Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Networking Driver Performance and Measurement - e1000 A Case Study

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

Mitch Williams
Intel Corporation
mitch.a.williams@intel.com

Abstract

Networking performance is a popular topic in Linux and is becoming more critical for achieving good overall system performance. This paper takes a look at what was done in the e1000 driver to improve performance by (a) increasing throughput and (b) reducing of CPU utilization. A lot of work has gone into the e1000 Ethernet driver as well into the PRO/1000 Gigabit Ethernet hardware in regard to both of these performance attributes. This paper covers the major things that were done to both the driver and to the hardware to improve many of the aspects of Ethernet network performance. The paper covers performance improvements due to the contribution from the Linux community and from the Intel group responsible for both the driver and hardware. The paper describes optimizations to improve small packet performance for applications like packet routers, VoIP, etc. and those for standard and jumbo packets and how those modifications differs from the small packet optimizations. A discussion on the tools and utilities used to measure performance and ideas for other tools that could help to measure performance are presented. Some of the ideas may require help from the community for refinement and implementation.

Introduction

This paper will recount the history of e1000 Ethernet device driver regarding performance. The e1000 driver has a long history which includes numerous performance enhancements which occurred over the years. It also shows how the Linux community has been involved with trying to enhance the drivers' performance. The notable ones will be called out along with when new hardware features became available. The paper will also point out where more work is needed in regard to performance testing. There are lots of views on how to measure network performance. For various reasons we have had to use an expensive, closed source test tool to measure the network performance for the driver. We would like to engage with the Linux community to try to address this and come up with a strategy of having an open source measurement tool along with consistant testing methods.

This paper also identifies issues with the system and the stack that hinder performance. The performance data also indicates that there is room for improvement.

A brief history of the e1000 driver

The first generation of the Intel® PRO/1000 controllers demonstrated the limitation of the 32-bit 33MHz PCI bus. The controllers were able to saturate the bus causing slow response times for other devices in the system (like slow video updates). To work with this PCI bus bandwidth limitation, the driver team worked on identifying and eliminating inefficiencies. One of the first improvements we made was to try to reduce the number of DMA transactions across the PCI bus. This was done using some creative buffer coalescing of smaller fragments into larger ones. In some cases this was a dramatic change in the behavior of the controller on the system. This of course was a long time ago and the systems, both hardware and OS have changed considerably since then.

The next generation of the controller was a 64-bit 66MHz controller which definitely helped the overall performance. The throughput increased and the CPU utilization decreased just due to the bus restrictions being lifted. This was also when new offload features were being introduced into the OS. It was the first time that interrupt moderation was implemented. This implementation was fairly crude, based on a timer mechanism with a hard time-out time set, but it did work in different cases to decrease CPU utilization.

Then a number of different features like descriptor alignment to cache lines, a dynamic inter-frame gap mechanism and jumbo frames were introduced. The use of jumbo frames really helped transferring large amounts of data but did nothing to help normal or small sized frames. It also was a feature which required network infrastructure changes to be able to use, e.g. changes to switches and routers to support jumbo frames. Jumbo frames also required the system stacks to change. This took some time to get the issues all worked out but they do work well for certain environments. When used in single subnet LANs or clusters, jumbo frames work well.

Next came the more interesting offload of checksumming for TCP and IP. The IP offload didn't help much as it is only a checksum across twenty bytes of IP header. However, the TCP checksum offload really did show some performance increases and is widely used today. This came with little change to the stack to support it. The stack interface was designed with the flexibility for a feature like this. Kudos to the developers that worked on the stack back then.

NAPI was introduced by Jamal Hadi, Robert Olsson, et al at this time. The e1000 driver was one of the first drivers to support NAPI. It is still used as an example of how a driver should support NAPI. At first the development team was unconvinced that NAPI would give us much of a benefit in the general test case. The performance benefits were only expected for some edge case situations. As NAPI and our driver matured however, NAPI has shown to be a great performance booster in almost all cases. This will be shown in the performance data presented later in this paper.

Some of the last features to be added were TCP Segment Offload (TSO) and UDP fragment checksums. TSO took work from the stack maintainers as well as the e1000 development team to get implemented. This work continues as all the issues around using this have not yet been resolved. There was even a rewrite of the implementation which is currently under test (Dave Miller's TSO rewrite). The UDP

fragment checksum feature is another that required no change in the stack. It is however little used due to the lack of use of UDP checksumming.

The use of PCI Express has also helped to reduce the bottleneck seen with the PCI bus. The significantly larger data bandwidth of PCIe helps overcome limitations due to latencies/overheads compared to PCI/PCI-X buses. This will continue to get better as devices support more lanes on the PCI Express bus further reducing bandwidth bottlenecks.

