First INFN International School on Architectures, tools and methodologies for developing efficient large scale scientific computing applications

Ce.U.B. – Bertinoro – Italy, 12 – 17 October 2009

Domenico Galli

Domenico Galli Digitally signed by Domenico Galli DN: c=IT, o=INFN, ou=Personal Certificate, l=Bologna, cn=Domenico Galli Date: 2009.10.15 09:08:38 +02'00'

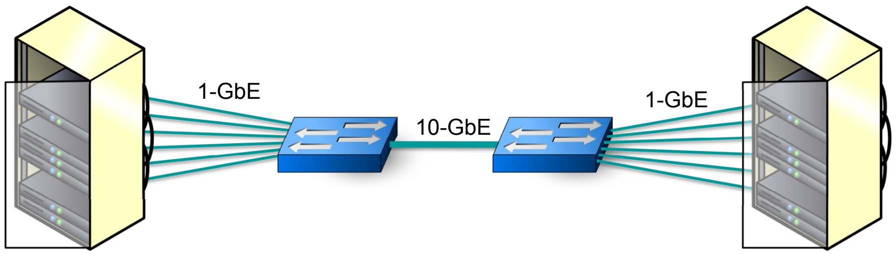
High Throughput Data Transmission Through Network Links

Outline

- **Need** of **High-Speed Links** in **HEP** applications:
  - 2 Use Cases.
- High speed data-link **technologies** in **HEP**:
  - Commodity links;
    - 10 Gb/s links.
- **Bottlenecks** in moving data through High-Speed Links.
- **Optimization**: Network **workload sharing** among **CPU cores**:
  - The Linux network layer:
    - Transmission and reception.
  - Process-to-CPU affinity;
  - IRQ-to CPU affinity;
- **Performances** of transmission through **10 Gb/s Ethernet**:
  - **UDP** transfer;
  - **TCP** transfer:
    - Nagle's algorithm;
    - Zero copy;
    - TCP hardware offload.

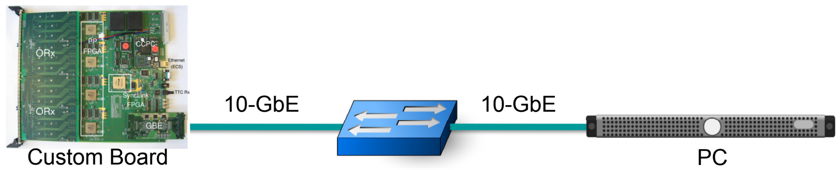
High Speed Network Links

- **Fastest available** network link technology in the market (e.g. 10-GbE at present) usually employed in LAN **backbones**:
  - Connecting **network devices** together:
    - E.g.: connecting together network switches in a LAN.
  - **Data flow** managed by Switch **Firmware**.
    - Switch manufacturer will care avoiding bottlenecks;
    - We only need to test the device…
- Front-end (PC, custom electronics) usually connected to lower speed devices.

