Tuning 10Gb network cards on Linux

A basic introduction to concepts used to tune fast network cards

Breno Henrique Leitao
IBM
leitao@linux.vnet.ibm.com

Abstract

The growth of Ethernet from 10 Mbit/s to 10 Gbit/s has surpassed the growth of microprocessor performance in mainstream servers and computers. A fundamental obstacle to improving network performance is that servers were designed for computing rather than input and output.

The processing of TCP/IP over Ethernet is traditionally accomplished by a software running on the central processor of the server. As network connections scale beyond Gigabit Ethernet speeds, the CPU becomes burdened with the large amount of TCP/IP protocol processing required by the current network speed. Reassembling out-of-order packets, handling the protocols headers, resource-intensive memory copies, and interrupts put a tremendous load on the system CPU, resulting in a CPU doing almost I/O traffic instead of running applications.

During the network evolution, each generation of new cards presents a lot of features that increase the performance of the network, and when not properly configured, can harm the overall system and network performance.

Linux, on the other side, is an operating system that runs from embedded system to super computers, and its default configuration is not tuned to run 10 Gbit/s network cards on wire speed, and it possibly will limit the total available throughput artificially. Hence, some modifications in the system is required to achieve good performance. Most of these performance modifications are easy to do, and doesn't require a deep knowledge of the kernel code in order to see the results.

This paper will describe most of the basic settings that should be set in a Linux environment in order to get the maximum speed when using fast network adapters. Since this is a vast topic, this paper will focus on the basic concepts.

1 Terminology

This section will cover basic terminology used in the article. For other terminologies, RFC2647[1] is a good place to start looking for.

Throughput

The term throughput basically has the same meaning as data transfer rate or digital bandwidth consumption, and denotes the achieved average useful bit rate in a computer network over a physical communication link. The throughput is basically measured with little overhead, and for reference, it is measured below the network layer and above the physical layer. In this way, throughput includes some protocol overhead and retransmissions.

When talking about network terminology, it is worth to remember that one Kbit/s means 1,000 bit/s and not 1,024 bit/s.

Round trip time

Round trip time (RTT) is the total amount of time that a packet takes to reach the target destination and get back to the source address.

Bandwidth delay product

Bandwidth Delay Product (BDP) is an approximation for the amount of data that can be in flow in the network during a time slice. Its formula is simply a product of the link bandwidth and the Round Trif Time. To compute the BDP, it is required to know the speed of the slowest link in the path and the Round Trip Time (RTT) for the same path, where the bandwidth of a link is expressed in Gbit/s and the round-trip delay is typically between 1

msec and 100 msec, which can be measured using `ping` or `traceroute`.[1]

In TCP/IP, the BDP is very important to tune the buffers in the receive and sender side. Both side need to have an available buffer bigger than the BDP in order to allow the maximum available throughput, otherwise a packet overflow can happen because of out of free space.

1.1 Jumbo Frames

Jumbo frames are Ethernet frames that can carry more than the standard 1500 bytes of payload. It was defined that jumbo frames can carry up to 9000 bytes, but some devices can support up to 16128 bytes per frame. In this case, these super big packets are referenced as Super Jumbo Frames. The size of the frame is directly related to how much information is transmitted in a slot,[2] and is one of the major factors to increase the performance on the network, once the amount of overhead is smaller.

Almost all 10 Gbit/s switches support jumbo frames, and new switches supports super jumbo frames. Thus, checking if the network switches support these frames is a requirement before tuning this feature on.

Frame size is directly related to the interface Maximum Transfer Unit (MTU), and it is a specific adapter configuration that must be set for each node in a network. Moreover, all interfaces in the same network should have the same MTU in work to communicate properly, a different MTU on a specific node could cause awful issues.

Setting the MTU size uses the following command `ifconfig <interface> mtu <size>`.

It is important to note that setting the MTU to a high number, does not mean that all your traffic would use jumbo packets. For example, a normal ssh session is almost insensible to a MTU change, once almost all SSH packets are sent using small frames.

Once you the jumbo frames are enabled on an interface, the software application should start using it, otherwise the performance will not be as good as expected.

[1] Although there are better ways to measure the packet RTT, these tools are enough for the purpose of this paper.

[2] As referenced on 802.3.

It is also observed that TCP responsiveness is improved by larger MTUs. Jumbo frames accelerate the congestion window increase by a factor of six compared to the standard MTU. Jumbo frames not only reduce I/O overhead on end-hosts, they also improve the responsiveness of TCP.

1.2 Multi streams

The network throughput is heavily dependent on the type of traffic that is flowing in the wire. An important point is the number of streams, also viewed as a socket, opened during a time slice. It involves creating and opening more than one socket and parallelizing the data transfer among these sockets. The link usage is better depending on how many streams are flowing. Even if the system is well tuned, it is not easy to achieve the wire speed using just a single stream.

The rule of thumb says that multi stream applications are preferred instead of an application that has just a connection and try to send all the data through it. Actually a lot of software, as application servers, allows setting the number of opened sockets, mainly when the application is trying to connect to a database.

As practical case, the Netperf tool provides a more accurate result when using multi stream mode with 8 actives streams.

1.3 Transmission queue size

The transmission queue is the buffer that holds packets that is scheduled to be sent to the card. Tuning the size of this buffer is necessary in order to avoid that packet descriptors are lost because of no available space in the memory.

Depending on the type of the network, the default 1000 packets value could not be enough and should be raised. Actually, a good number is around 3000 depending of the network characteristics.