There is a new initiative called Intel® I/O Acceleration Technology (I/OAT) which achieves the benefits of TCP Offload Engines (TOE) without any of the associated disadvantages. Analysis of where the packet processing cycles are spent was performed and features designed to help accelerate the packet processing. These features will be showing up over the next six to nine months. The features include Receive Side Scaling (RSS), Packet Split and Chipset DMA. Please see the [Leech/Grover] paper "Accelerating Network Receive Processing: Intel® I/O Acceleration Technolgy" presented here at the symposium. RSS is a feature which identifies TCP flows and passes this information to the driver via a hash value. This allows packets associated with a particular flow to be placed onto a certain queue for processing. The feature also includes multiple receive queues which are used to distribute the packet processing onto multiple CPUs. The packet split feature splits the protocol header in a packet from the payload data and places each into different buffers. This allows for the payload data buffers to be page-aligned and for the protocol headers to be placed into small buffers which can easily be cached to prevent cache thrash. All of these features are designed to reduce or eliminate the need for TOE. The main reason for this is that all of the I/OAT features will scale with processors and chipset technologies.

Performance

As stated above the definition of performance varies depending on the user. There are a lot of different ways and methods to test and measure network driver performance. There are basically two elements of performance that need to be looked at, throughput and CPU utilization. Also, in the case of small packet performance, where packet latency is important, the packet rate measured in packets per second is used as a third type of measurement. Throughput does a poor job of quantifying performance in this case.

One of the problems that exists regarding performance measurements is which tools should be used to measure the performance. Since there is no consistent open source tool, we use a closed source expensive tool. This is mostly a demand from our customers who want to be able to measure and compare the performance of the Intel hardware against other vendors on different Operating Systems. This tool, IxChariot by IXIA[1], is used for this reason. It does a good job of measuring throughput with lots of different types of traffic and loads but still does not do a good job of measuring CPU utilization. It also has the advantage that there are endpoints for a lot of different OSes. This gives you the ability to compare performance of different OSes using the same system and hardware. It would be nice to have and Open Source tool which could do the same thing. This is discussed in Section , "Where do we go from here."

There is an open source tool which can be used to test small packet performance. The tool is the packet generator or 'pktgen' and is a kernel module which is part of the Linux kernel. The tool is very useful for sending lots of packets with set timings. It is the tool of choice for

[1] Other brands and names may be claimed as the property of others.

anyone testing routing performance and routing configurations.

All of the data for this section was collected using Chariot on the same platform to reduce the number of variables to control except as noted.

The test platform specifications are:

- Blade Server
- Dual 2.8GHz Pentium® 4 Xeon™ CPUs, 512KB cache 1GB RAM
- Hyperthreading disabled
- Intel® 80546EB LAN-on-motherboard (PCI-X bus)
- Competition Network Interface Card in a PCI-X slot

The client platform specifications are:

- Dell[2] PowerEdge® 1550/1266
- Dual 1266MHz Pentium® III CPUs, 512KB cache, 1GB RAM,
- Red Hat[2] Enterprise Linux 3 with 2.4.20-8smp kernel,
- Intel® PRO/1000 adapters

Comparison of Different Driver Versions

The driver performance is compared for a number of different e1000 driver versions on the same OS version and the same hardware. The difference in performance seen in Figure 1 was due to the NAPI bug that Linux community found. It turns out that the bug was there for a long time and nobody noticed it. The bug was causing the driver to exit NAPI mode back into interrupt mode fairly often instead of staying in NAPI mode. Once corrected the number of interrupts taken was greatly reduced as it should be when using NAPI.

Comparison of Different Frames Sizes verses the Competition

Frame size has a lot to do with performance. Figure 2 shows the performance based on frame size against the competition. As the chart shows, frame size has a lot to do with the total throughput that can be reached as well as the needed CPU utilization. The frame sizes used were normal 1500 byte frames, 256 bytes frames and 9Kbyte jumbo frames.

NOTE: The competition could not accept a 256 byte MTU so 512 bytes were used for performance numbers for small packets.

Comparison of OS Versions

Figure 3 shows the performance comparison between OS versions including some different options for a specific kernel version. There was no reason why that version was picked other than it was the latest at the time of the tests. As can be seen from the chart in Figure 3, the 2.4 kernels performed better overall for pure throughput. This means that there is more improvement to be had with the 2.6 kernel for network performance. There is already new work on the TSO code within the stack which may have improved the performance already as the TSO code has been known to hurt the overall network throughput. The 2.4 kernels do not have TSO which could be accounting for at least some of the performance differences.

