Page-Flip Technology for use within the Linux Networking Stack

John A. Ronciak
Intel Corporation
`john.ronciak@intel.com`

Jesse Brandeburg
Intel Corporation
`jesse.brandeburg@intel.com`

Ganesh Venkatesan
Intel Corporation
`ganesh.venkatesan@intel.com`

Abstract

Today's received network data is copied from kernel-space to user-space once the protocol headers have been processed. What is needed is to provide a *hardware (NIC) to user-space* zero-copy path. This paper discusses a page-flip technique where a page is *flipped* from kernel memory into user-space via page-table manipulation. Gigabit Ethernet was used to produce this zero-copy receive path within the Linux stack which can then be extrapolated to 10 Gigabit Ethernet environments where the need is more critical. Prior experience in the industry with page-flip methodologies is cited.

The performance of the stack and the overall system is presented along with the testing methodology and tools used to generate the performance data. All data was collected using a modified TCP/IP stack in a 2.6.x kernel. The stack modifications are described in detail. Also discussed is what hardware and software features are required to achieve page-flipping.

The issues involving page-flipping are described in detail. Also discussed are problems related to this technology concerning the Virtual Memory Manager (VMM) and processor cache. Another issue that is discussed is what would be needed in an API or code changes to enable user-space applications.

The consequences and possible benefits of this technology are called out within the conclusions of this study. Also described are the possible next steps needed to make this technology viable for general use. As faster networks like 10 Gigabit Ethernet become more commonplace for servers and desktops, understanding and developing zero-copy receive mechanisms within the Linux kernel and networking stack is becoming more critical.

Introduction

Data arriving at a network port undergoes two copy operations (a) from the device memory to kernel memory as a DMA by the device into host memory and (b) from kernel memory to application memory, copied by the processor. Techniques that avoid the second copy are designated zero-copy; no additional copy operations are involved once the data is copied into host memory. Avoiding the second copy can potentially improve throughput and reduce CPU utilization. This has been demonstrated in [Hurd] [Duke] and [Gallatin]. Several techniques have been discussed in the literature for avoiding the second copy namely page flipping, direct data placement (DDP) and remote DMA (RDMA).

Significant performance benefits were demonstrated with the zero copy implementation in the transmit path. We investigate the effectiveness of the page flipping on newer platforms (faster processor(s) and faster memory). Additional motivation for this experiment and paper came from a discussion on the netdev (and linux-kernel) mailing list where David Miller mentioned his idea of

> On receive side, clever RX buffer flipping tricks are the way to go and require no protocol changes and nothing gross like TOE or weird buffer ownership protocols like RDMA requires.[1]

Approaches

Our initial approach consisted of attempting to modify the 2.4 kernel to support direct modification of PTE's in user and kernel space. This method was based on the assumption that any PTE could represent any location in memory which we later found out not to be true. Our findings indicated that we needed to rely more upon the OS abstraction layers to complete our page-flip implementation. This had the side benefit of making our changes less x86 specific as well. Eventually we settled upon a 2.6 based kernel and effectively implemented our original idea but instead just install a new page into the application space in much the same way as the swapper does. The biggest hurdles came from understanding how the Linux memory manager and its various kernel structures work and relate to each other.

[1]`http://marc.theaimsgroup.com/?l=linux-netdev&w=2&r=1&s=TCP+offloading+interface&q=b`

For our final experiments we used 2.6.4 or newer kernels with what eventually amounted to small changes to the kernel to support page flipped PAGE_SIZE data.

The kernel code consisted of these changes (see patch at the end of this document):

1. Driver modifications to support header and data portions of a packet in separate buffers, where the data buffer is always aligned to a PAGE_SIZE boundary.
2. Add a flag to the skb structure to indicate to the stack that the hardware and driver prepared a zero copy capable receive structure.
3. Modifications to the `skb_copy_datagram_iovec()` function to support calling the new `flip_page_mapping()` function when zero copy capable skbs are received.
4. A new `flip_page_mapping()` function that executes the installation of the driver page into the user's receive data space. This routine handles fixing up permissions.
5. A modification was made to the skb free routines to handle a frags[i] where the .page member was zero after that page had changed ownership to user space.