1.4 SMP IRQ affinity

The main way that a CPU and a I/O device communicates is through interrupts. Interrupt means that when a device wants attention, it raises an interruption, and the CPU handles it, going to the device and checking

```
           CPU0  CPU1     CPU2    CPU3
 16:        832  1848     1483    2288  XICS  Level  IPI
 17:          0     0        0       0  XICS  Level  hvc_console
 18:          0     0        0       0  XICS  Level  RAS_EPOW
 53:         53     0  1006570       0  XICS  Level  eth0-TxRx-0
 54:         28     0        0  159907  XICS  Level  eth0-TxRx-1
 55:    2105871     0        0       0  XICS  Level  eth0-TxRx-2
 56:    1421839     0        0       0  XICS  Level  eth0-TxRx-3
 57:         13   888        0       0  XICS  Level  eth0-tx-0
 58:         13     0      888       0  XICS  Level  eth0-tx-1
 59:         13     0        0     888  XICS  Level  eth0-tx-2
 60:          4     0        0       0  XICS  Level  eth0:lsc
256:          1     0        0       0  XICS  Level  ehea_neq
261:          0     0        0       0  XICS  Level  eth1-aff
262:        119  5290        0       0  XICS  Level  eth1-queue0
273:       7506     0        0    1341  XICS  Level  ipr
```

Figure 1: Which CPU is handling which device IRQ

what are the devices needs. On an SMP system, the specific CPU handling your interruption is very important for performance.

Network devices usually has from one to a few interrupt line to communicate with the CPU. On multiple CPU system, each CPU call handles an interruption request. Usually the round robin algorithm is used to choose the CPU that will handle a specific interruption for a specific card. In order to check which CPU handled the device interruption, the pseudo-file `/proc/interrupts` will display all the interrupts lines, and which CPU handled the interruptions generated for each interrupt line. Figure 1 is an example of the content of `/proc/interrupts` pseudo-file.

In order to achieve the best performance, it is recommended that all the interruptions generated by a device queue is handled by the same CPU, instead of IRQ balancing. Although it is not expected, round robin IRQ distribution is not good for performance because when the interruption go to another fresh CPU, the new CPU probably will not have the interrupt handler function in the cache, and a long time will be required to get the properly interrupt handler from the main memory and run it. On the other hand, if the same CPU handles the same IRQ almost all the time, the IRQ handler function will unlikely leave the CPU cache, boosting the kernel performance.

In this manner, no IRQ balancing should be done for networking device interrupts, once it destroys performance. It is also important that TX affinity matches RX affinity, so the TX completion runs on the same CPU as RX. Don't mapping RX completion to RX completion causes some performance penalty dues cache misses.

Almost all current distros comes with a daemon that balance the IRQs, which is very undesirable, and should be disable. In order to disable the `irqbalance` tool, the following command should stop it.

```
# service irqbalance stop
```

If disabling this daemon is a very drastic change, then it is possible to disable `irqbalance` only for those CPUs that has an IRQ bound, and leave the IRQ balance enabled on all other CPUs. This can be done by editing the file `/etc/sysconfig/irqbalance`, and changing the `IRQBALANCE_BANNED_CPUS` option. So, all the CPUs that are bound to an interface queue, must be set as banned in the `irqbalance` configuration file.

Once the `irqbalance` daemon is disabled or configured, the next step is to configure which CPU will handle each device interruption.

In order to bind an interrupt line to a CPU, kernel 2.4 or superior provides scheme on the `/proc` interface which was created to allow the user to choose which

group of processors will handle a specific interrupt line. This configuration lives at `/proc/<IRQ number>/smp_affinity`, and can be changed any time on-the-fly. The content of the `smp_affinity` is a hexadecimal value that represents a group of CPU, as for example, `ff` representing all eight CPUs available. So, each field in the bit mask corresponds to a processor, and to manipulate it properly, a binary to hexadecimal transformation will be required.

Each digit in `ff` represents a group of four CPUs, with the rightmost group being the least significant. The letter `f` is the hexadecimal representation for the decimal number 15 and represent `1111` in binary, and each of the places in the binary representation corresponds to a CPU.

CPU	Binary	Hex
0	0001	0x1
1	0010	0x2
2	0100	0x4
3	1000	0x8

By combining these bit patterns (basically, just adding the Hex values), it is possible to a group of processors. For example, a representation of a group of two CPUs, for instance CPU0 (0x1) and CPU2 (0x4), is the sum of both individually, which means 0x1 + 0x4 = 0x5.

1.5 Taskset affinity

On a multi stream setup, it is possible to see some tasks that are transmitting and receiving the packets, and depending on the schedule policy, the task can migrate from one CPU to other from time to time. Since task migration is a huge penalty for performance, it is advised that a task is bound to one or few CPUs, as described above for IRQ. Since disabling the receive and sent task from being floating in all the CPUs, the cache misses rate is decreased, and the performance improved.

In order to bind a task to a CPU, the command `taskset` should be used as follows.

```
$ taskset -p 0x1 4767
pid 4767's current affinity mask: 1
pid 4767's new affinity mask: 1
```

1.6 Interrupt coalescence

Most modern NICs provide interruption moderation or interruption coalescing mechanisms to reduce the number of IRQs generated when receiving frames. As Ethernet frames arrive, the NIC saves them in memory, but the NIC will wait a receive delay before generating an interruption to indicate that one or more frames have been received. This moderation reduces the number of context switches made by the kernel to service the interruptions, but adds extra latency to frame reception.

Using interruption coalescence doesn't allow the system to suffer from IRQ storms generated during high traffic load, improving CPU efficiency if properly tuned for specific network traffic.

Enabling interruption coalescence could be done using the tool `ethtool` together with parameter `-C`. There are some modules that do not honor the `ethtool` method of changing the coalescence setting and implements its own method. In this case, where this option is specific to the device driver, using `modinfo` to discover the module parameter that enables it is the best option.

