmission error occurs (e.g., broken pipe). Both of these events mark the first time the user can be guaranteed the kernel will not need the data buffers any more to guarantee delivery. Therefore, they constitute the best candidates for triggers to activate the completion events the kernel must send to the user.

In order to send the signal reliably, the system must ensure that it triggers one and only one event. For this, the kernel must activate the event notification from only one
of the sf.bufts while the buffers are being recycled. I achieve this by linking all sf.bufts in a particular transfer to the first sf.buf created for the send. This means that every sf.buf must register and unregister with the reference sf.buf (the first one created), at the time of their allocation and deallocation respectively. Once the reference sf.buf notices that its last child sf.buf unregisters, it activates the event notification for the user and goes through the deallocation process. In the event that the reference sf.buf is no longer needed for its virtual address but the transfer is still in progress, the kernel cannot deallocate it so it marks it dormant and unavailable for the remainder of the write until the notification is sent to the user application.

All the modifications discussed above were performed in the sf.buf data structures and mechanism. However, the NIC driver should provide enough support for the system to work efficiently. In particular, the driver should make sure that the buffers' reference counts are updated appropriately as soon as the acknowledgement for the data reaches the network subsystem. This allows the buffers to be freed as soon as possible. If the driver does not update the reference counts when the data arrives, this creates the possibility that the buffers have to wait for driver timeouts to become available for recycling. These timeouts usually have periods in the order of seconds, imposing an upper bound on the buffer recycling rate, which could lead to considerable performance degradation.

3.3 Zero-mapping Asynchronous TCP Send

The last step in producing the zero-copy and zero-mapping asynchronous send was to take the zero-copy but mapping send and do away with the kernel mappings. As usually happens on a normal send, the kernel maps the buffer data into its address space in case it needs to access it for performing checksums or for other operations. With modern interfaces, however, this is not necessary since the kernel is capable of offloading the checksum computation to the network card. This allows the kernel to merely obtain the physical address of the data buffer, construct the mbuf clusters
with this information and later program the network interface to read the data and transmit. This however, implies some minor changes in the behavior of the network interface driver. Since the driver expects to receive from the mbuf cluster a kernel virtual address for the buffer, it must be modified in order to support zero-mapping I/O. In particular, it must be changed to accept the physical address of the buffer and skip the address translation. Since the driver must be able to distinguish this special case from that of a normal send, the mbuf, which carries the data buffer, must also be modified to carry a special flag that the driver recognizes.

Finally, as one of the objectives was to produce the zero-copy send while minimizing the modifications to the kernel and the applications using it, I added “backward” compatibility for systems using network interfaces that do not support checksum offloading. To achieve this compatibility, the kernel was modified further to identify systems requiring in-kernel checksum computation. Once the kernel reaches the code to compute the checksum, it generates an ephemeral kernel mapping for the buffer, accesses the data, performs the computation, and then immediately removes the mapping.

These ephemeral mappings are merely temporary kernel virtual addresses that the kernel assigns to data in such cases as zeroing and copying pages. Since they are temporary, the kernel has no need to keep them consistent after they are used, and they are therefore less costly than persistent or normal mappings. The other benefit is that the bookkeeping associated with them happens only on the local processor, which is promising for implementations of the zero-copy I/O scheme on SMPs. If this idea translates to a multiprocessor implementation, even in the worst case of not having a checksum offloading interface, the sending system would still prevent TLB shoot-downs and all related overhead.
Chapter 4

Evaluation

This chapter presents all aspects related to the experimental evaluation of the zero-copy and zero-mapping asynchronous TCP send. The chapter is organized as follows. Section 4.1 describes the experiments performed, while section 4.2 presents the workload used for the experiments. Then section 4.3 describes the hardware and software setup used in the experiments, and finally section 4.4 presents the results along with a discussion.

