sendmsg copy avoidance with MSG_ZEROCOPY

Willem de Bruijn
willemb@google.com

Eric Dumazet
edumazet@google.com

Abstract

Copy avoidance can save many cycles for large production workloads such as storage servers. Linux offers various copy avoidance mechanisms, such as sendfile, vmsplice and virtio zerocopy. We review these existing interfaces and introduce a copy avoidance interface for generic socket communication. Processes converted to this MSG_ZEROCOPY interface see a cycle reduction ranging from 92% of process cycles (39% systemwide) for a netperf micro- benchmark to 5-8% for lightly modified production workloads in machine learning and content delivery.

The feature combines a send() flag with a completion notification channel over the socket error queue. Aside from introducing the interface, we focus on technical constraints. Primarily, supporting complex protocols like TCP, where segmentation, retransmission and reordering introduce a non-trivial relationship between user buffers and packets on the wire. To handle these, completion notification requires careful reference counting across the transmit stack. Other concerns are bounding notification latency and working set size, avoiding TOCTTOU attacks with shared read-write page mappings and amortizing the cost of notification processing over the socket error queue.

Introduction

Applications that handle large data streams, such as storage and content delivery, can end up spending most time copying data around. A simple netperf stream can spend 79% of its cycles in copy_user_generic_string, as shown in the experiments section.

Shared memory is a popular alternative. Userspace device drivers move all processing into the application address space, but do so at the cost of losing operating system safety, abstraction and resource multiplexing. Copy avoidance replaces copies in existing datapaths with pinned pages. The Linux kernel exposes a number of copy avoidance mechanisms across subsystems.

The sendfile system call allows processes to move data from the page cache in the kernel to another file, pipe or socket without having to copy the data first into the process and then back to the kernel. For network transmission, the page cache fragments are linked directly into the skbuff's frags[] array. The skbuff takes a reference on the page to ensure that it is not removed from the page cache. It does not restrict the page contents from being updated. The skbuff is marked with flag SKBTX_SHARED_FRAG. Where payload access is necessary, such as on checksum generation, this flag is checked and if found set, shared pages are replaced with private copies.

The splice [3] system call extends copy free transfer of data from page cache entries to more generic kernel buffers represented as unix pipes. A process can move data between pipes, files and sockets by calling splice with the two file descriptors and the SPLICE_F_MOVE option. vmsplice adds the ability of splicing user data into a pipe in the kernel without copying. User pages are pinned while contents is in use in the kernel. Subsequent splicing of data from this pipe to another pipe, socket or file by default causes a copy. But, if vmsplice is called with flag SPLICE_F_GIFT, then the kernel will forward the page fragments without copying.

A page inserted with the gift option must not be modified while it is in use in the kernel. Indeed, the vmsplice manual page states that a buffer passed with the gift flag may never be reused by the process. A page that is passed to the kernel and moved to a socket for transmission will see any changes made to the page page inbetween vmsplice and copy to the device local buffer reflected on the wire.

The vmsplice interface does not notify the caller when it is safe to modify page contents. Applications using this mechanism for networking often do reuse memory, relying on other methods to detect whether data has been sent, such as polling the SIOCOUTQ ioctl. Such methods are imprecise and error-prone. An empty transmit queue does not indicate that data has left the machine. Packets may be waiting in the device transmit queue, for instance, or a clone can be mirrored to a packet socket if a tcpdump instance is running.

Another copy avoidance path within the kernel itself shows how completion notifications can be implemented. virtio zerocopy [4] avoids the copy on transmission from guest virtual memory to packet buffers in the host transmit stack. The vhost-net driver emulates a guest network device in the host kernel. It has access to shared guest memory and the virtio transmit descriptor ring. The driver moves data from guest buffers into the standard host transmit path by sending over a kernel tuntap socket. To call the tuntap send path it converts descriptors into a msghdr with data iov io vector array. Unlike user processes, the driver then calls sendmsg as a function call from with the host kernel to pass data to the tun driver in the same address space. In copy avoidance mode, this func-

```
ret = send(fd, buf, sizeof(buf), MSG_ZEROCOPY);
if (ret != sizeof(buf))
  error(1, errno, "send");

pfd.fd = fd;
pfd.events = 0;
if (poll(&pfd, 1, -1) != 1 ||
    pfd.revents & POLLERR == 0)
  error(1, errno, "poll");

ret = recvmsg(fd, &msg, MSG_ERRQUEUE);
if (ret == -1)
  error(1, errno, "recvmsg");

read_notification(msg);
```

Figure 1: Using the MSG_ZEROCOPY interface

tion has been modified to replace copying with building an skbuff that directly references the pointers in the io vector. The reference count on each page linked into the skbuff array is incremented, as destruction of the skbuff fragment array will later decrement this count. To notify the vhost driver of completion of the zerocopy operation, the tun driver embeds a pointer to a callback function into the skbuff.