NAPI

"New API" (NAPI) is a feature that solves the same issue that Interrupt coalescence solved without hurting latency too much. NAPI was created in the linux kernel aiming to improve the performance of high-speed networking, and avoid interrupt storms. NAPI basically creates a mixture of interrupts and polling, called "adaptive interrupt coalescing." It means that in high traffic the device interruptions are disabled and packets are collected by polling, decreasing the system load. When the traffic is not high, then the interrupt scheme takes place, avoiding long latencies because of the pooling scheme. Also, NAPI-compliant drivers are able to drop packets in NIC (even before it reaches the kernel) when necessary, improving the system overall performance.

NAPI was first incorporated in the 2.6 kernel and was also backported to the 2.4.20 kernel. In general, running an NAPI enabled driver is a plus to get good performance numbers.

2 Offload features

Originally TCP was designed for unreliable low speed networks, but the networks changed a lot, link speed be-

comes more aggressive and quality of service is now a strict requirement for a lot of applications. Although these huge changes is still happening, the TCP protocol is almost the same as it was designed. As the link speed grows, more CPU cycle is required to be able to handle all the traffic. Even the more current CPUs waste a considerable amount of cycle in order to process all the CPU communication.

A generally accepted rule of thumb[3] is that one hertz of CPU is required to send or receive one bit of TCP/IP[2]. For example a five gigabit per second of traffic in a network requires around five GHz of CPU for handling this traffic. This implies that two entire cores of a 2.5 GHz multi-core processor will be required to handle the TCP/IP processing associated with five gigabit per second of TCP/IP traffic. Since Ethernet is bidirectional, it means that it is possible to send and receive 10 Gbit/s. Following the 1 Hz per bit rule, it would need around eight 2.5 GHz cores to drive a 10 Gbit/s Ethernet network link.

Looking into this scenario, the network manufacturers started to offload a lot of repetitive tasks to the network card itself. It leaves the CPU not so busy executing the usual networking tasks, as computing the checksum and copying memory around.

The manufactures classify offload features in two types, those that are stateful and those that are stateless, where a state references the TCP state. This section will cover only the stateless features, and the stateful feature will be briefly described in Section 5.8.

Almost all offload features are configured using the `ethtool` tool. In order to check which features are set, run `ethtool` using the `-k` parameter, as follows:

```
# ethtool -k eth2

Offload parameters for eth2:
rx-checksumming: off
tx-checksumming: off
scatter-gather: off
tcp segmentation offload: off
udp fragmentation offload: off
generic segmentation offload: off
```

[3]This general rule of thumb was first stated by PC Magazine around 1995, and is still used as an approximation nowadays.

In order to enable or disable any feature, the parameter `-K` should be used with `on` to enable the feature, and `off` to disable it. The following example enables TX checksum feature for interface `eth2`.

```
# ethtool -K eth2 tx on
```

2.1 RX Checksum

The TCP RX checksum offload option enables the network adapter to compute the TCP checksum when a packet is received, and only send it to the kernel if the checksum is correct. This feature saves the host CPU from having to compute the checksum, once the card guaranteed that the checksum is correct.

Enabling RX checksum offload can be done using `ethtool -K ethX rx on`[4]

Note that if receive checksum offload is disabled, then it is not possible to enable large receive offload (LRO).

2.2 TX Checksum

TX Checksum works almost as RX checksum, but it asks the card to compute the segment checksum before sending it. When enabling this option, the kernel just fill a random value in the checksum field of the TCP header and trust that the network adapter will fill it correctly, before putting the packet into the wire.

In order to enable TX checksum offload, the command `ethtool -K ethX tx on` should be run.

When the TX and RX offload are enabled, the amount of CPU saved depends on the packet size. Small packets have little or no savings with this option, while large packets have larger savings. On the PCI-X gigabit adapters, it is possible to save around five percent in the CPU utilization when using a 1500 MTU. On the other hand, when using a 9000 MTU the savings is approximately around 15%.

It is important to note that disabling transmission checksum offload also disables scatter and gather offload since they are dependent.

[4]Replace ethX to the target interface

2.3 Scatter and Gather

Scatter and Gather, also known as Vectored I/O, is a concept that was primarily used in hard disks[3]. It basically enhance large I/O request performance, if supported by the hardware. Scatter reading is the ability to deliver data blocks stored at consecutive hardware address to non-consecutive memory addresses. Gather writing is the ability to deliver blocks of data stored at non-consecutive memory addresses to consecutively addressed hardware blocks.

One of the constraints that happens to DMA is that the physical memory buffer should be contiguous in order to receive the data. On the other hand, a device that supports scatter and gather capability allows the kernel to allocate smaller buffers at various memory locations for DMA. Allocating smaller buffers is much easier and faster than finding for a huge buffer to place the packet.

When scatter and Gather is enable, it is also possible to do a concept called *page flip*. This basically allows the transport and other headers to be separated from the payload. Splitting header from payload is useful for copy avoidance because a virtual memory system may map the payload to an possible application buffer, only manipulating the virtual memory page to point to the payload, instead of copying the payload from one place to another.

The advantage of Scatter and Gather is to reduce overhead allocating memory and copying data, also as having a better memory footprint.

In order to enable Scatter and gather, the command `ethtool -K ethX sg on` should be run.

2.4 TCP Segmentation Offload

TCP segmentation offload (TSO), also called Large Segment Offload (LSO), is feature used to reduce the CPU overhead when running TCP/IP. TSO is a network card feature designed to break down large groups of data sent over a network into smaller segments that pass through all the network elements between the source and destination. This type of offload relies on the network interface controller to segment the data and then add the TCP, IP and data link layer protocol headers to each segment.

The performance improvement comes from the fact that the upper layers deliver a huge packet, as 64K, to the card and the card splits the this packet in small frames which honor the MTU size.