4.1 Methodology

To obtain the results, three different experiments were completed. First, I used netperf to determine maximum TCP throughput between the client and server. Netperf is a network performance benchmark that works by sending streams of data to a specified server. For this experiment, netperf was used from the server to the client since that is the direction that the interesting data travels for HTTP communications. The purpose of this experiment was to determine an upper bound for the throughput of a server that uses the copying TCP send. Additionally, another version of netperf was created by modifying the original to use a read-only zero-copy and zero-mapping scheme, which performs zero-copy send only on data pages that are marked read-only. The purpose of this implementation was to establish an absolute upper bound for the performance improvements obtainable from any zero-copy and zero-mapping scheme. Since the read-only zero-copy implementation has the lowest overhead, this experiment should show the maximum gain that can be obtained by using any zero-copy scheme.
The other two experiments were carried out to determine the performance improvements under load. The web server used in these experiments, was an event-driven server developed in-house based on the open-source mini_httpd web server. One experiment consisted of making 20000 requests to the server for a particular file by using the Apache HTTP server benchmark tool (apachebench or ab). The idea behind this experiment was to remove the effects of the working set size by making sure that all the files requested were in the server’s memory. For this experiment, I varied the number of concurrent connections from 1 to 16 to determine the peak performance of the server. I also used six different file sizes 1, 6.25, 12.5, 25, 50 and 100 KB to understand the effect that this variable had on performance. The 50 KB file was the biggest round size file that could be guaranteed to fit in a single TCP window of 64 KB, thus removing the effect of multiple windows. 100 KB was the minimum size to guarantee that the file transfer required more than a single window, therefore allowing us to analyze the effects on multiple window transfers. 1, 6.25, 12.5 and 25 KB were selected as a progression to analyze the throughput while still removing the effects of multiple TCP windows.

The final experiment consisted of testing the web server under a real-world load by using a trace from a web server and the trace-driven sclient client emulator program [1]. The purpose of this experiment was to study the behavior of all schemes under the effects that a real-world load imposes on the web server. For both load experiments, ab and sclient, I used mapping and zero-mapping versions of both zero-copy schemes to study the effect that mapping has on performance. The read-only zero-copy scheme was again used to establish an upper bound for performance for mapping and zero-mapping cases.

It is important to mention that for all experiments, the server’s CPU was 100% busy. Care was taken to guarantee that the client machine or any other server resource did not become a bottleneck.
4.2 Real-World Workload

The trace test was performed by running a refined version of a trace obtained from the Rice University computer science department website. The trace has a working set of 16000 files, over 1GB of data, and about 250000 requests. This entire working set, however, was too big for the resources of the server machine. For that reason, the trace’s working set was reduced to a size that would surely fit in the server’s 1GB of physical memory. The log was thus truncated by picking the requests made to the 7500 most frequent files and including them in the refined trace. The working set size of the new trace became approximately 180MB with only about 5% less requests. The file size distribution remained largely unchanged. Figure 4.1 shows the cumulative distribution of requests as a function of file size for both the original and the truncated logs. As the figure shows, the distribution is very similar between the two traces as the only differences are in the two ends of the functions. The request order remained completely unaltered.

Since the entire working set fits into physical memory, there were no other dif-
ferences between the trace based and the single file tests besides for the varying file sizes in the workload. In particular, all disk I/Os due to either page or cache faults were avoided in order to minimize the number of factors and make the comparison between the various TCP send schemes clearer.

4.3 Experimental Setup

The test environment consisted of two workstations connected by two 1Gbps links allowing a maximum of 2Gbps of data in either direction. Each machine featured two Intel Pro/1000 MT desktop adapters, which is a network interface card that supports gigabit bandwidth and checksum offloading for outbound and inbound data. The client machine making all the requests had an Intel Xeon 2.4 GHz processor with 2GB of RAM and a 64 bit 66 MHz PCI bus, while the server machine had an Intel PIII 866 MHz processor with 1GB of physical memory and a 64 bit 66 MHz interconnect. Both computers were running FreeBSD 4.6.2-RELEASE as the operating system. The server machine was the only kernel that needed modifications for the implementation and experiments. Since only the server machine had its kernel modified, more resources were given to the client machine to prevent the server from overrunning it and causing missed data or bad results. Additionally, the TCP window size was set to 64KB to maximize the amount of data sent in each window while still maintaining moderate memory constraints.

4.4 Results

In this section I present the results from the experiments along with a detailed analysis for each of them.
4.4.1 Zero-Copy and Zero-Mapping AIO

In this section, we study the performance improvement measured as throughput obtained by using the asynchronous zero-copy and zero-mapping scheme proposed in this thesis. Figure 4.2 presents peak throughputs with respect to concurrent connections for single file and workload tests. The chart compares the throughput of the proposed zero-copy and zero-mapping aio with upper and lower bounds I established. The lower bound corresponds to the performance obtained by using a web server sending data in a normal copying fashion, while the upper bound corresponds to the performance obtained from using a read-only zero-copy and zero-mapping send scheme.