Socket Interface

MSG_ZEROCOPY introduces a copy avoidance interface for standard socket communication. The feature is implemented for communication with remote peers over TCP, UDP and RAW and packet sockets. The implementation builds on the virtio copy avoidance infrastructure.

The change to send is minimal. Passing the new flag MSG_ZEROCOPY requests direct transmission of shared user pages. In this case the kernel pins the user pages and creates skbuff fragments directly from these pages. Similar to SPLICE_F_GIFT, contents must not be modified once the send call commences, or changes may be reflected in the packets on the wire.

Completion Notifications

The interface includes a completion notification interface to let the send process know when it is safe to reuse memory. The kernel already has an interface for notifying a process asynchronously of socket events. The transmit timestamp interface queues a timestamp for a packet (or range of TCP bytestream) on the socket error queue. A hardware tx timestamp requested with SOF_TIMESTAMPING_TX_HARDWARE acts as a transmit completion notification.

The transmit timestamp is queued onto the error queue with a copy of the original packet to be able to associate timestamps with the original send request. Copying all this data is expensive, however, and not always unambiguous. Timestamp option SOF_TIMESTAMPING_OPT_ID queues a per-socket identifier alongside the data and timestamp. Each send call requesting a timestamping increases a per-socket counter. The current value of the counter is queued in the skbuff while in transit and returned alongside the timestamp. The value identifier unambiguously the original call, so sending packet payload back up to userspace is no longer needed. Option SOF_TIMESTAMPING_OPT_TSONLY queues only the identifier and timestamp.

Zerocopy notifications are not transmit notifications. A packet can be transmitted while a clone of the same data is queued elsewhere in the network stack. The transmit timestamp interface cannot be used as is, therefore. But it is analogous to the TSONLY mode.

A zerocopy completion notification message is a simple scalar value that identifies the system call whose data has been sent. Each socket maintains an internal 32-bit counter. Each send call with flag MSG_ZEROCOPY that successfully sends data increments the counter. The counter is not incremented on failure or if called with length zero.

Notifications are queued asynchronously on the socket error queue. When the last reference to the packet fragment array is released, a callback function is triggered, similar to the virtio case. In this case, the callback action is to queue the scalar value associated with the packet onto the error queue.

Figure 1 demonstrates the API. In the simplest case, each send syscall is followed by a poll and recvmsg on the error queue. Reading from the error queue is always a non-blocking operation. The poll call will block until an error is outstanding and will set POLLERR in its output flags. That flag does not have to be set in its events field: errors are signaled unconditionally.

The example is for demonstration purpose only. In practice, it is much more efficient to not wait for notifications, but read without blocking every couple of send calls. Notifications can be processed out of order with other operations on the socket. A socket that has an error queued would normally block other operations until the error is read. But zerocopy notifications have a zero error code, and do not block send and other calls.

Multiple outstanding packets can be read at once using the recvmmsg call. This is often not needed. The notification interface can coalesce multiple scalar values onto a single range-based message.

Figure 2 demonstrates how to parse the control message. A notification follows the standard format of an error on the error queue. Information about the error can be read alongside the packet contents as a msg_control control message. For zerocopy notifications, the packet in msg_data itself is empty. The level and type fields in the control data are protocol family specific and differ between IPv4, IPv6 and packet sockets. If of the family-specific error type, such as IP_RECVERR, then embedded is a sock_extended_err structure. For zerocopy notifications, error origin is SO_EE_ORIGIN_ZEROCOPY. ee_errno is zero, as explained, to avoid blocking other system calls on this socket. ee_data and ee_info fields each hold a completion value. The messages returns not a single completion event, but range [ee_info,ee_data]. The example code in the figure ignores the lower end for simplicity. Other fields in the struct must be treated as undefined.