Some studies suggests that enabling TSO saves around 10% in the CPU when using a 1500 MTU.

In order too enable TCP segmentation offload, the command `ethtool -K ethX tso on` should be run.

2.5 Large Receive Offload

Large Receive Offload (LRO) is a technique that increases inbound throughput of high-bandwidth network connections by reducing CPU overhead. It works by aggregating, in the card, multiple incoming packets from a single stream into a larger buffer before they are passed to the network stack, thus reducing the number of packets that have to be processed, and all headers overhead. This concept is basically the opposite of TSO. LRO combines multiple Ethernet frames into a single receive, thereby decreasing CPU utilization when the system is receiving multiple small packets.

There are two types of LRO, one that are just a change in the network structure that is usually enabled by default in new drivers. This one aggregates frames in the device driver rather than in the card itself. The other one is a hardware feature that aggregate the frames into the card itself and provides much better results. The last one is specific for each device driver and it is usually enabled using a module parameter, as `lro_enable` for s2io driver.

2.6 Generic Segmentation Offload

After TSO was implemented, It was observed that a lot of the savings in TSO come from traversing the kernel networking stack once instead than many times for each packet. Since this concept was not dependent of the hardware support, the Generic Segmentation Offload (GSO) was implemented to postpone the segmentation as late as possible. Once this is a general concept, it can be applied to other protocols such as version 6 of TCP, UDP, or even DCCP.

GSO like TSO is only effective if the MTU is significantly less than the maximum value of 64K.

In order to enable generic segmentation offload, the command `ethtool -K ethX gso on` should be run.

3 Kernel settings

The Linux kernel and the distributions that package it typically provides very conservative defaults to certain network kernel settings that affect networking parameters. These settings can be set via the `/proc` filesystem or using the `sysctl` command. Using `sysctl` is usually better, as it reads the contents of `/etc/sysctl.conf` or any other selected chosen config script, which allows to keep the setting even after the machine restart. Hereafter only the modifications using the sysctl will be covered.

In order to change a simple configuration, a command as `sysctl -w net.core.rmem_max=16777216` should be run. In this case, the parameter `-w` want to set the value `16777216` into the variable `net.core.rmem_max`. Note that this is a temporary setting, and it will be lost after the system is restart. However most of the configuration should hold after a system restart, and in this case a modification in the file `/etc/sysctl.conf` will be necessary.

Also, it is possible to create a file with a lot of configuration, which can be called using the command that follows.

```
# sysctl -p /script/path
```

It's also possible to see all the parameters related to network using the following parameter.

```
# sysctl -a | grep net
```

On other hand, checking just a option can be done as follows.

```
# sysctl net.ipv4.tcp_window_scaling
```

3.1 TCP Windows Scaling

The TCP/IP protocol has a header field called *window*. This field specifies how much data the system which sent the packet is willing and able to receive from the other end. In other words, it is the buffer space required at sender and receiver to save the unacknowledged data that TCP must handle in order to keep the pipeline full.

In this way, the throughput of a communication is limited by two windows: congestion window and receive window. The first one tries not to exceed the capacity of the network (congestion control) and the second one tries not to exceed the capacity of the receiver to process data (flow control). The receiver may be overwhelmed by data if for example it is very busy (such as a Web server). As an example, each TCP segment contains the current value of the receive window. So, if the sender receives an *ACK* which acknowledge byte 4000 and specifies a receive window of 10000 (bytes), the sender will not send packets after byte 14000, even if the congestion window allows it.

Since TCP/IP was designed in the very far past, the window field is only 16 bits wide, therefore the maximum window size that can be used is 64KB. As it's known, 64KB is not even close to what is required by 10Gbit/s networks. The solution found by the protocol engineer was windows scaling describer in RFC 1323, where it creates a new TCP option field which left-shift the current window value to be able to represent a much larger value. This option defines an implicit scale factor, which is used to multiply the window size value found in a regular TCP header to obtain the true window size.

Most kernels enables this option by default, as could be seen running the command `sysctl net.ipv4.tcp_window_scaling`.

3.2 TCP Timestamp

TCP timestamp is a feature also described by RFC 1323 that allow a more precise round trip time measurement. Although timestamp is a nice feature, it includes an eight bytes to the TCP header, and this overhead affects the throughput and CPU usage. In order to reach the wire speed, it is advised to disable the timestamp feature, setting running `sysctl -w net.ipv4.tcp_timestamps=0`

3.3 TCP fin timeout setting

When TCP is finishing a connection, it gets into a state called **FIN-WAIT-2**, which still wasting some memory resources. If the client side doesn't finish the connection, the system needs to wait a timeout to close it by itself. On Linux, it is possible to determine the time that must elapse before TCP/IP can release the resources for a closed connection and reuse them. By reducing the value of this entry, TCP/IP can release closed connections faster, making more resources available for new connections. So, when running on a server that has a big

amount of closed socket, this adjust saves some system resources. In order to adjust this value, the parameter `net.ipv4.tcp_fin_timeout` should be changed to a smaller number than the default. Around 15 and 30 seconds seem to be a good value for moderate servers.

3.4 TCP SACK and Nagle

TCP Sack is a TCP/IP feature that means TCP Selective Acknowledgement and was described by RFC 2018, It was aimed for networks that have big windows and usually lose packets. It consists in sending explicit **ACK** for the segments of the stream has arrived correctly, in a non-contiguous manner, instead of the traditional TCP/IP algorithm that implements cumulative acknowledgement, which just acknowledges contiguous segments.

As most of the 10 Gbit/s network are reliable and usually losing packets is not the common case, disabling SACK will improve the network throughput. It is important to note that if the network is unreliable, and usually lose packets, then this option should be turned on.

In order to disable it, set `0` into the `net.ipv4.tcp_sack` parameter.

