Zero-copy networking

By **Jonathan Corbet**
July 3, 2017

In many performance-oriented settings, the number of times that data is copied puts an upper limit on how fast things can go. As a result, zero-copy algorithms have long been of interest, even though the benefits achieved in practice tend to be disappointing. Networking is often performance-sensitive and is definitely dominated by the copying of data, so an interest in zero-copy algorithms in networking comes naturally. A set of patches under review makes that capability available, in some settings at least.

When a process transmits a buffer of data, the kernel must format that data into a packet with all of the necessary headers and checksums. Once upon a time, this formatting required copying the data into a single kernel-space buffer. Network hardware has long since gained the ability to do scatter/gather I/O and, with techniques like TCP segmentation offloading, the ability to generate packets from a buffer of data. So support for zero-copy operations has been available at the hardware level for some time.

On the software side, the contents of a file can be transmitted without copying them through user space using the `sendfile()` system call. That works well when transmitting static data that is in the page cache, but it cannot be used to transmit data that does not come directly from a file. If, as is often the case, the data to be transmitted is the result of some sort of computation — the application of a template in a content-management system, for example — `sendfile()` cannot be used, and zero-copy operation is not available.

The `MSG_ZEROCOPY` patch set from Willem de Bruijn is an attempt to make zero-copy transmission available in such settings. Making use of it will, naturally, require some changes in user space, though.

Requesting zero-copy operation is a two-step process. Once a socket has been established, the process must call `setsockopt()` to set the new `SOCK_ZEROCOPY` option. Then a zero-copy transmission can be made with a call like:

```
status = send(socket, buffer, length, MSG_ZEROCOPY);
```

One might wonder why the `SOCK_ZEROCOPY` step is required. It comes down to a classic API

mistake: the send() system call doesn't check for unknown flag values. The two-step ritual is thus needed to avoid breaking any programs that might have been accidentally setting MSG_ZEROCOPY for years and getting away with it.

If all goes well, a transmission with MSG_ZEROCOPY will lock the given buffer into memory and start the transmission process. Transmission will almost certainly not be complete by the time that send() returns, so the process must take care to not touch the data in the buffer while the operation is in progress. That immediately raises a question: how does the process know when the data has been sent and the buffer can be used again? The answer is that the zero-copy mechanism will place a notification message in the error queue associated with the socket. That notification can be read with something like:

```
status = recvmsg(socket, &message, MSG_ERRORQUEUE);
```

The socket can be polled for an error status, of course. When an "error" packet originating from SO_EE_ORIGIN_ZEROCOPY shows up, it can be examined to determine the status of the operation, including whether the transmission succeeded and whether it was able to run in the zero-copy mode. These status packets contain a sequence number that can be used to associate them with the operation they refer to; the fifth zero-copy send() call will generate a status packet with a sequence number of five. These status packets can be coalesced in the kernel, so a single packet can report on the status of multiple operations.

The mechanism is designed to allow traditional and zero-copy operations to be freely intermixed. The overhead associated with setting up a zero-copy transmission (locking pages into memory and such) is significant, so it makes little sense to do it for small transmissions where there is little data to copy in the first place. Indeed, the kernel might decide to use copying for a small operation even if MSG_ZEROCOPY is requested but, in that case, it must still go to the extra effort of generating the status packet. So the developers of truly performance-oriented programs will want to take care to only request zero-copy behavior for large buffers; just where the cutoff should be is not entirely clear, though.

Sometimes, zero-copy operation is not possible regardless of the buffer size. For example, if the network interface cannot generate checksums, the kernel will have to perform a pass over the data to do that calculation itself; at that point, copying the data as well is nearly free. Anytime that the kernel must transform the data — when IPSec is being used to encrypt the data, for example — it cannot do zero-copy transmission. But, for most straightforward transmission cases, zero-copy operation should be possible.

Readers might be wondering why the patch does not support zero-copy reception; while the patch set itself does not address this question, it is possible to make an educated guess. Reading is inherently harder because it is not generally known where a packet is headed when the network interface receives it. In particular, the interface itself, which must place the packet somewhere, is probably not in a position to know that a specific buffer should be used. So incoming packets end up in a pile and the kernel sorts them out afterward. Fancier interfaces have a fair amount of programmability, to the point that zero-copy reception is not entirely infeasible, but it remains a more complex problem. For many common use

cases (web servers, for example), transmission is the more important problem anyway.