Notification processing is one reason why copy avoidance is not always more efficient than copying, especially not for small packets. The experiments section demonstrates this

```
uint32_t read_notification(struct msghdr *msg)
{
  struct sock_extended_err *serr;
  struct cmsghdr *cm;

  cm = CMSG_FIRSTHDR(msg);
  if (cm->cmsg_level != SOL_IP &&
      cm->cmsg_type != IP_RECVERR)
    error(1, 0, "cmsg");

  serr = (void *) CMSG_DATA(cm);
  if (serr->ee_errno != 0 ||
      serr->ee_origin != SO_EE_ORIGIN_ZEROCOPY)
    error(1, 0, "serr");

  return serr->ee_data;
}
```

Figure 2: Reading a completion notification

with a micro-benchmark at various send sizes. Many real workloads will have a mixture of both large and small buffers. A common example is combining small application protocol headers with larger payloads from a user cache. The interface supports mixing send calls that have the `MSG_ZEROCOPY` flag set with those without for this reason. The completion notification counter only increments on calls with the flag set.

Revert to Copy

Copy avoidance is not always possible. Devices that do not support scatter-gather I/O cannot not send packets consisting of kernel protocol headers and user payload. Other limitations are not immediately apparent at send time. A packet may need to be converted to having private pages deep in the stack, for instance to compute a checksum. In all these cases, the kernel returns a completion notification when it releases hold on the shared pages. It has to, because the process has no other way of learning when it is safe to reuse memory. In these cases, the notification may arrive before the (copied) data is fully transmitted. A zerocopy completion notification is not a transmit completion notification, therefore.

Implementation

`MSG_ZEROCOPY` builds on the zerocopy `virtio` infrastructure. On `send`, user pages are pinned to ensure that they exist for the duration of transmission. They are linked into the fragment array of the `skbuff` and the `skbuff` transmit flag `SKB_DEV_ZEROCOPY` is set. This flag and the array are, strictly speaking, stored in the `skb_shared_info` segment and thus shared with all clones of a packet. On destruction of the shared data, the kernel tests the zerocopy flag and, if set, calls a callback function hung off of field `skb_shinfo(skb)->destructor_arg`. That pointer points to a struct of type `ubuf_info`. That structure stores, besides the callback function pointer, stores the argument to return (the notification range) and state needed at runtime, such as a reference counter.

Reference counting

The transmit stack can clone packets. A listening tcpdump process will cause a clone to be queued for reception, for instance. This particular case will trigger a copy that converts the packet to one with private fragments for reasons discussed in the security section. But not all clones can be converted to a copy. TCP keeps buffers queued for retransmission until they are acknowledged. When the TCP stack transmits a packet, this is a clone of the one kept on the queue. Triggering a deep copy on every clone precludes supporting TCP, therefore. Accounting for clones is straightforward. All clones share the same `skb_shinfo(skb)` and this shared region is reference counted and freed only when the last clone is released. In normal operation for TCP this will be on reception of the ACK.

Note that it is not safe to simply wait for an incoming ACK to test whether shared memory is no longer in use in the kernel. Such a strategy is no safer than the SIOCOUTQ trick. ACK packets can be spoofed by a malicious peer or man-in-the-middle. Even without malicious intent, an ACK may arrive for a packet while it is in the process of being retransmitted. In that case a clone exists in the transmit stack, while the one on the transmit queue is freed.

The packet in the retransmit queue and the packet being transmitted are not necessarily identical. Generic segmentation offload (GSO), for instance, breaks a large packet up into a train of smaller MTU-sized packets. All these packets must be associated with the original system call: its scalar value must only be queued on the error queue when all these packets have been sent and freed. For this reason, the `ubuf_info` maintains a reference count independent from the `skb_shared_info` one.

Bytestream Packetization of the TCP bytestream further complicates this relationship of packets on the wire to buffers passed to `send`. GSO maintains a 1:M relationship with a large packet, but not necessarily with any buffer originally passed to `send`.

A single system call can send many Megabytes and thus generate many packets. Exact packetization depends on state such as path MTU, so cannot be predicted by the calling process. To maintain the simple syscall counter notification interface, a notification may only be queued once all packets have been sent and acknowledged, so all point to the same `ubuf_info`.