On the other side, Nagle algorithm should be turned on. Nagle is a TCP/IP feature described by **RFC 896** and works by grouping small outgoing messages into a bigger segment, increasing bandwidth and also latency. Usually Nagle algorithm is enabled on standard sockets, and to disable it, the socket should set `TCP_NODELAY` in its option.

3.5 Memory options

Linux supports global setting to limit the amount of system memory that can be used by any one TCP connection. It also supports separate per connection send and receive buffer limits that can be tuned by the system administrator.

After kernel 2.6.17, buffers are calculated automatically and usually works very well for the general case. Therefore, unless very high RTT, loss or performance requirement is present, buffer settings may not need to be tuned. If kernel memory auto tuning is not present (Linux 2.4 before 2.4.27 or Linux 2.6 before 2.6.7), then replacing the kernel is highly recommended.

In order to ensure that the auto tuning is present and it is properly set, the parameter `net.ipv4.tcp_moderate_rcvbuf` should be set to 1.

Once the auto tuning feature is on, the limits of memory used by the auto tuner should be adjusted, since most of the network memory options have the value that need to be set. The value is an array of 3 values that represents the minimum, the initial and maximum buffer size as exemplified above. These values are used to set the bounds on auto tuning and balance memory usage. Note that these are controls on the actual memory usage (not just TCP window size) and include memory used by the socket data structures as well as memory wasted by short packets in large buffers.

```
net.ipv4.tcp_rmem = 4096 87380 3526656
```

There are basically three settings that define how TCP memory is managed that need a special care and will be described hereafter. They are `net.ipv4.tcp_rmem`, which define the size (in bytes) of receive buffer used by TCP sockets, `net.ipv4.tcp_wmem` which define the amount of memory (in bytes) reserved for send buffers, and the `net.ipv4.tcp_mem`, which define the total TCP buffer-space allocatable in units of page (usually 4k on x86 and 64k on PPC). So, tuning these settings depends on the type of the Ethernet traffic and the amount of memory that the server has.

All of those value should be changed so that the maximum size is bigger than the BDP, otherwise packets can be dropped because of buffer overflow. Also, setting `net.ipv4.tcp_mem` to be twice the delay bandwidth delay product of the path is a good idea.

Secondarily, there are a lot of configurations that need to be tuned depending on the how network is working, and if the performance requirement was met. The important ones are `net.core.rmem_max`, `net.core.wmem_max`, `net.core.rmem_default`, `net.core.wmem_default`, and `net.core.optmem_max`. The kernel file `Documentation/network/ip-sysctl.txt` has a description for almost all of these parameters and should be consulted whenever necessary.

Another useful setting that deserves a check is `net.core.netdev_max_backlog`. This parameter defines the maximum number of packets queued on the

input side, when the interface receives packets faster than kernel can process them, which is usual on 10 Gbit/s network interface. This option is opposite for the `txqueuelen`, that define the length of the queue in the transmission side. As the value for the maximum backlog depends on the speed of the system, it should be fine tuned. Usually a value near 300000 packets is a good start point.

4 Bus

One important aspect of tuning a system is assuring that the system is able to support the desired needs, as throughput and latency. In order to ensure that a system is able to run on the desired speed, a brief overview of common the bus subsystem will be described. Since 10 Gbit/s network cards are only available on PCI extended (PCI-X) and PCI Express (PCI-E), this article will focus on these buses.

PCI is the most common bus for network cards, and it is very known for its inefficiency when transferring small bursts of data across the PCI bus to the network interface card, although its efficiency improves as the data burst size increases. When using the standard TCP protocol, it is usual to transfer a large number of small packets as acknowledgement and as these are typically built on the host CPU and transmitted across the PCI bus and out the network physical interface, this impacts the host overall throughput.

In order to compare what is required to run a 10 Gbit/s card on full speed, some bus performance will be displayed. A typical PCI bus running at 66 MHz, has an effective bandwidth of around 350 MByte/s.[5] The more recent PCI-X bus runs at 133 MHz and has an effective bus bandwidth of around 800 MByte/s. In order to fill a 10 Gigabit Ethernet link running in full-duplex, the order of 1.25GByte/s of bandwidth is required in each direction, or a total of 2.5 GByte/s, more than three times the PCI-X implementations, and nearly seven times the bandwidth of a legacy PCI bus. So, in order to get the best performance of the card, a PCI-E network card is highly recommended.

Also, usually most of the network traffic is written or read from the disk. Since the disk controller usually shares the same bus with network cards, a special consideration should be taken in order to avoid bus stress.

[5] PCI Encoding overhead is two bits in 10.

4.1 PCI Express bandwidth

PCI Express version 1.0 raw signaling is 250 MByte/s lane, while PCI Express version 2.0 doubles the bus standard's bandwidth from 250 MByte/s to 500MByte/s, meaning a x32 connector can transfer data at up to 16 GByte/s

PCI Express version 3.0 that is already in development and is scheduled to be released in 2010, will be able to deliver 1 GByte/s per lane.

4.2 Message-Signalled Interrupt

One important PCI-E advantage is that interrupts are transferred in-line instead of out-of-band. This feature is called Message-signalled Interrupt (MSI). MSI enables a better interrupt handling since it allows multiple queueable interrupts.

Using MSI can lower interrupt latency, by giving every kind of interruption its own handler. When the kernel receives a message, it will directly call the interrupt handler for that service routine associated with the address. For example, there are many types of interrupts, one for link status change, one for packet transmitted status, one for packet received, etc. On the legacy interrupt system, the kernel is required to read a card register in order to discover why the card is interrupting the CPU, and to call the proper interrupt handler. This long path causes a delay that doesn't happen when the system is using MSI interrupts.