As was noted in the introduction, the benefits from zero-copy operation are often less than one might hope. Copying is expensive, but the setup required to avoid a copy operation also has its costs. In this case, the author claims that a simple benchmark (`netperf` blasting out data) runs 39% faster, while a more realistic production workload sees a 5-8% improvement. So the benefit for real-world systems is not huge, but it may well be enough to be worth going for on highly-loaded systems that transmit a lot of data.

The patch set is in its fourth revision as of this writing, and the rate of change has slowed considerably. There do not appear to be any fundamental objections to its inclusion at this point. For those wanting more details, this paper [PDF] by De Bruijn is worth a read.

Index entries for this article
Kernel Networking/Performance

Log in to post comments

[–] Zero-copy networking

Posted Jul 4, 2017 0:35 UTC (Tue) by **mikemol** (guest, #83507) [Link] (3 responses)

> That works well when transmitting static data that is in the page cache, but it cannot be used to transmit data that does not come directly from a file. If, as is often the case, the data to be transmitted is the result of some sort of computation — the application of a template in a content-management system, for example — sendfile() cannot be used, and zero-copy operation is not available.

Is there no mechanism which can be used to wrap a chunk of memory in a file descriptor? I would think that an incredibly useful and powerful abstraction. If I can wrap a file with a block of memory via mmap, why couldn't I do the reverse?

That's not to say this work doesn't have value, but still...

Reply to this comment

[–] Zero-copy networking

Posted Jul 4, 2017 3:47 UTC (Tue) by **lkundrak** (subscriber, #43452) [Link]

There is memfd_create(2).

Reply to this comment

[–] Zero-copy networking

Posted Jul 4, 2017 7:15 UTC (Tue) by **liam** (guest, #84133) [Link]

Mmmmmm....memfd?

Reply to this comment

[–] fd <-> memory

Posted Jul 6, 2017 22:16 UTC (Thu) by **Jandar** (subscriber, #85683) [Link]

You could open /dev/zero and mmap it MAP_PRIVATE. This wouldn't give you a fd corresponding to an arbitrary memory region but you could setup this memory in advance to create the content into it. Zero-copy with sendfile should be possible with this.

Reply to this comment

[–] Zero-copy networking

Posted Jul 4, 2017 0:38 UTC (Tue) by **mikemol** (guest, #83507) [Link] (4 responses)

Also, wouldn't zero-copy reception effectively be RDMA?

Reply to this comment

[–] Zero-copy networking

Posted Jul 4, 2017 8:00 UTC (Tue) by **cladisch** (★ **supporter** ★, #50193) [Link] (3 responses)

With remote DMA, the sender of the packet puts the memory address of the receiver's buffer into the packet, and the receiving interface just writes it there. This requires that the receiver has told the sender in advance what a valid buffer address would be (so this cannot work with most existing protocols), and this is, of course, completely unsafe if you cannot trust the sender.

In theory, it would be possible to configure the interface to put packets from a specific source address and port and to a specific destination address and port into a specific buffer. The article mentions that such fancy interfaces might exist.

Reply to this comment

[–] Zero-copy networking

Posted Jul 4, 2017 8:36 UTC (Tue) by **Cyberax** (★ **supporter** ★, #52523) [Link] (2 responses)

That's pretty much how userspace network stacks work these days. To replicate it, a client will have to set up some kind of a lockless ring buffer and a way for the kernel to wake up the the client. But the API won't look like sockets anymore.

Reply to this comment

[–] Zero-copy networking

Posted Jul 6, 2017 1:10 UTC (Thu) by **vomlehn** (guest, #45588) [Link] (1 responses)

It's still possible to put a sockets API on such a ring buffer based interface but any approach I know of will then require for receive...a copy out of the ring buffer. I think that, ultimately, we need recognize that we need a new set of APIs for zero-copy operations. You use those if you want maximum performance, otherwise you use the familiar sockets interface, which is layered on the zero copy APIs.