Packetization can span across system calls. That is, a single `skbuff` can embed data from two or more consecutive `send` calls. If a packet is already queued for transmission and not too large, TCP will try to append new data to that in the subsequent `send` call. In that case, a single packet will point to data from multiple system calls, so must queue as many notification values. To avoid having an M:N relationship between `skbuff` and `ubuf_info`, and a list of pointers for each `skbuff`, struct `ubuf_info` stores a range instead of a scalar.

UDP is conceptually much simpler than TCP, but the same range based notification is used here, too, for corking.

Completion Notification

Notifications must be delivered under all circumstances, which places additional constraints on the implementation.

Allocation During memory pressure, the allocation of a notification skb can fail. The notification skb must therefore be allocated immediately in the original `send` system call and that call must fail if the allocation fails.

The zerocopy lifecycle uses two objects. The `ubuf_info` to account pages while shared and the notification `skbuff` to queue a value once they are released. To save one allocation at send time, the two share the same storage. The smaller struct fits in `skb->cb[]`, avoiding the need for a separate kmalloc.

The structures are allocated from socket optmem to avoid interfering with accounting of normal transmit and receive paths, and feedback loops that depends on these. Transmission, in particular, is sensitive to allocation. TCP small queues limits the amount of data in the stack based on outstanding send buffers.

Socket Lookup When it is time to queue a notification, in general this can be looked up from the socket pointer in the `skbuff` that is being freed. But if that `skbuff` was orphaned, the socket reference has already been released. For this reason `ubuf_info` must maintain a private reference on the socket. If it finds that it holds the last reference on the socket at notification time, the notification is dropped as it will never be read.

Coalescing Notification processing is a potentially expensive operation. Coalescing can significantly reduce this cost. A process can delay notification processing to read multiple notifications in batch with `recvmmsg` and individual notifications may contain a range.

The kernel also coalesces consecutive notifications. When a notification is about to be enqueued, it checks whether the new range extends the range of the notification at the tail of the notification queue. If so, it drops the new notification packet and instead increases the range upper value (field `ee_data`) of the outstanding notification. For protocols that acknowledge data in-order, like TCP, each notification can be squashed into the previous so that no more than one notification is outstanding at any one point while the socket is connected.

Ordered delivery is not ensured for unreliable protocols such as UDP, nor for TCP on socket teardown. When that happens, the `skbuffs` on the transmit queue are all purged and any unsent packets trigger an immediate notification. Packets that have a clone in the transmit stack, however, will wait until that clone is freed, causing out-of-order completion arrival.

Security

It must not be possible for a malicious process to circumvent access control by changing the data between policy enforcement and use, a classic TOCTTOU attack. For network transmission, a packet must not be able to evade packet payload filters.

Most operations are on packet headers only, and headers are never stored in shared memory. The Linux network stack expects headers to lie in the linear segment (`skb->head`). `MSG_ZEROCOPY` only places shared pages in the frags array. For most protocols, the headers are generated by the kernel. Even where not, as in packet sockets, the initial bytes up to `MAX_HEADER` or packet size, whichever is smaller, are copied into the private linear segment. Small packets, as a result, are completely copied, even when the copy avoidance flag is set. This is another example where a notification must be queued even when zerocopy is inactive, and a reason for using the feature only on large send requests.

Most operations in the transmit stack only touch headers, but there are a few notable exceptions that must be handled safely. At entry to all these codepaths, the stack can convert a `skbuff` with shared fragments to one with a private copy by calling `skb_orphan_frags`. This tests whether the packet has shared pages and if so converts them to private copies by calling `skb_copy_ubufs`.

Checksum Generation Checksum generation is the canonical example. Modifying payload after checksum generation does not affect the integrity of the system. A process can only hamper its own communication. But copy avoidance is not an effective performance optimization if payload has to be read for checksum generation. When the two operations happen together, the data is warm in the cache and the copy operation is essentially free: this is the basis of the checksum-and-copy optimization in Linux system calls. Copy avoidance is disabled, therefore, if the checksum operation is not offloaded to hardware.

With hardware checksum generation, the device copies host memory to local buffers before computing the checksum. The operation is not atomic, but a host process cannot modify bytes after they have been checksummed. The copy from host to device memory protects against all such vulnerabilities to concurrent memory access that a device may have. Device implementations may vary in principle, but this is a well tested solution. The `sendfile` interface has shared pages from the page cache with devices for a long time.