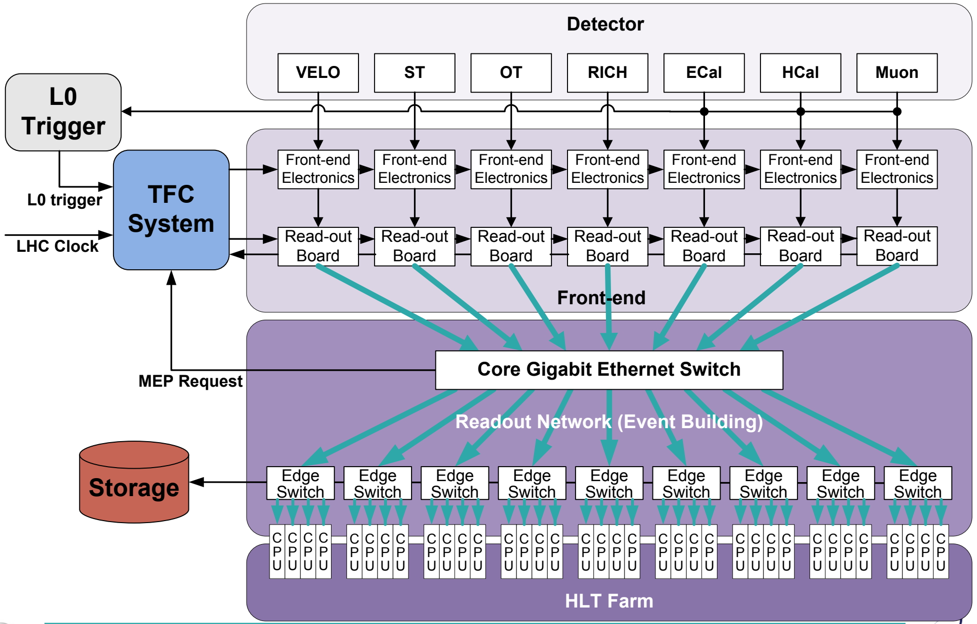


Front-end Access to High Speed Network

- **HEP applications** sometimes need High speed network links **directly connected to the front-end**:
  - PCs;
  - Custom electronic boards.
- **Data Flow** managed by **OS** or **FPGA software**.
  - Need to check **bottlenecks** which could limit the throughput.
- Use case 1: **On-line data path**:
  - Data Acquisition – Event Building – High Level Trigger.
- Use case 2: **Network Distributed Storage**:
  - Offline computing centers (Tier-1).

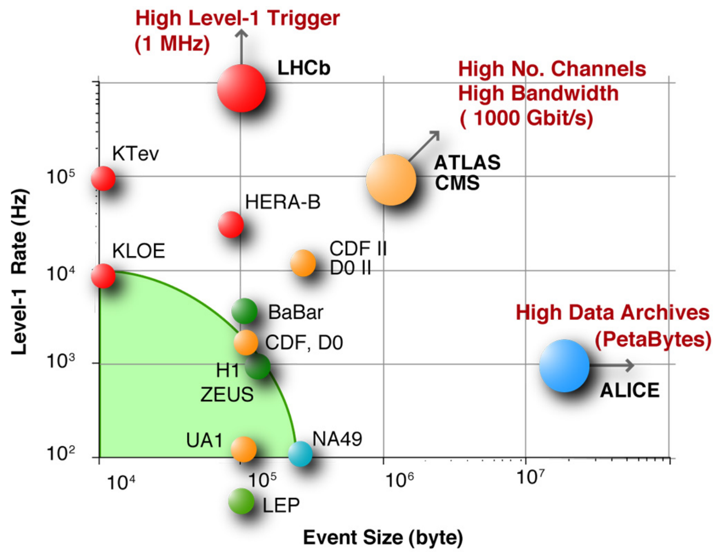


Use case 1: The On-Line Data Path



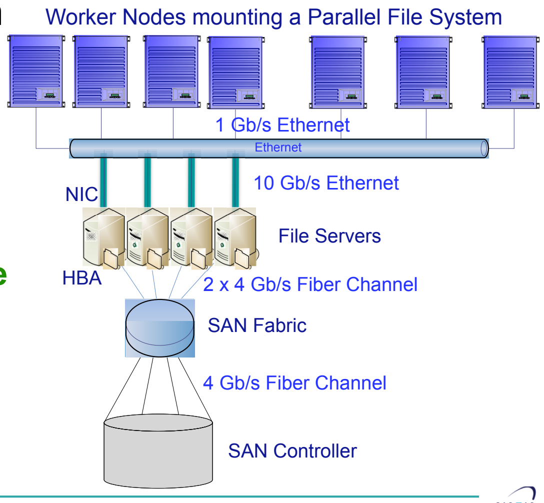
Use case 1: The On-Line Data Path (II)

- Trend in data packet rate and size.