By "new", I just mean new for Linux. There are candidate APIs out there that should be considered. The HPC guys are kind of nuts about shipping data, so they have at least one possibility, though I haven't used them. I've heard comments implying that their environment doesn't mandate the level of security a more general Linux deployment might need but this is, again, hearsay.

Reply to this comment

[–] Zero-copy networking

Posted Apr 25, 2018 14:48 UTC (Wed) by **mikemol** (guest, #83507) [Link]

I think zero-copy should be possible with a sockets API.

You still use the ring buffer, but you only drain the buffer when there are read() calls on the socket. Send ACKs only for the packets whose data have been consumed by read() calls.

This further allows you to combat bufferbloat at the application layer.

Reply to this comment

[–] Zero-copy networking

Posted Jul 4, 2017 2:04 UTC (Tue) by **ncm** (guest, #165) [Link] (5 responses)

The last time I discussed zero-copy networking with a kernel person (this was in the context of NetBSD's UVM, which worked automatically, given certain preconditions), it was explained to me that the benefit available plummets when you go from one to two-or-more CPUs. The unintuitive reason was that flipping the page tables, as UVM did, invalidates caches on other CPUs, which more than eats up any savings from not copying. Anyway, chip designers put enormous efforts into making sequential copying fast. It's the single most heavily optimized operation in any system, so it is stiff competition.

So, this scheme, that requires just a little more cooperation from user space to avoid re-mappings, seems like the minimum needed to make the process useful at all. (That approximates the definition of good engineering.) A practical saving of 5% isn't much for most of us, but it would pay salaries of dozens of engineers at a Facebook or Netflix. Indeed, it would be negligent of each of them not to have implemented something like this, so I count it likely that many have cobbled-up versions of this in-house that work just well enough for their purposes. This work was done at Google; probably they, too, had a cobbled-up version that worked just well enough, which then took several times more engineering effort to clean up for publication and upstreaming. The negligent among the other big shops get the benefit for free, but only years later, and the rest (Google included) can delete their hard-to-maintain cobbled-up versions.

Probably the biggest benefits of this work go to (often big) customers of small vendors who haven't the engineering staff to devote to faking it, and who in any case don't control what kernel their code runs on. The small shops can reasonably be asked to have their code use the feature when running on a kernel that supports it. Google, meanwhile, will have reasonable hope that, eventually, companies they buy already use the feature, and the code that comes with them doesn't need to be jimmied to use a cobbled-up, proprietary version.

Reply to this comment

[–] Zero-copy networking

Posted Jul 4, 2017 17:55 UTC (Tue) by **jhoblitt** (subscriber, #77733) [Link] (3 responses)

I have heard that the CPU utilization on the megasize infrastructures averages around 5%. I don't have a citation for that, and have no idea how close that number is to reality. However, if that is in the ballpark, a 5% syscall savings probably won't have a major TCO impact for most applications.

Reply to this comment

[–] Zero-copy networking

Posted Jul 4, 2017 21:46 UTC (Tue) by **ncm** (guest, #165) [Link]

It is legitimate to bring up numbers, but I doubt that the monetary value of performance is measured by TCO alone, or by average CPU utilization. There are service level agreements, and peak-traffic latency, and potential income. How many ads can be delivered, and displayed for the required amount of time before the user clicks past, at 5:30 PM? How frequently does the streaming video skip, at 6:30 PM, and how does that affect subscription renewal rates? How many product pages can be presented per potential customer per minute, and how does the conversion rate (from looking to spending money) vary with how long they have to wait to see the next page?

Reply to this comment

[–] Zero-copy networking

Posted Jul 5, 2017 3:47 UTC (Wed) by **gdt** (subscriber, #6284) [Link] (1 responses)

5% gains are worthwhile. CPUs get faster by about 5% a year from process improvements alone. So a 5% gain obtained from software can delay a hardware upgrade by a year. For which you can calculate a TCO value.

The more worrying thought is that semiconductor process miniaturisation seems to be coming to an end: each node shrink is successively later. At some point in the late-2020s future the major source for throughput improvement will move to software. Ie: assume a 5nm process is economic around 2024, allow two generations of CPU design tuning, after that the "CPU" barrel is empty of economic performance improvements. We can still look to other hardware components for performance enhancement -- particularly in the way hardware hangs together -- but such system-level design change usually requires enabling software in the operating system and in key applications. In short, this patch is a look into the future of computer performance.