Encryption Cryptographic operations are increasingly common in the datapath. The same hardware offload argument holds, but the potential vulnerability is more subtle. Block ciphers convert the raw symmetric key to a different key on each block. If plaintext is modified between retransmission, two different plaintext blocks will be encoded with the same key, leaking some information. For this reason, and because encryption is again a data touching operation which obviates the benefit of copy avoidance, a deep copy must be made if zerocopy packets enter the IPSec transmit path.

Deep Packet Inspection The administrator can insert increasingly powerful programmable filters in the transmit path. The `bpf` and `u32` programs can be called from iptables and the packet scheduler to make policy decisions on any bytes in the payload. If payload can be changed after passing these hooks, a malicious process could bypass security policy.

Reliability

Resource exhaustion is another concern when pinning memory pages. To protect machine integrity, unprivileged users must not be able to pin an unlimited number of pages. The

implementation bounds the number of pages for unprivileged users to an administrator defined per-user limit on locked pages `ulimit -l`.

Processes have a number of options to bound their working set. The simplest is to stop passing `MSG_ZEROCOPY` once a working set size limit is reached. But this causes a performance discontinuity that may be unacceptable for reliable production services.

A process can also release its own reference on a page, depending on how it allocated the page. An `munmap` call reduces process working set size, if not total system memory utilization. But in general, shared memory transmission will have lower system memory pressure than copying. A process can replace a shared page with a private copy by copying it into another page and remapping that to the same virtual address using `mremap`. This operation is not atomic, so care must be taken in multi-threaded processes. But unlike `munmap` alone, it can safely release hold on shared pages administered by a userspace allocator like `malloc`.

A particular concern in long running production services is avoiding unreasonable resource use at the tail by slow connections. A `close` call releases all packets on the transmit queue. It does not guarantee that all data has left the host, however. A clone may be queued in the traffic shaping layer, for instance. As a `close` call closes the entire socket including its error queue, the process has lost the ability to receive completion notifications. Instead, a tcp socket can be purged with `connect` AF_UNSPEC. This purges the transmit queue without closing the error queue. While not advisable, such a disconnected socket can be reused with a subsequent `connect`. The zerocopy counter is not reset in that case.

Notification Latency

Working set size is a function of both number of pages and time that pages are outstanding. In the extreme case, if a packet is sent to a local socket that is never read, latency is unbounded. To avoid this case, shared pages are converted to a private copy on all paths that loop back to local sockets. On reaching the receive path in `__netif_receive_skb_core`, `skb_orphan_frags` triggers an `skb_copy_ubufs`. The same happens for packet sockets and the tun transmit queue.

Perceived working set size can be larger than true working set size if many fragments share a single `ubuf_info`. In one experiment, 75 system calls ended up coalescing onto the same `ubuf_info`. Notification latency for completion of the first buffer was unreasonably long. Even though the kernel had unpinned the memory, the process was unaware and thus unable to reuse this memory. To avoid such degenerate cases, the structure now counts the number of bytes that it represents. If this total exceeds a configured limit the append will fail. The consequence of such a hard break is that data cannot be appended to an existing packet where it would have been if data were copied, so packetization on the wire differs. The current limit was experimentally chosen to be 512KB, though more data is needed to measure the rate of packetization changes with limit and whether the limit can be increased to reduce that effect.

	process cycles		
size	copy	zerocopy	%
4K	27,609	11,217	41
16K	21,370	3,823	18
64K	20,557	2,312	11
256K	21,110	2,134	10
1M	20,987	1,610	8
	system cycles		
size	copy	zerocopy	%
4K	49,217	39,175	79
16K	43,540	29,213	67
64K	42,189	26,910	64
256K	43,006	27,104	63
1M	42,759	25,931	61

Figure 3: Netperf throughput as a function of send buffer size

Experiments

A microbenchmark indicates how copy avoidance efficiency depends on buffer size. Figure 3 shows cycles reported by perf for a netperf process sending a single 10 Gbps TCP_STREAM at increasingly large send sizes. Reported is the median of at least 3 runs. Netperf is pinned to cpu 2, network interrupts to cpu 3. RPS and RFS are disabled and the kernel is booted with `idle=halt`.

The first column shows the buffer size. The next three columns show Mcycles spent in the netperf process context without and with `MSG_ZEROCOPY` and their ratio. The second figure shows the same information, but now for systemwide cycles on the two cpus that run the process and interrupt handler (`perf record -a -C A,B`).