MSI-X is an extension to MSI to enable support for more vectors and other advantages. MSI can support at most 32 vectors while MSI-X can up to 2048. Also, when using MSI, each interrupt must go to the same address and the data written to that address are consecutive. On MSI-X, it allows each interrupt to be separately targeted to individual processors, causing a great benefit, as a high cache hit rate, and improving the quality of service.

In order to ensure that MSI or MSI-X are enabled on a system, the `lspci` command should be used to display the PCI device capabilities. The capabilities that describe these features and should be enabled are `Message Signalled Interrupts` or `MSI-X` as described by Figure 2.

```
# lspci | grep "10 Giga"
0002:01:00.0 Ethernet controller: Intel Corporation 82598EB 10 Gigabit AF
                                                    Network Connection (rev 01)

# lspci -vv -s 0002:01:00.0  | grep Message
Capabilities: [50] Message Signalled Interrupts: 64bit+ Queue=0/0 Enable+
```

Figure 2: lspci output

4.3 Memory bus

Memory bus might not be as fast as required to run a full duplex 10 Gbit/s network traffic. One important point is ensure that the system memory bandwidth is able to support the traffic requirement. In order to prove that the memory bandwidth is enough for the desired throughput, a practical test using Netperf can be run. This test displays the maximum throughput that the system memory can support.

Running a Netperf test against the *localhost* is enough to discover the maximum performance the machine can support. The test is simple, and consist of starting the `netserver` application on the machine and running the test, binding the client and the server in the same processor, as shown by Figure 3. If the shown bandwidth is more than the required throughput, then the memory bus is enough to allow a high load traffic. This example uses the `-c` and `-C` options to enable CPU utilization reporting and shows the asymmetry in CPU loading.

```
# netperf -T0,0 -C -c
              Utilization
              Send     Recv
...  Throughput  local    remote  ...
     10^6bits/s  % S      % S

      13813.70   88.85    88.85
```

Figure 3: Test for memory bandwidth

4.4 TCP Congestion protocols

It is a very important function of TCP to properly match the transmission rate of the sender and receiver over the network condition[4]. It is important for the transmission to run at high enough rate in order to achieve good performance and also to protect against congestion and packet losses. Congestion occurs when the traffic offered to a communication network exceeds its available transmission capacity.

This is implemented in TCP using a concept called windows. The window size advertised by the receiver tells the sender how much data, starting from the current position in the TCP data byte stream can be sent without waiting for further acknowledgements. As data is sent by the sender and then acknowledged by the receiver, the window slides forward to cover more data in the byte stream. This is usually called sliding window.

Some usual congestion avoidance algorithm has a property to handle with the window size called "additive increase and multiplicative decrease" (AIMD). This property is proven to not be adequate for exploring large bandwidth delay product network, once general TCP performance is dependent of the Congestion control algorithm[5]. And the congestion control algorithm is basically dependent of network bandwidth, round trip time and packet loss rate. Therefore, depending on the characteristics of your network, an algorithm fits better than another.

On Linux it is possible to change the congestion avoidance algorithm on the fly, without even restarting your connections. To do so, a file called `/proc/sys/net/ipv4/tcp_available_congestion_control` lists all the algorithms available to the system, as follows:

```
# cat /proc/sys/net/ipv4/tcp_available_
congestion_control
cubic reno
```

Also, it is possible to see what algorithm the system is currently using, looking into file `/proc/sys/net/ipv4/tcp_congestion_control`. To set a new algorithm, just echo one of those available algorithm name into this file, as follows:

```
# echo cubic > /proc/sys/net/ipv4/tcp
_congestion_control
```

This article will outline some of the most common TCP/IP collision avoidance algorithms.

4.5 RENO

TCP Reno was created to improve and old algorithm called Tahoe[6]. Its major change was the way in which it reacts when detect a loss through duplicate acknowledgements. The idea is that the only way for a loss to be detected via a timeout and not via the receipt of a dupack is when the flow of packets and `acks` has completely stopped. Reno is reactive, rather than proactive, in this respect. Reno needs to create losses to find the available bandwidth of the connection.

Reno is probably the most common algorithm and was used as default by Linux until kernel 2.6.19.

When using Reno on fast networks, it can eventually underutilize the bandwidth and cause some variation in the bandwidth usage from time to time[7]. For example, in slow start, window increases exponentially, but may not be enough for this scenario. On a scenario when using 10Gbit/s network with a 200ms RTT, 1460B payload and assuming no loss, and initial time to fill pipe is around 18 round trips, which result in a delay of 3.6 seconds. If one packet is lost during this phase, this time can be much higher.

Also, to sustain high data rates, low loss probabilities are required. If the connection loose one packet then the algorithm gets into AIMD, which can cause severe cut to the window size, degrading the general network performance.

On the other hand, Reno in general is more TCP-friendly than FAST and CUBIC.

4.6 CUBIC

CUBIC is an evolution of BIC. BIC is algorithm that has a pretty good scalability, fairness, and stability during the current high speed environments, but the BIC's growth function can still be too aggressive for TCP, especially under short RTT or low speed networks, so CUBIC was created in order to simply its window control and improve its TCP-friendliness. Also, CUBIC implements an optimized congestion control algorithm for high speed networks with high latency

In general, CUBIC is on the best algorithm to get the best throughput and TCP-fairness. That is why it is implemented and used by default in Linux kernels since version 2.6.19.

4.7 FAST

FAST (FAST AQM Scalable TCP) was an algorithm created with high-speed and long-distance links in mind.

A FAST TCP flow seeks to maintain a constant number of packets in queues throughout the network. FAST is a delay-based algorithm that try to maintain a constant number of packets in queue, and a constant window size, avoiding the oscillations inherent in loss-based algorithms, as Reno. Also, it also detects congestion earlier than loss-based algorithms. Based on it, if it shares a network with loss-based "protocol," the FAST algorithms tend to be less aggressive. In this way, FAST has the better Friendliness than CUBIC and Reno.