Use Case 2: Network Storage in a SAN

- **File servers** in a **Storage Area Network** (SAN) which exports data to client nodes via Ethernet.
- Common situation in case of large computing farms:
  - Computing nodes access the mass storage through a pool of **Parallel File System** disk-servers:
    - E.g.: GPFS or Lustre.



High Speed Data Link Technology

- **Trend toward COTS technologies**:
  - HERA-B:
    - **Shark link** (proprietary, by Analog Devices) until level 2, than **Fast Ethernet**.
  - BaBar:
    - **Fast Ethernet**.
  - DØ:
    - **Fast Ethernet** / **Gigabit Ethernet**.
  - CDF:
    - **ATM** / **SCRAMnet** (proprietary, by Systran, low latency replicated non-coherent shared memory network).
  - CMS:
    - **Myrinet** (proprietary, Myricom) / **Gigabit Ethernet**.
  - Atlas / LHCb / Alice:
    - **Gigabit Ethernet**.
  - Possible new experiments:
    - **10-Gigabit Ethernet** (soon also on copper), **16-48-Gigabit infiniBand**, **100-Gigabit Ethernet**.

Commodity Links

- More and more often used in **HEP** for **DAQ**, **Event Building** and **High Level Trigger Systems**:
  - Limited costs;
  - Maintainability;
  - Upgradability.
- **Demand of data throughput** in **HEP** is **increasing** following:
  - Physical event rate;
  - Number of electronic channels;
  - Reduction of the on-line event filter (trigger) stages.
- **Industry** has **moved on** since the design of the DAQ for the LHC experiments:
  - **10 Gigabit/s Ethernet** well established;
  - **48 Gigabit/s InfiniBand** available;
  - **96 Gigabit/s InfiniBand** is being actively worked on;
  - **100 Gigabit/s Ethernet** is being actively worked on.

10 Gb/s Technologies

- **Ethernet**:
  - **10 Gb/s** well established
    - Various **optical** standards, short range **copper** (**CX4**), long range **copper** over **UTP CAT6A** standardised), widely used as aggregation technology.
  - Begins to conquer MAN and WAN market (succeeding SONET).
  - Large market share, vendor independent IEEE **standard** (802.3x).
  - Very active R&D on **100 Gigabit/s** and 40 Gigabit/s (will probably die).
- **Myrinet**:
  - Popular cluster-interconnect technology, **low latency**.
  - **10 Gb/s standard** (**optical** and **copper** (CX4) exist)
  - **Single vendor** (Myricom).
- **InfiniBand**:
  - Cluster interconnect technology, **low latency**.
  - **8 Gb/s** and **16 Gb/s** standards (**optical** and **copper**).
  - **Open** industry **standard**, **several vendors** (OEMs) but **very few chipmakers** (Mellanox).
  - Powerful protocol/software stack (reliable/unreliable datagrams, QoS, out-of-band messages etc…).

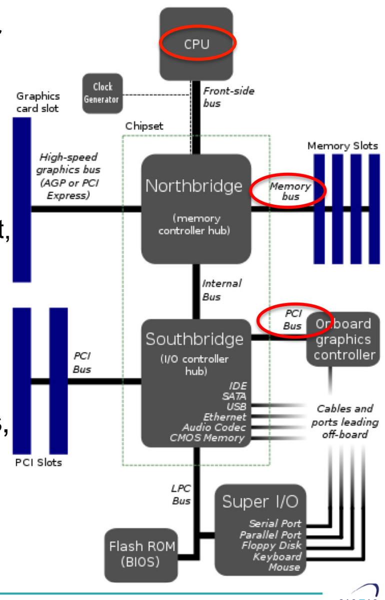
10 Gb/s Technologies (II)

Ethernet
1260 port switch

InfiniBand
3456 port switch

Bottlenecks

- **Direct access** to a high-speed network from a device can incur in 3 major system **bottlenecks**:
  - The **peripheral bus bandwidth**:
    - PCI, PCI-X, PCI-e.
  - The **memory bus bandwidth**:
    - Front Side Bus, AMD HyperTransport, Intel QuickPath Interconnect.
  - The **CPU utilization**.
- "**Fast network**, **slow host**" scenario:
  - **Moore's law**: "Every 18-24 months, computing power doubles…";
  - **Gilder's law**: "Every 12 months, optical fiber bandwidth doubles…".