(a) single files

(b) trace-based

Figure 4.2: Zero-Copy and Zero-Mapping Asynchronous Send

The read-only zero-copy and zero-mapping scheme is the upper bound because it consists of a normal socket write with the modification that at the time of buffering, the kernel, instead of blindly copying the data into the mbuf structures, it first checks to see whether each data’s virtual memory page is mapped read-only. If that is the
case then the write assumes it is safe to hold that portion of the user level buffers
and use it in place of the network mbufs, otherwise then it proceeds as a normal
copying network write. The idea behind this implementation is that the user will not
be able to write to the data page without causing a protection fault therefore it is
not necessary for the kernel to protect the data in any way.

As can be seen from the graph, the upper and lower bound schemes converge in
performance for the 1KB file. This is because in this case the upper bound zero-copy
server actually sends the data after copying to the network buffers. In other words, the
upper bound zero-copy send degenerates to a copying send. The reason for this lies
in the design of the user level application and its interaction with the kernel. When
the web server sends a response on the network, it does so via the writev system call,
which takes a vector of buffers to write and it outputs them on the file descriptor, in
this case a socket. When the web server calls writev, however, it gives the operating
system two buffers, one containing the HTTP headers and the other with the actual
response. Since the header buffers are not contained in read-only pages, they cannot
be sent zero-copy under this scheme. Therefore this scheme will send the first page
of data in a copying fashion, implying that files of up to around 4KB in size will be
sent in their entirety by a copying send. Unlike the upper bound zero-copy send,
the aio send does send small files in zero-copy fashion. This is because in the server
application, the HTTP header and body are sent with two separate write requests.

As the chart shows the aio zero-copy server obtains lower performance than the
lower bound for the 1 KB and 6.25 KB, where the performance decrease in terms of
percentage is higher for the smaller file. This can be explained by the extra overhead
that the aio framework and the zero-copy setup impose on the server. In order to
work correctly, the aio zero-copy version must delay the release of the user buffers,
which costs the server extra processing time. Since the only two overhead differences
between the copying send and the proposed aio send are the AIO framework and the
zero-copy setup, this performance degradation can only be attributed to the overhead
they cause. These two factors will be discussed further in a later section as I try to separate the effect that each has on performance.

For files 12.5 KB and bigger, the benefits of sending data zero-copy offset the added overheads, and thus improve overall performance. As the graph shows, the percentage gain of the zero-copy over the copying version increases from 12.5KB up until the 50KB file. The reason for this percentage increase is that as more data is sent, more time is spent on copying the data rather than initiating the send. Therefore, the zero-copy version optimizes a bigger chunk of the work. The percentage increase seems to slow for the 100KB file because in this case the send spans over two TCP windows instead of just one like all the other cases. Having 2 TCP windows increases the TCP handshaking, therefore increasing the overall cost of the send, which decreases the relative cost of the copies.

The maximum performance gains were obtained for the 100 KB file, where the upper bound scheme showed 50% throughput over the lower bound. In this case, the proposed zero-copy and zero-mapping aio showed improvements of 40% as compared to the lower bound, or 6% within the upper bound read-only zero-copy scheme.

### 4.4.2 Zero-Mapping Benefits

In this section, we analyze the performance benefits offered by the zero-mapping component in the zero-copy and zero-mapping aio scheme. For this we compare the throughput performance of two servers: one writing to the network using the zero-copy and zero-mapping aio and the other writing via a zero-copy only aio.

Figure 4.3 highlights the performance improvements obtained from adding zero-mapping to a zero-copy only scheme. As can be seen from the figure, the throughput improvements from using the zero-mapping send varies from 1% to 3% for single file request streams as well as for the workload test in which 2% improvement was observed.

An interesting observation from the data is that the percentage improvement from
using zero-mapping seems to be very stable with respect to the file size and follows no trend. This should go against the intuition that since zero-mapping offers improvements on a per-page basis just like zero-copy does, the percentage improvements should increase as the file size increases too. In other words, as the file size increases, more pages are sent zero-mapping at the same fixed send initialization cost, therefore performance increase would go up as well. I attribute this to the magnitude of the increases.