Experiment

Our test platform consisted of a pre-release system with a dual 2.4 GHz Intel® Pentium® 4 processor supporting Hyper-Threading Technology, and 512 megabytes of RAM. This machine had a network card that supported splitting the header and data portions of a packet into different buffers, and validating the IP, TCP and Ethernet checksums.

Assumptions

For this experiment we made some assumptions to simplify and to work with the hardware that we had available.

- Our application had to allocate a receive data area in multiples of 4K bytes, and that memory had to be PAGE_SIZE aligned.
- We modified the freely available nttcp-1.47 to use valloc instead of malloc, resulting in PAGE_SIZE aligned memory starting addresses.
- Our network used Maximum Transmission Units (MTU) to allow for 4KB or 8KB of data to be packaged in every packet.
- Upon splitting of the packet into header and data portions, this resulted in an aligned data block
- The 2.6.4 kernel was configured for standard 4KB PAGE_SIZE and debugging options were turned off.

Methodologies

After making the required code changes and debugging, we measured the performance of the new "page flip" code against the "copy once" method of receiving data.

These measurements consisted of two major test runs, one where the application never touched the data (notouch) being received, and the other where the application did a comparison of the data to an expected result (touch), effectively forcing the data into the cache and also validating that data was not corrupted in any way through this process.

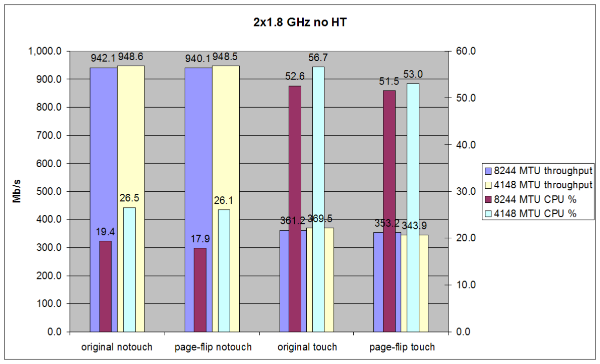

Figure 1: 1.8 GHz comparison

For every instance of the test, three runs were done and the results were averaged for each data point.

Oprofile was used to record the hot-spots for each run.

CPU utilization and network utilization were measured with sar from the sysstat package. NOTE: Our initial results were skewed by a version of sar that incorrectly measured CPU and network utilization (showing more than 1Gb/s transferred in a single direction), be aware that some versions of sar that shipped with your distribution may need to be updated.

Results of Performance Analysis

It is apparent from the touch graphs in Figure 1 that the page flip slightly reduces CPU on slower processors. However, the touch throughput decreases as well, with a decrease in efficiency (Mbits/CPU = eff) for the 4148 MTU from 6.52 (original) to 6.48 (page-flip). The decrease in efficiency is even smaller for 8244 MTUs, where the efficiency went from 6.86 to 6.85. The difference in CPU from the 8244 to the 4148 MTU case is most likely due to header processing as the data throughput is very similar.

The difference between Figure 1 and Figure 2 is simply the processor's speed being adjusted

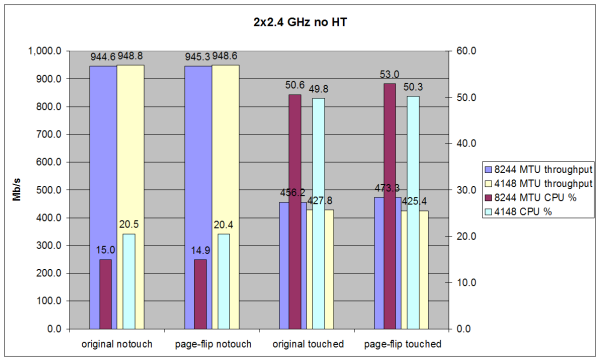

Figure 2: 2.4 GHz comparison

in the bios using a multiplier change. The results from Figure 2 show that the faster processor is more efficient overall, but that even if there is a slight increase in throughput for the page-flip case, the efficiency is still less than if the copy was being done. The efficiency for the 4148 MTU touch data case went from 8.59 to 8.45. For the 8244 byte MTU the efficiency goes from 9.02 to 8.93, even though the throughput goes up.