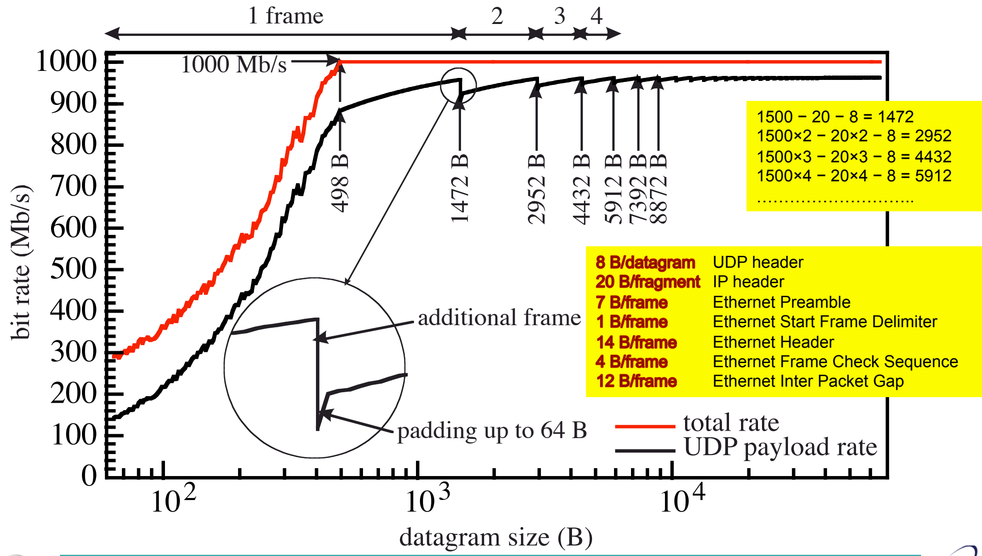


Nomenclature

- **Frame**: **Ethernet** Data Packet:
  - **Standard** Frames: 46 B – 1500 B payload size;
  - **Jumbo** Frames: 46 B – 9000 B payload size.
- **Datagram**: **IP**/**UDP** Data Packet:
  - 20 B – 64 KiB (65535 B) total size.
- **Fragment**: fragment of IP Datagram which fits into an Ethernet frame.
- **Segment**: **TCP** Data Packet:
  - Usually fits into the maximum Ethernet payload size (1500/9000 B).

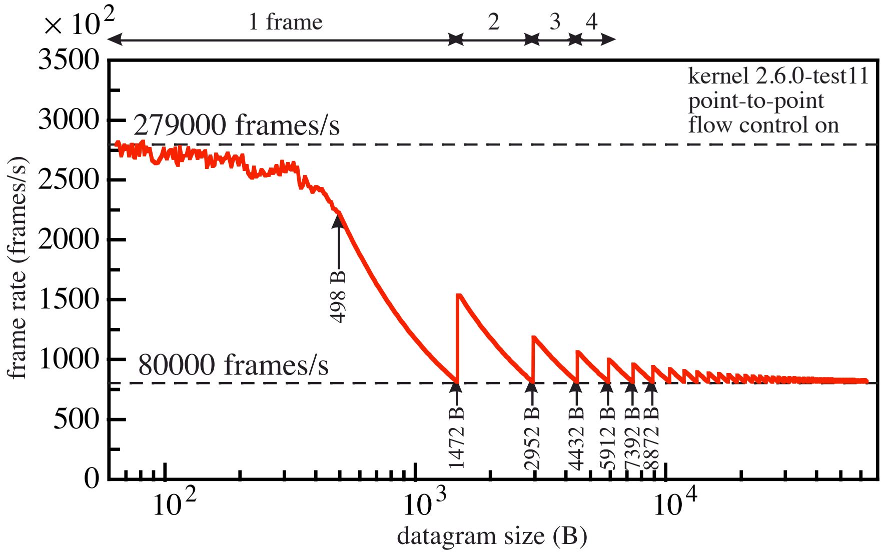
1-Gigabit Ethernet UDP Bit-Transfer Rate

- **Year 2005**, bus **PCI-X** (**bottleneck**).