Reply to this comment

[–] Zero-copy networking

Posted Jul 13, 2017 14:05 UTC (Thu) by **Wol** (subscriber, #4433) [Link]

Transistor miniaturisation has pretty much peaked ... The width of a conducting line, and the insulator between it, can now be measured in atoms, and it's not many of them ...

The biggest problem with shrinking dies further is now quantum leakage, which will only grow. In order to make chips faster, they are now having seriously to contend with the speed of light, which is why chips are spreading more and more into the realms of 3d.

Cheers,
Wol

Reply to this comment

[–] Zero-copy networking

Posted Jul 11, 2017 10:47 UTC (Tue) by **abo** (subscriber, #77288) [Link]

Netflix does have a zero-copy sendfile() in their FreeBSD based CDN. They also (partially)

implemented TLS in the kernel.

https://medium.com/netflix-techblog/protecting-netflix-vi...

Reply to this comment

[–] Zero-copy networking

Posted Jul 4, 2017 12:18 UTC (Tue) by **sorokin** (guest, #88478) [Link] (7 responses)

Linux people should keep an eye on what Microsoft do. Microsoft had zero copy networking for ages. I believe they had had I/O completion ports (queues where notifications that I/O operation is completed are pushed) even before Linux got epoll support.

In Microsoft they realized that locking pages in memory is expensive. And in Windows 8 they come up with an API called Registered I/O. It requires an application to register buffers in advance. Then I/O operations just use these registered buffers and therefore don't need to lock any pages.

I believe in Linux kernel people should just skip designing zero-copy operations altogether and just implement Registered I/O.

Reply to this comment

[–] Zero-copy networking

Posted Jul 4, 2017 18:49 UTC (Tue) by **einstein** (subscriber, #2052) [Link] (1 responses)

Good point - In this case, Linux devs ought to look at borrowing worthwhile features even from crappy OSes.

Reply to this comment

[–] Zero-copy networking

Posted Dec 19, 2017 22:23 UTC (Tue) by **immibis** (subscriber, #105511) [Link]

The Windows I/O stack has always felt lightyears ahead of Linux to me.

Just because several components are tied together in one product with annoying marketing, doesn't mean they all suck. I'm sure the kernel I/O developers had nothing to do with the start screen.

Reply to this comment

[–] Zero-copy networking

Posted Jul 5, 2017 13:17 UTC (Wed) by **abatters** (★ **supporter** ★, #6932) [Link]

I don't know Windows, but this is exactly what I was thinking. Register the buffers ahead of time to save the overhead of pinning and unpinning the pages over and over again. See also: SCSI generic (drivers/scsi/sg.c) mmap()ed I/O.

Reply to this comment

[–] Zero-copy networking

Posted Jul 5, 2017 14:35 UTC (Wed) by **clameter** (subscriber, #17005) [Link] (2 responses)

Linux has had zero copy networking for more than 10 years. Use the RDMA subsystem to send messages. The RDMA subsystem can even receive messages(!!!). The RDMA subsystem can not only perform remote DMA operations but also send and receive datagrams.

Reply to this comment

[–] Zero-copy networking

Posted Jul 6, 2017 21:52 UTC (Thu) by **wtarreau** (subscriber, #51152) [Link] (1 responses)

It has even worked for userland for a while. HAProxy successfully makes use of splice() to perform zero-copy transfers between sockets (receive AND send).

Also it seems to me that this send(MSG_ZEROCOPY) is not much different from doing a vmsplice().

Reply to this comment

[–] Zero-copy networking

Posted Jul 11, 2017 0:00 UTC (Tue) by **klempner** (subscriber, #69940) [Link]

The entire point of MSG_ZEROCOPY is the notification that the kernel is done so the memory can be unpinned and potentially freed/reused. This isn't a problem for HAProxy's socket to splice pipe to socket doesn't have that problem because in that case the memory never has a userspace address.