Surprises and Unexpected Results

We expected that the copy may actually have some beneficial side effects, and our data shows that it does. Especially as processor clock rate increases, the copy becomes less costly in CPU-utilization, while the page flip maintains a constant load which is heavier than the copy was initially.

Oprofile analysis indicated that the locks associated with the page-flip code cause the majority of the stalls in this code path.

Oprofile also showed that the stall associated with the TLB (translation look-aside buffer) flush was very painful.

Conclusions

We had several surprises along the way, but feel confident that at least with our current code base, we can conclude that using a page-flip methodology to receive network data is less efficient than simply doing a copy. The major contributors to this counterintuitive result seem to be cache issues (especially obvious in the "touched data" tests), and a heavier cost associated with the work necessary to prepare and complete the page-flip.

There may be environments such as embedded systems and slower processors where page-flipping will help significantly in decreasing CPU utilization or increasing performance.

Our feeling is that page flipping will not scale in CPU utilization as well as a plain copy does.

There is much room however for optimization of the page-flip code path, which will be followed up with the community. Our expectation is that this optimization will be fighting an uphill battle just to achieve parity with a copy, and then will mostly likely not be able to keep up with speed advances in the processor.

Also, we had to remind ourselves that the cache warming cost must be paid somewhere along all receive paths. Using page-flip methods only moves the cost of the cache miss to the application instead of taking the cost of the miss in the kernel. If the application is waiting impatiently for data, its likely that the cache will be seeded with the data and the application will get all of its data out of cache and have very fast access at that point.

Current issues

The current patch has several outstanding issues that we worked around.

1. There isn't much (if any) commercially available hardware that supports header split receives.

2. Ideally hardware (as mentioned by David Miller) would be able to have flow identification and fill `PAGE_SIZE` buckets with data. This would eliminate the requirement for specific MTU sizes.

3. The current code has a bug when a network data consumer causes a `clone_skb()` to occur. If a page-flipped page pointer `nr_frags[].page` is referenced in the skb being cloned, then a zero pointer is read and the system faults. This is due to the ownership of the page changing from kernel space to user space before the clone is completed. It is not immediately clear if this is an easily surmountable problem, but is easy to work around for our tests.

4. The assumptions we made to enable testing this new code path, like specifying MTU, recompiling the application, etc, create such strict requirements that the usefulness of this code outside of an academic environment is severely limited.

Future directions possible

It is likely that on a system with lots of context switching going on (high load) that the page-flip would be more beneficial. Testing in these environments would provide useful results.

If tested on other architectures besides x86, such as x86-64, IA64 and PPC this code may yield significantly different results.

We did create a driver patch (Appendix B) for the currently available e1000 driver and hardware that prepares packets (using a copy) for processing through the page-flip modified network stack to the user application. We saw that the copy necessary in the driver to do this made the differences between "driver with copy followed by a flip in the stack" and a "driver with a copy followed by another copy in the stack" almost nonexistent. We believe this is because of the cache warming done by the Appendix B driver as it prepares the flip capable structure. Making this code behave more like the flip capable hardware (possibly with a cache flush) would be very useful to increase the amount of experimentation that could be done with the non-hardware specific kernel patches.

References

[Hurd] Dana Hurd Zero-copy interfacing to TCP/IP Dr. Dobbs Journal Sep 1995

[Duke] Trapeze Project: `http://www.cs.duke.edu/ari/trapeze/slides/freenix/sld001.htm`

[Gallatin] Drew Gallatin `http://people.freebsd.org/~ken/zero_copy/`

Appendix A Kernel Patch

This patch will be available at `http://www.aracnet.com/~micro/flip/flip_2_6_4.patch.bz2`

Appendix B mock zero copy e1000 patch

This patch will be available at `http://www.aracnet.com/~micro/flip/e1000_flip.patch.bz2`

Proceedings of the Linux Symposium

Volume Two

July 21st–24th, 2004
Ottawa, Ontario
Canada