The performance improvements obtained from zero-mapping can be completely attributed to the elimination of the unnecessary page table and TLB operations. In the zero-copy but mapping send, for each page that is sent zero-copy, two page table operations are performed, namely a page insertion and removal.

### 4.4.3 Overhead and Performance Penalties

Previous sections made mention of the fact that there is extra overhead involved in implementing the proposed zero-copy send within the aio framework. This overhead
is the reason why the zero-copy and zero-mapping aio does not converge to the upper bound performance. The two main causes of overhead in the zero-copy and zero-mapping aio are the aio framework and the extra zero-copy consistency setup. In this section we try to quantify the performance penalty that each type of overhead imposes on the server. For this purpose, I implemented another version of the zero-copy send also in the aio framework. The idea of this extra implementation is to be a midpoint between the read-only zero-copy presented in Section 4.4.1 and the proposed zero-copy asynchronous I/O solution. I produced this zero-copy scheme by removing one of the sources of overhead, namely the zero-copy setup, from the zero-copy aio scheme. Therefore, this read-only zero-copy aio consists of a zero-copy send for read only pages sitting inside the aio framework. The idea behind this read-only zero-copy send is the same as for the read-only zero-copy scheme. Namely, for all pages that need to be sent, the kernel first checks the write permissions, and if the page happens to be mapped read-only, the kernel holds on to the page and uses it as a backing buffer in much the same way as the network mbufs. The only difference between this version and the read-only zero-copy is the aio framework surrounding the prototype read-only zero-copy aio. Therefore, comparing these two schemes should highlight the overhead due to the aio framework.

Since the read-only zero-copy aio deals only with read-only pages, it does not suffer from the extra overhead necessary to make the solution more general. In particular, it does not need to do the extra sf_buf bookkeeping and delayed notification explained in Section 3.2 that is necessary to associate buffers with received acknowledgements to later free them. Since the extra bookkeeping is the only difference between the proposed zero-copy aio and the read-only zero-copy aio, any difference in performance between the two can be attributed to its overhead. In order to measure the performance overhead caused by the zero-copy consistency setup, I compare the performance of read-only aio and the proposed zero-copy aio.
Figure 4.4: AIO Framework Overhead

AIO Overhead

In this section, I show the overhead of using asynchronous I/O over synchronous I/O. Figure 4.4 shows a comparison between the read-only zero-copy and read-only zero-copy aio sends for the mapping case. As the graph shows, the performance penalty varies from 3% to about 10% for single file tests. From intuition, the reader should expect that as the file size increases, the penalty in terms of percentage should decrease. The reason for this is that the overhead is constant per each transfer regardless of the file size, which means that as the file size increases the overhead should represent a smaller portion of the total time. The entire aio overhead is due to things like queuing of requests and freeing system resources.

Zero-Copy Overhead

This section discusses the performance penalties caused by the zero-copy consistency setup. For this purpose, we use the read-only aio write and compare it to the proposed asynchronous write as explained before. Figure 4.5 shows the performance differences for mapping versions of the two schemes. As the figure shows, the zero-copy costs
vary from 1% to 2.5%. In contrast to the aio overhead discussed above, the zero-copy overhead seems to have less of a pattern in terms of percentages. Since most of the zero-copy overhead happens on a per sf_buf basis, thus on a per page basis, one should expect the penalty to be related to the file size, which would make the percentage penalty to be relatively constant in terms of file size.

![Graph showing zero-copy overhead](image)

Figure 4.5: Zero-Copy Overhead

### 4.4.4 Copy-on-Write Performance

In order to measure the performance improvements from copy-on-write (COW) based schemes, I also implemented one such send. To the best of my knowledge, my implementation follows all the guidelines that the best implementations do.

The way this COW-based send works is by the kernel mapping the data buffers into its address space and protecting them to enforce read-only access. The kernel then lazily removes protections as the application attempts to write to the buffers. This implies that 3 different scenarios can develop during a send.

- The page is written to after the receiving side has acknowledged it, in which case, the kernel must incur in a write-fault to remove the protection.
• The page is written to after it has been buffered in the networking subsystem but before it is acknowledged by the receiving side. In this case, the kernel incurs in a cow-fault and must therefore copy the entire page.

• The page is never written to, in which case the kernel must not incur in any extra overhead.