The fundamental problem with application to vmsplice pipe to TCP socket is that you don't know when the pages in question are done and can be freed/modified, and if you modify them you're liable to start, say, leaking crypto keys out to the network if that memory gets reused before the TCP stack is done with it.

Reply to this comment

[–] Zero-copy networking

Posted Jul 5, 2017 21:27 UTC (Wed) by **k8to** (guest, #15413) [Link]

Async I/O a la completion ports shouldn't be news to anyone, given that this was the standard means of doing I/O on VMS in the 1980s. The downsides are that's it's tricker to get right (there's a lot more opportunity for creating bugs in the application code), and that reaping the benefits means receiving data in an interface that looks essentially nothing like sockets.

Those are prices that must be paid for completely maximizing throughput in high transaction count scenarios, but they're an awkward price for most users.

The registered I/O tweak is relatively recent and somewhat informative however, driven as it is by modern hardware requirements instead of ancient design concerns.

Reply to this comment

[–] Zero-copy networking

Posted Jul 5, 2017 6:37 UTC (Wed) by **marcH** (subscriber, #57642) [Link] (5 responses)

> Reading is inherently harder because it is not generally known where a packet is headed when the network interface receives it.

Interruptions is another reason why receiving is "harder"/more expensive than sending. As usual, predicting the future is very difficult.

> With remote DMA, the sender of the packet puts the memory address of the receiver's buffer into the packet, and the receiving interface just writes it there. This requires that the receiver has told the sender in advance...

For high performance, not just RDMA but message passing interfaces also involve a certain amount of "telling in advance", at least for large messages.

The socket API sucks, really not designed for high performance: "I'm receiving, send me whatever whenever and I'll manage somehow!" Very slowly.

Last I checked I wasn't even possible for user space to tell the kernel: "I'm expecting to recv() 2kbytes, don't wake me up and waste our time until you got at least that". BTW I suspect many applications would break if IP fragmentation was real.

Reply to this comment

[–] Zero-copy networking

Posted Jul 6, 2017 3:56 UTC (Thu) by **ncm** (guest, #165) [Link] (1 responses)

I have seen IP fragmentation in the wild.

Reply to this comment

[–] Zero-copy networking

Posted Jul 6, 2017 10:42 UTC (Thu) by **paulj** (subscriber, #341) [Link]

IP fragments are how some applications send larger-than-MTU sized application-layer messages.

Reply to this comment

[–] Zero-copy networking

Posted Jul 6, 2017 17:32 UTC (Thu) by **sorokin** (guest, #88478) [Link] (1 responses)

> Last I checked I wasn't even possible for user space to tell the kernel: "I'm expecting to recv() 2kbytes, don't wake me up and waste our time until you got at least that".

Although I agree that having this type of function in kernel would be good, I believe the benefits of it are small compared to what we have now. Let me explain. There is 2 options: either the chunk of data you expect is small in size or it is big.

In case it is small. It is likely fit one packet or a few packets with a small interval between them. In this case by the time the program is woken up tailing packets have already arrived.

In case it is big. The program is unlikely to buffer all the data in memory. Instead it is likely to write this data to disk/handle it somehow as it is coming. And in this case waking up the program is what you want.

So the only use case for this function would be a program that receives large amount of data buffering it in memory until the last byte arrives.

Reply to this comment

[–] Zero-copy networking

Posted Jul 6, 2017 17:43 UTC (Thu) by **marcH** (subscriber, #57642) [Link]

> In this case by the time the program is woken up tailing packets have already arrived.

This fully qualifies the definition of a race condition.

On one hand there's the performance question which you addressed. On the other hand there's the fact that some applications don't bother coding a retry loop around the recv() call. Combine this with the race condition above and let the fun begin.

Reply to this comment

[–] Zero-copy networking

Posted Sep 21, 2017 7:38 UTC (Thu) by **njs** (subscriber, #40338) [Link]

> Last I checked I wasn't even possible for user space to tell the kernel: "I'm expecting to recv() 2kbytes, don't wake me up and waste our time until you got at least that".

recv(2) says it was added in Linux 2.2 :-) (see MSG_WAITALL)

Unfortunately this doesn't help if you're using non-blocking sockets.

Reply to this comment