[2]Other brands and names may be claimed as the property of others.

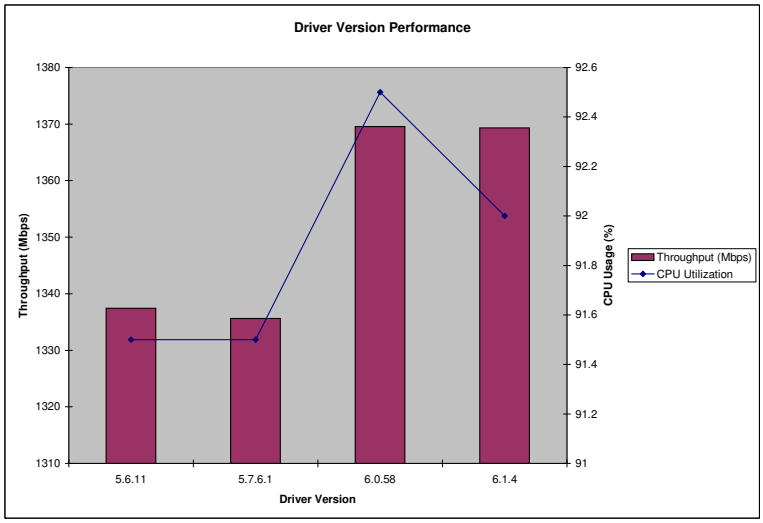

Figure 1: Different Driver Version Comparison

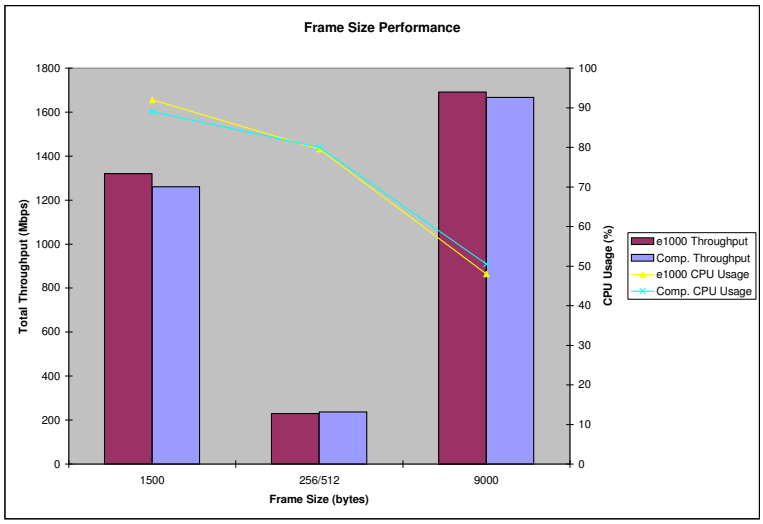

Figure 2: Frame Size Performance Against the Competition

Results of Tuning NAPI Parameters

Initial testing showed that with default NAPI settings, many packets were being dropped on receive due to lack of buffers. It also showed that TSO was being used only rarely (TSO was not being used by the stack to transmit).

It was also discovered that reducing the driver's weight setting from the default of 64 would eliminate the problem of dropped packets. Further reduction of the weight value, even to very small values, would continue to increase throughput. This is shown in Figure 4.

The explanation for these dropped packets is simple, because the weight is smaller, the driver iterates through its packet receive loop (in `e1000_clean_rx_irq`) fewer times, and hence writes the Receive Descriptor Tail regis-

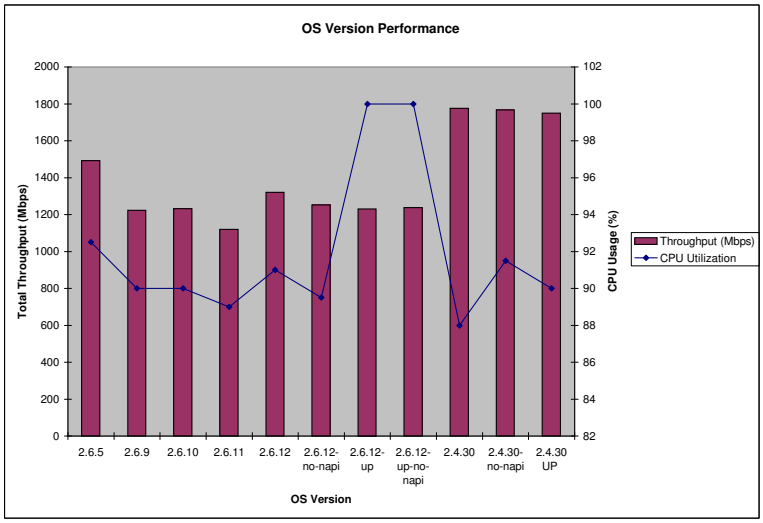

Figure 3: OS Version Performance

ter more often. This notifies the hardware that descriptors are available more often and eliminates the dropped packets.