1-Gigabit Ethernet Frame Transfer Rate

- **Year 2005**, bus **PCI-X** (**bottleneck**).



10-GbE Network I/O

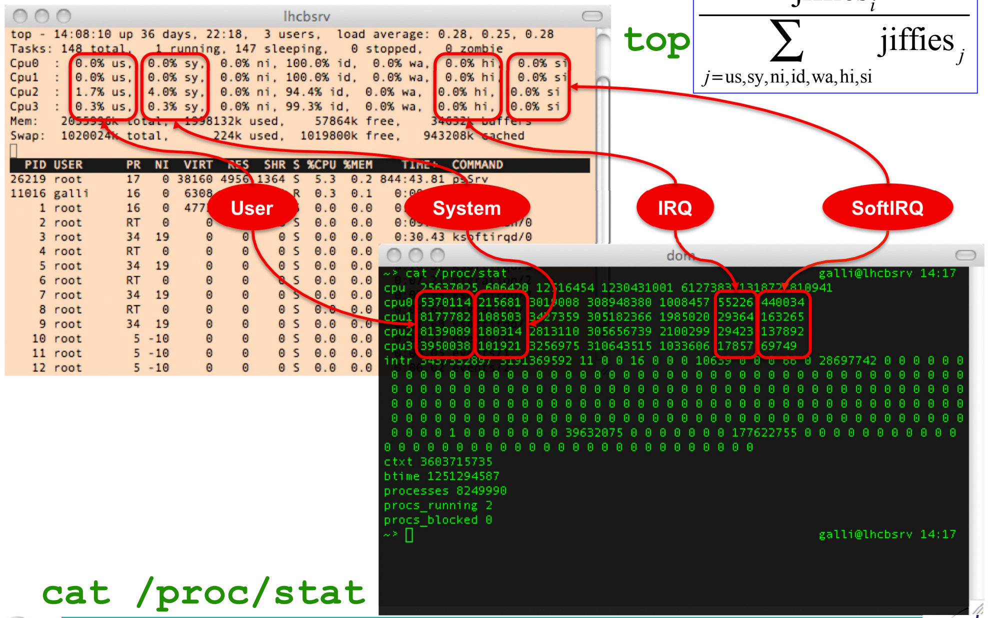
- "Fast network, slow host" scenario.
- **Bottlenecks** in I/O performance:
  - The PCI-X **bus bandwidth** (peak throughput 8.5 Gbit/s in 133 MHz flavor):
    - Substituted by the **PCI-E**, (20 Gbit/s peak throughput in x8 flavor).
  - The **memory bandwidth**:
    - **FSB** has increased the clock from 533 MHz to 1600 MHz.
    - New Memory Architectures:
      - AMD **HyperTransport**;
      - Intel **QuickPath Interconnect**.
  - The **CPU utilization**:
    - **Multi-core** architectures.

Sharing Workload among CPU Cores

- To take advantage of the **multiple cores** of recent CPUs, **workload** should be **shared** among different cores.
- The Linux Kernel splits the process of **sending/ receiving** data packets into **different tasks**:
  - **Differently scheduled** and **accounted**;
  - Can be **partially distributed** over several CPU cores.
- Statistics of **kernel accounting** partitions accessible through the `/proc/stat` pseudo-file:
  - Data relative to **each CPU core**;
  - Partitions relevant to network processing: **User**, **System**, **IRQ** and **SoftIRQ**;
  - Number of **jiffies** (1/1000th of a second) spent by CPU core in each different mode.