The application chosen for this experiment was netperf, which is a benchmark utility that measures point-to-point bandwidth by streaming data from one end to the other. In order to obtain performance measurements under all the different scenarios described above, I created three different versions of netperf. One that never writes to the data buffers (cow), one that causes write-faults on all data pages (cow-wf) and one that causes cow-faults on all pages (cow-cf). Additionally, the original version of netperf that used copy sends and another that used the upper bound read-only zero-copy and zero-mapping send were used for comparison.

Figure 4.6: Copy-on-Write Performance

Figure 4.6 shows the results from a default length run of netperf under the upper and lower bound schemes as well as the three cow-based schemes. As can be seen from the figure, the cow send under the absence of faults reaches performance close
to that of the upper bound. The difference is only of about 4% and can be attributed to the establishing of the protections. When write-faults are caused on all sent pages, the cow-based send offers 15% lower performance than the upper bound.

One interesting observation from this experiment was that the worst-case performance for cow-based schemes is difficult to reach in LAN environments. As the reader might recall, the worst case happens when the kernel establishes protections, copies and removes protections for all pages. What was found in this experiment was that copying pages happened slower than the receiving of acknowledgements, which reduced the number of total cow-faults. For this reason we see only a 30% drop in performance with respect to the upper bound, instead of a 60% drop like the copying send shows.

4.4.5 IO-Lite Performance

Unfortunately, a comparable IO-Lite implementation was not produced by the time of this writing, hence, no performance measurements could be performed for IO-Lite-based zero-copy schemes. However, one can establish a theoretical performance for such schemes.

As most of the new-API schemes, IO-Lite works by providing a set of special purpose buffers. These buffers are shared between the applications and all subsystems within the kernel. This mode of operation allows the kernel to perform clean zero-copy sends by loaning out the buffer pages to the networking subsystem. Additionally, these schemes can also make use of the zero-mapping idea that I present in this thesis.

There is no reason to believe that IO-Lite-based schemes or any other new-API schemes can outperform the upper bound zero-copy and zero-mapping scheme presented in this thesis. This is because the new-API and the upper bound schemes use the data buffers in very similar fashion. Therefore, any comparisons between my proposed zero-copy and zero-mapping TCP send and the upper bound can also be translated into comparisons between my proposed aio send and IO-Lite based
zero-copy schemes.

4.4.6 Summary

The experiments discussed above can be summarized as follows:

- The practical upper bound for the zero-copy improvements over regular send as measured with netperf is a little over 60%.

- The practical upper bound for the zero-copy improvements over regular send as measured with the web server application is just below 50%.

- The improvements obtained from using the proposed zero-copy aio TCP send over the regular send reach 40% or within 6% of the measured application upper bound.

- Zero-mapping offers improvements of up to 3% in the architecture studied.
Chapter 5

Related Work

Work in the area of zero-copy schemes can be classified in three broad areas:

User Accessible Interface Memory

The best scenario with minimal data transfer overhead is one in which the network interface memory is accessible and pre-mapped into user space. Data never needs to traverse the memory bus until it is accessed by the application.

Unfortunately, this requires complicated hardware support and substantial software changes. For one thing, cache consistency has to be maintained either through software flushing or special hardware arrangements. Obviously, this scheme also requires applications to use special buffer management calls to allocate and use the interface memory. Software compatibility and portability do not exist.

User-Kernel Shared Memory

These schemes define new APIs with shared semantics between the user and kernel address spaces, and use DMA to move data between the shared memory and the network interface. One proposal in this category is fbufs [6]. It uses a per-process buffer pool that is pre-mapped in both address spaces thus eliminating the user-kernel data copy.

One major disadvantage of this approach is application compatibility. All the programs have to be converted to use this alternate set of APIs and programming model. Managing the shared buffer pool requires close cooperation between the application, networking software and the device driver, all allocating memory from the same pool.
Due to the nature of shared memory, an error-prone application may inadvertently modify memory contents previously sent, causing data corruption problems that are hard to debug.

**User-Kernel Page Remapping + COW**

These schemes have the advantage that they are very easy to implement and they keep the existing APIs intact. They usually work by using DMA to transfer data between interface memory and kernel buffers, and remapping buffers. To maintain copy semantics and therefore APIs, they rely on COW to provide data consistency.