It would be obvious to conclude that the increase in throughput can be explained by the dropped packets, but this turns out to not be the case. Indeed, one can eliminate dropped packets by reducing the weight down to 32, but the real increase in throughput doesn't come until you reduce it further to 16.

The answer appears to be latency. With the higher weights, the NAPI polling loop runs longer, which prevents the stack from running its own timers. With lower weights, the stack runs more often, and processes packets more often.

We also found two situations where NAPI doesn't do very well compared to normal interrupt mode. These are 1) when the NAPI poll time is too fast (less than time it takes to get a packet off the wire) and 2) when the processor is very fast and I/O bus is relatively slow. In both of these cases the driver keeps entering NAPI mode, then dropping back to interrupt mode since it looks like there is no work to do. This is a bad situation to get into as the driver has to take a very high number of interrupts to get the work done. Both of these situations need to be avoided and possibly have a different NAPI tuning parameter to set a minimum poll time. It could even be calculated and used dynamically over time.

Where the Community Helped

The Linux community has been very helpful over the years with getting fixes back to correct errors or to enhance performance. Most recently, Robert Olsson discovered the NAPI bug discussed earlier. This is just one of countless fixes that have come in over the years to make the driver faster and more stable. Thanks to all to have helped this effort.

Another area of performance that was helped by the Linux community was the e1000 small packet performance. There were a lot of comments/discussions in netdev that helped to get the driver to perform better with small packets. Again, some of the key ideas came from Robert

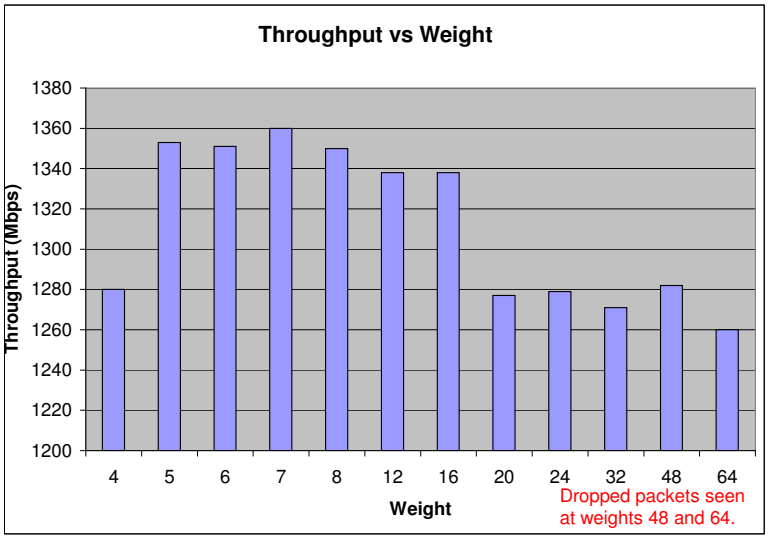

Figure 4: NAPI Tuning Performance Results

Olsson with the work he has done on packet routing. We also added different hardware features over the years to improve small packet performance. Our new RSS feature should help this as well since the hardware will be better able to scale with the number of processor in the system. It is important to note that e1000 benefitted a lot from interaction with the Open Source Community.

Where do we go from here

There are a number of different things that the community could help with. A test tool which can be used to measure performance across OS versions is needed. This will help in comparing performance under different OSes, different network controllers and even different versions of the same driver. The tool needs to be able to use all packet sizes and OS or driver features.

Another issue that should be addressed is the NAPI tuning as pointed out above. There are cases where NAPI actually hurts performance but with the correct tuning works much better.

Support the new I/OAT features which give most if not all the same benefits as TOE without the limitations and drawbacks. There are some kernel changes that need to be implemented to be able to support features like this and we would like for the Linux community to be involved in that work.

Conclusions

More work needs to be done to help the network performance get better on the 2.6 kernel. This won't happen overnight but will be a continuing process. It will get better with work from all of us. Also, work should continue to make NAPI work better in all cases. If it's in your business or personal interest to have better network performance, then it's up to you help make it better.

Thanks to all who have helped make everything perform better. Let us keep up the good work.

References

[Leech/Grover] Accelerating Network Receive Processing: Intel® I/O Acceleration Technolgy; Ottawa Linux Symposium 2005