Linux Kernel Accounting

$$\frac{\text{jiffies}_i}{\sum_{j=\text{us,sy,ni,id,wa,hi,si}} \text{jiffies}_j}$$



Linux Kernel Accounting (II)

- **User**: User applications which send/receive data packets are typically **ordinary processes** which run in **user mode**:
  - **Non-privileged** execution mode;
  - **No access** to portions of memory allocated by the kernel or by other processes.
- **System**: to access a network device, the applications execute **system calls**, where the execution is switched to **kernel mode**:
  - **Privileged** execution mode (code assumed to be fully trusted);
  - **Any instruction** can be executed and **any memory address** can be referenced;
  - The **portion of the kernel** which is responsible of the required service is actually executed.

Linux Kernel Accounting (III)

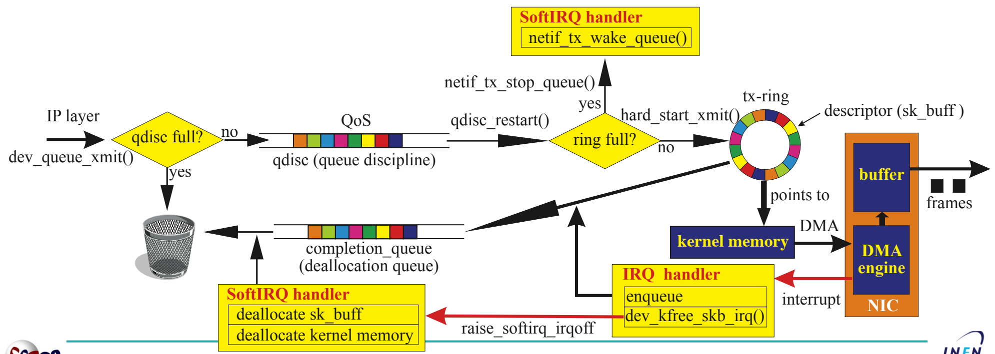
- **IRQ**: Transmission/reception code executed **out of the logical execution flow** of the applications:
  - **Driven by the motion of data packets** through the network.
    - E.g.: when new data packets reach the Network Interface Card (NIC) of a PC through a network cable, a procedure must be executed in order to process the received data and forward them to the appropriate user application which is waiting for data.
  - To this aim the kernel provides **hardware interrupt handlers**, which are **software routines executed upon the reception of hardware interrupt** signals, in our case raised by the NIC.

Linux Kernel Accounting (IV)

- **SoftIRQ**: Code executed **out of interrupt context** (interrupt reception enabled), **scheduled by hardware interrupt** handlers:
  - While the kernel is processing hardware interrupts (**interrupt context**), the interrupt reception is disabled, hence **interrupts** received in the meantime are **lost**.
  - To avoid such a situation, the hardware interrupt handlers perform **only the work which must be accomplished immediately** (**top half**), so **limiting to the minimum the amount of time spent with interrupts disabled**.
  - The **real work** is instead **deferred** to the execution of so-called **software interrupt handlers** (**bottom half**), which are usually scheduled by hardware interrupt handlers;
  - Always executed **on the same CPU** where they were originally raised.