5 Benchmark Tools

There are vast number of tools that can be used to benchmark the network performance. In this section, only `Netperf`, `Mpstat` and `Pktgen` will be covered. These are tools that generate specifics kind of traffic and then shows how fast the network transmitted it.

5.1 Netperf

Netperf[6] is a tool to measure various aspects of networking performance. It is primarily focus is on data transfer using either TCP or UDP. Netperf has also a vast number of tests like stream of data and file transfer among others.

[6] `http://www.netperf.org`

```
# netperf -t TCP_STREAM -H 192.168.200.2 -l 10
TCP STREAM TEST from 0.0.0.0 port 0 AF_INET to 192.168.200.2 port 0 AF_INET
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

 87380  16384  16384    10.00    7859.85
```

Figure 4: Netperf running for 10 seconds

```
# netperf -H 192.168.200.2  -t TCP_RR -l 10
TCP REQUEST/RESPONSE TEST from 0.0.0.0 port 0 AF_INET to
                                                  192.168.200.2 port 0 AF_INET
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  87380  1        1       10.00    42393.23
```

Figure 5: Netperf running with TCP_RR

It is easy to benchmark with Netperf. An installation of the tool is required in the client and server. On the server side, a daemon called `netserver` should be started before the tests begin, so that the Netperf client could connect to it. Netperf usually is listening on port 12865.

In order to ensure that the Netperf setup is working properly, a simple test as `netperf -H <hostname>` is enough to prove that the communication is good between the client and the server.

Netperf tests are easier when considering to use the set of scripts files provided with the Netperf distribution. These scripts are usually located at `/usr/share/doc/netperf/examples`. These scripts has a set of features and configuration that is used on those test cases. Since these scripts are heavily configurable, it is not required to read the entire Netperf manual in order to run more complex tests.

Netperf supports a lot of tests cases, but in this paper only TCP/UDP streams and transaction will be covered.

Streams

On streams test, Netperf supports a vast number of testcases, but only three are widely used, they are `TCP_STREAM`, `TCP_MAERTS`[7] and `UDP_STREAM`. The difference between `TCP_STREAM` and `TCP_MAERTS`, is that on first, the traffic flows from the client to the server, and on `TCP_MAERTS`, the traffic flows from the server to the client. Thus running both `TCP_STREAM` and `TCP_MAERTS` in parallel generate a full-duplex test.

Figure 4 shows a simple example of `TCP_STREAM` test case running against IP `192.168.200.2` for ten seconds.

Transactions

Transaction is another area to investigate and tune in order to improve the quality of the network, and to do so, Netperf provides a testcase to measure request/response benchmark. A transaction is defined as a single reply for a single request. In this case, request/response performance is quoted as "transactions/s" for a given request and response size.

[7] MAERTS is an anagram of STREAM

```
#!/bin/bash

NUMBER=8
TMPFILE=`mktemp`
PORT=12895

for i in `seq $NUMBER`
do
    netperf -H $PEER -p $PORT -t TCP_MAERTS -P 0  -c -l $DURATION
            -- -m 32K -M 32K -s 256K -S 256K >> $TMPFILE &
    netperf -H $PEER -p $PORT -t TCP_STREAM -P 0  -c -l $DURATION
            -- -m 32K -M 32K -s 256K -S 256K >> $TMPFILE &
done

sleep $DURATION

echo -n "Total result: "

cat $TMPFILE |  awk '{sum += $5} END{print sum}'
```

Figure 6: Script to run Netperf using multistream

In order to measure transactions performance, the test type should be `TCP_RR` or `UDP_RR`, as shown by Figure 5.

In order to do some complex benchmark using transaction, a script called `tcp_rr_script` is also available and can be configured to accomplish the user's needed.

Since Netperf uses common sockets to transmit the data, it is possible to set the socket parameters used by a test. Also, it is possible to configure some details of the specific test case that is being used. To do so, the parameter `-- -h` should be appended in the end of the Netperf line, as `netperf -t TCP_RR -- -h` for the request performance test case.

5.2 Netperf multi stream

Netperf supports multi stream in version four only, which may not be available to some distro distro. Netperf version two is the mostly distributed and used version. The script described in Figure 6 simulates a multi stream Netperf, and is widely used in the network performance community.

5.3 Pktgen

Pktgen is a high performance testing tool, which is included in the Linux kernel as a module. Pktgen is currently one of the best tools available to test the transmission flow of network device driver. It can also be used to generate ordinary packets to test other network devices. Especially of interest is the use of pktgen to test routers or bridges which often also use the Linux network stack. Pktgen can generate high bandwidth traffic specially because it is a kernel space application. Hence it is useful to guarantee that the network will work properly on high load traffic.

It is important to note that pktgen is not a Netperf replacement, pktgen cannot execute any TCP/IP test. It is just used to help Netperf in some specific transmission test cases.