`MSG_ZEROCOPY` reduces cycle cost, with savings increasing with buffer size. At 4KB, the call takes 41% of the copy-based equivalent. At 1MB, it takes only 8%. For network transmission, process cycles do not fully capture the overhead. Systemwide, improvements are still significant, but smaller, at 79% to 61% of the equivalent copy-based workload.

Perf record indicates the cause for these performance differences. Figure 4 drills down into the 1M send buffer size case. The copy based path spends 79% of its process cycles copying data to the kernel. The variant with `MSG_ZEROCOPY` has a lower absolute event count. The copy function is notably absent. The top items in its place are functions for page pinning and for inserting user frags into the `skbuff`.

The interface is currently being evaluated in production workloads in machine learning and content delivery. Those mature production systems perform many actions besides copying, so savings are expected to be smaller. Even though these production codebases are large and multi-layered, conversion to this interface proved not too complex. Both applications already hold data in a user queue until it is sent. Extending this period until the send notification is read was the most complex change.

The machine learning application is a mixed workload on a distributed version of the open source Tensorflow[2] ML framework. This early distributed version ran over a pro-

```
copy:
Samples: 42K of event 'cycles',
Event count (approx.): 21258597313
 79.41%   33884  netperf  [k] copy_user_generic_string
  3.27%    1396  netperf  [k] tcp_sendmsg
  1.66%     694  netperf  [k] get_page_from_freelist
  0.79%     325  netperf  [k] tcp_ack
  0.43%     188  netperf  [k] __alloc_skb

zerocopy:
Samples: 1K of event 'cycles',
Event count (approx.): 1439509124
 30.36%     584  netperf  [k] gup_pte_range
 14.63%     284  netperf  [k] __zerocopy_sg_from_iter
  8.03%     159  netperf  [k] skb_zerocopy_add_frags_iter
  4.84%      96  netperf  [k] __alloc_skb
  3.10%      60  netperf  [k] kmem_cache_alloc_node
```

Figure 4: Perf report for netperf sending 1MB at a time

prietary RPC protocol. Overall reduction in wallclock time depends greatly on workload. As expected, in particular on buffer size. The `BM_RPC` benchmark regresses by 15% with 2B RPCs, improves by 23% on 98KB RPCs. The wallclock time of a more typical real mixed workload was 7.5% shorter with `MSG_ZEROCOPY`. Tensorflow now supports the opensource gRPC [1] protocol for distributed operation. We are working on porting `MSG_ZEROCOPY` to that.

A proprietary content delivery system on top of standard HTTPS saw a 5% improvement in peak QPS. A generic remote block device application, on the other hand, saw less than 2%. This is unsurprising, once we factor in that that process performs a hash function for integrity protection on all data prior to send.

Summary

Copy avoidance replaces copying with shared memory. A robust implementation relies on page pinning or even page flipping, trading per byte overhead with per page overhead. It is not a panacea, but when applied sensibly, can significantly reduce cycle cost of large data serving applications.

The `MSG_ZEROCOPY` interface extends existing copy avoidance in Linux to common sockets. It is implemented for TCP, UDP, RAW and packet sockets for communication with remote peers. It combines a simple system call flag with a completion notification channel over the socket error queue. The implementation extends existing copy avoidance for virtual machine networking with support for cloning and complex bytestreams. Microbenchmarks show savings up to 92% of process cycles and 39% of overall system cycles for a simple netperf TCP benchmark. Early experiments with real workloads are in the 5-8% savings for lightly modified machine learning and content delivery applications.

Thanks Thanks to Soheil Hassas Yeganeh and Grzegorz Calkowski for their help in converting complex production services to `MSG_ZEROCOPY`.

References

[1] grpc.io. Sourcecode at http://github.com/grpc/grpc.

[2] Martín Abadi, e. a. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[3] Torvalds, L. Explaining splice() and tee(). LKML email thread, April 19th, 2006 (Re: Linux 2.6.17-rc2). Retrieved from http://lkml.iu.edu/hypermail/linux/kernel/0604.2/0779.html.

[4] Tsirkin, M. S. tun zerocopy support. netdev email thread, July 20th, 2012. Retrieved from https://lwn.net/Articles/507716/.