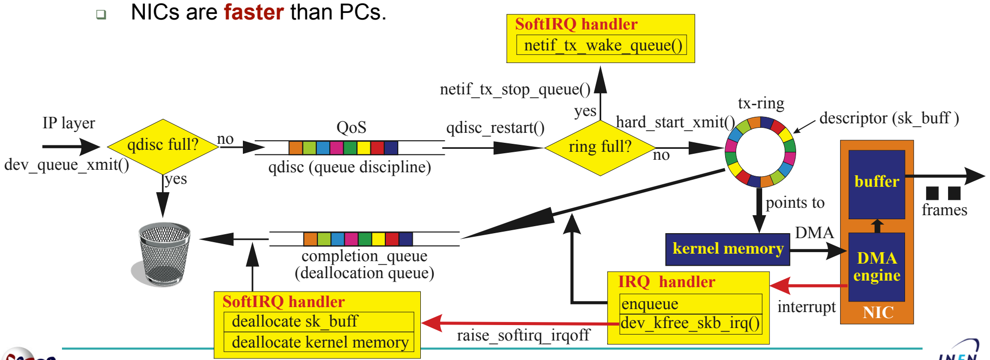
Packet Transmission

- Packet sent from IP layer to Queue Discipline (`qdisc`).
- Any appropriate **Quality of Service** (QoS) in `qdisc`:
  - `pfifo_fast` (packet fifo);
  - `RED` (Random Early Drop);
  - `CBQ` (Class Based Queuing).
- `qdisc` notifies network driver when it's **time to send**: it calls `hard_start_xmit()`:
  - Place all ready `sk_buff` pointers in `tx_ring`;
  - Notifies NIC that packets are ready to send.



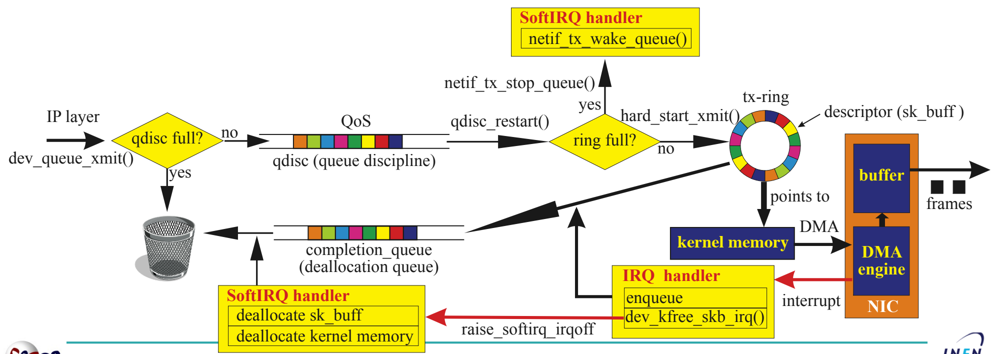
Packet Transmission (II)

- If **immediate sending** is **not possible**:
  - The driver **stops the queuing** of packets by calling `netif_tx_stop_queue()`:
    - **No more calls** to `hard_start_xmit()` **allowed**.
      - Until the queue is woken up by a call to **netif_tx_wake_queue()**.
  - A **SoftIRQ** is scheduled and the packet **transmission** "over the wire" is **deferred** to a later time.
- Could happen if the **device** is running **out of resources**.
- System could in principle generate packets for transmission **faster** than the device can handle.
- Using recent PCs and NICs, **in practice**, this **never happens**:
  - NICs are **faster** than PCs.



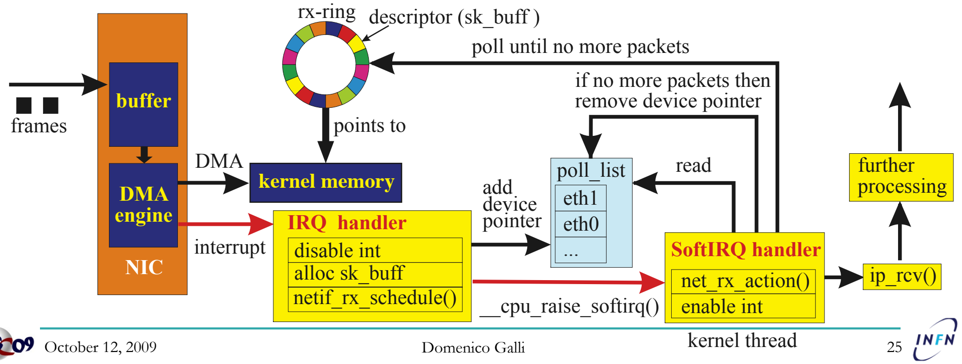
Packet Transmission (III)

- The NIC signals the kernel (via **interrupt**) when packets are **successfully** transmitted:
  - **Highly variable on when interrupt is sent**!
- Interrupt handler **enqueues** transmitted packets **for deallocation** (`completion_queue`);
- At next `softirq`, all packets in the `completion_queue` are deallocated:
  - Meta-data contained in the `sk_buff` struct;
  - Packet data not needed anymore.