Pktgen can generate highly traffic rates due a very nice kernel trick. The trick is based on cloning ready-to-transmit packets and transmitting the original packet and the cloned ones, avoiding the packet construction phase. This idea basically bypass almost all the protocol kernel layers, generating a high bandwidth traffic without burdening the CPU.

```
Params: count 10000000  min_pkt_size: 60  max_pkt_size: 60
     frags: 0  delay: 0  clone_skb: 1000000  ifname: eth3
     flows: 0 flowlen: 0
     dst_min: 10.1.1.2  dst_max:
     src_min:   src_max:
     src_mac: 00:C0:DD:12:05:4D  dst_mac: 00:C0:DD:12:05:75
     udp_src_min: 9  udp_src_max: 9  udp_dst_min: 9
     udp_dst_max:
     src_mac_count: 0  dst_mac_count: 0
     Flags:
Current:
     pkts-sofar: 10000000  errors: 16723
     started: 1240429745811343us
              stopped: 1240429768195855us idle: 1954836us
     seq_num: 10000011  cur_dst_mac_offset: 0
              cur_src_mac_offset: 0
     cur_saddr: 0xa010101  cur_daddr: 0xa010102
     cur_udp_dst: 9  cur_udp_src: 9
     flows: 0
Result: OK: 22384512(c20429676+d1954836) usec,
10000000 (60byte,0frags)
  446737pps 214Mb/sec (214433760bps) errors: 16723
```

Figure 7: pktgen output example

Figure 7 shows a typical example of the output after running a test with pktgen.

It is easy but not trivial to run a pktgen example and the steps will not be covered in this article. For more information, the file `Documentation/networking/pktgen.txt` at the kernel code is a good source.

5.4 Mpstat

Mpstat is a tool provided by `sysstat`[8] package that is probably the most used tool to verify the processors load during a network performance test. Mpstat monitors SMP CPUs usage and it is a very helpful tool to discover if a CPU is overloaded and which process is burdening each CPU.

Mpstat can also display statistics with the amount of IRQs that were raised in a time frame and which CPU handled them, as it is shown on Figure 8. This is a very useful test to guarantee that the IRQ affinity, discussed earlier, is working properly. In order to discover which IRQ number represents which device line, the file `/proc/interrupts` contains the map between the IRQ number and the device that holds the IRQ line.

[8] `http://pagesperso-orange.fr/sebastien.godard/`

5.5 Other technologies

This section intends to briefly describe some other technologies that help to get a better network utilization.

5.6 I/OAT

Intel I/OAT is a feature that improves network application responsiveness by moving network data more efficiently through Dual-Core and Quad-Core Intel Xeon processor-based servers when using Intel network cards. This feature improves the overall network speed.

In order to enable the Intel I/OAT network accelerations the driver named `ioatdma` should be loaded in the kernel.

```
12:12:25  CPU  18/s  59/s  182/s  216/s  338/s  471/s  BAD/s
12:12:25    0  0.09  0.88   0.01   0.00   0.00   0.00   2.23
12:12:25    1  0.13  0.88   0.00   0.00   0.00   0.00   0.00
12:12:25    2  0.15  0.88   0.01   0.00   0.00   0.00   0.00
12:12:25    3  0.19  0.88   0.00   0.00   0.00   0.00   0.00
12:12:25    4  0.10  0.88   0.01   0.00   0.00   0.00   0.00
12:12:25    5  0.13  0.88   0.00   0.00   0.00   0.00   0.00
12:12:25    6  0.08  0.88   0.01   0.00   0.00   0.00   0.00
12:12:25    7  0.20  0.88   0.00   0.00   0.00   0.00   0.00
```

Figure 8: Mpstat output

```
modprobe ioatdma
```

It is not recommended to remove the `ioatdma` module after it was loaded, once TCP holds a reference to the `ioatdma` driver when offloading receive traffic.

5.7 Remote DMA

Remote Direct Memory Access (RDMA) is an extension of DMA where it allows data to move, bypassing the kernel, from the memory of one computer into another computer's memory. Using RDMA permits high-throughput, low-latency networking, which is a great improvement in many areas. The performance is visible mainly when the communication happens between two near computers, where the RTT is small.

For security reasons, it is undesirable to allow the remote card to read or write arbitrary memory on the server. So RDMA scheme prevents any unauthorized memory accesses. In this case, the remote card is only allowed to read and write into buffers that the receiver has explicitly identified to the NIC as valid RDMA targets. This process is called *registration*.

On Linux there is a project called `OpenRDMA` that provides an implementation of RDMA service layers. RDMA layer is already implemented by common applications such as NFS. NFS over RDMA is already being used and the performance was proved to be much better than the standard NFS.

5.8 TCP Offload Engine

TCP Offload Engine (TOE) is a feature that most network cards support in order to offload the TCP engine to the card. It means that the card has the TCP states for an established connection. TOE usually help the kernel, doing trivial steps in the card as fragmentation, acknowledges, etc.

This feature usually improve the CPU usage and reduces the PCI traffic, but creates issues that are hard to manage, mainly when talking about security aspects and proprietary implementations. Based on these facts, Linux doesn't support TOE by default, and a vendor patch should be applied in the network driver in order to enable this feature.

6 References

[1] D. Newman, *RFC2647—Benchmarking Terminology for Firewall Performance*, `http://www.faqs.org/rfcs/rfc2647.html`

[2] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, Greg J. Regnier, *TCP Performance Re-Visited* `http://www.cs.duke.edu/~jaidev/papers/ispass03.pdf`

[3] Amber D. Huffman, Knut S. Grimsurd, *Method and apparatus for reducing the disk drive data transfer interrupt service latency penaltyr* US Patent 6640274

[4] Van Jacobson, Michael J. Karels, *Congestion Avoidance and Control*, `http://ee.lbl.gov/papers/congavoid.pdf`

[5] Chunmei Liu, Eytan Modiano, *On the performance of additive increase multiplicative decrease (AIMD) protocols in hybrid space-terrestrial networks*, `http://portal.acm.org/citation.cfm?id=1071427`

[6] Kevin Fall, Sally Floyd, *Comparisons of Tahoe, Reno, and Sack TCP* `http://www.icir.org/floyd/papers/sacks.pdf`

[7] Jeonghoon Mo, Richard J. La, Venkat Anantharam, and Jean Walrand *Analysis and Comparison of TCP Reno and Vegas*, `http://netlab.caltech.edu/FAST/references/Mo_comparisonwithTCPReno.pdf`

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*