These schemes have the problem that all these operations are expensive. Additionally, even though the applications need not be modified to adhere to new APIs, they must be modified to make sure they do not recycle buffers too soon to avoid extra data copies. Fortunately there are upper bounds to the amount of data outstanding on a write; therefore there is also an upper bound for the amount of data that should be copied because of buffer misuse in the worst case.
Chapter 6

Conclusions and Future Work

6.1 Summary of Contributions

Data copying during TCP writes causes significant overhead on network servers. Since the data must cross the CPU, a considerable amount of processing is consumed on unnecessary work. This along with other problems has prompted the creation of zero-copy send solutions. These solutions however, either modify the existing APIs or depend on costly virtual memory operations. My solution is to use an existing API that naturally fits into the problem, allowing the kernel to minimize virtual memory overhead. My main contributions.

- I proposed the addition of a zero-mapping component to existing versions of zero-copy sending schemes. The improved zero-copy schemes would leverage checksum offloading capabilities of modern network interface cards to further improve performance by avoiding costly TLB and page table operations unless the kernel needs to access the data. The zero-mapping scheme lazily maps data into the kernel's virtual address space via ephemeral mappings which are as costly as normal mappings on UP but are considerably cheaper on SMPs since they do not cause TLB shoot-downs.

- I proposed the design and implementation of an asynchronous zero-copy and zero-mapping TCP send. The zero-copy send avoids all data copies without protecting the data copy-on-write. Although the prototype implementation was done on a FreeBSD system, the ideas on which it is based are general enough that can be extended to other UNIX based systems. Furthermore, the TCP
send is based on a standard API, allowing existing applications to benefit in performance by only running on a modified kernel.

- I provided a detailed quantitative evaluation of the benefits of zero-mapping for three different zero-copy schemes as well as the benefits of asynchronous zero-copy and zero-mapping send in the context of a static content web server. The experiments demonstrated performance improvements of up to 40% for the asynchronous zero-copy send over the standard copying send. Similarly improvements of up to 3% in throughput were obtained by using the zero-mapping proposed in this thesis.

6.2 Future Work

My work on zero-copy and zero-mapping I/O can be extended in many different directions. Some of my ideas for future work on this subject follow.

6.2.1 SMP

The results in this work showed that zero-mapping improved performance by up to 3% in the context of a web server on a UP system. The zero-mapping proposed in this thesis works by mapping data lazily, only when necessary, and via ephemeral virtual mappings into the kernel’s address space. These temporary mappings make little if any difference in the case of UP systems, since they are just as costly as normal mappings on the local processor. However, on SMPs they are much cheaper because they only change the TLB of the local processor, therefore avoiding the blocking and synchronization costs of a TLB shoot-down. For this reason, I believe this zero-mapping scheme should offer bigger performance improvements in such scenarios. Future work in this area would involve verifying that the zero-mapping idea and implementation extends gracefully into MP scenarios. In particular one should make sure that no race conditions are created with the introduction of the ephemeral mappings.
6.2.2 TCP Receive

The solutions proposed in this thesis handle the case of sending data without copies or mappings. The situation is more complicated on the receiving end. In particular, in order to avoid copying, the data must land immediately in its final destination. Here again, modern programmable network interfaces provide the solution. If the read is already pending, the processor can pass the address of the read buffer to the interface and tell it to deposit the data immediately in its final location. If not, then the interface can buffer some amount of incoming data. If that buffer overflows, the interface can revert to the conventional delivery method, depositing the data in kernel buffers, from where the kernel then copies the data to its final destination.

6.2.3 TLB Avoidance for Disk I/O

Although somewhat different from network I/O, disk I/O turns out to have some striking similarities. Non-offloading network cards are analogous to programmed I/O in the sense that they force the CPU to access the data thus prohibiting zero-mapping. However, checksum offloading cards also have their counterparts in the form of DMA controllers, in the sense that they allow the data to be transferred completely independently of the CPU.

In the case of raw disk I/O, it is clearer how to tackle the problem of TLB operation avoidance. Namely, the DMA assisted transfers are allowed to occur by training the kernel to carry physical addresses instead of the virtual addresses and lazily map when necessary. In the case of filesystem I/O, however, it is more complicated because data alignment issues between user space, the buffer cache and the kernel may arise and force page remappings and even full data copies.
Bibliography