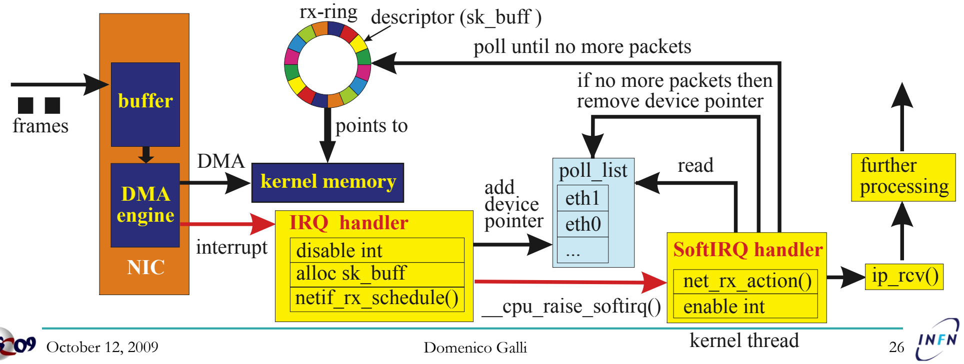
Packet Reception

- NIC **accumulates** a bunch of frames in an **internal buffer**.
- NIC start a **bus-mastered DMA transfer** from the buffer to a reserved space in the **kernel memory**.
  - Packet descriptors (metadata, sk_buff) pointing to data are stored in a circular ring (`rx-ring`).
- As soon as the **DMA transfer has terminated**, the NIC **notifies the kernel** of the new available packets:
  - By means of an **interrupt** signal raised on a **dedicated IRQ line**.
- The **Interrupt Controller** issues an interrupt to the dedicated **processor pin**.



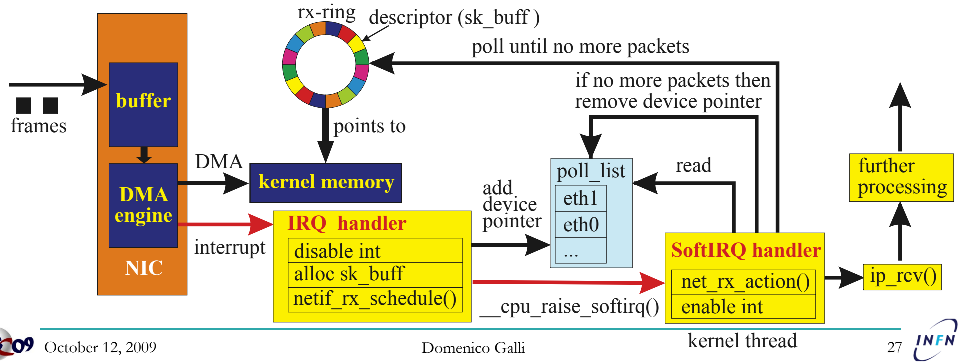
Packet Reception (II)

- The kernel reacts to the IRQ by executing *a* hardware **interrupt handler**.
- The handler **leaves** the packets in the `rx_ring` and **enables polling** mode for the originating NIC:
  - By **disabling the *IRQ* reception** for that NIC and **putting a reference to the NIC** in a **poll-list** attached to the interrupted CPU, and finally schedules a **SoftIRQ**.
- The SoftIRQ handler polls all the NICs registered in the poll-list to **draw packets** from the `rx_ring` (in order to process them) until a configurable number of packets at maximum, known as **quota** and controlled by the parameter `netdev_max_backlog`, is reached.



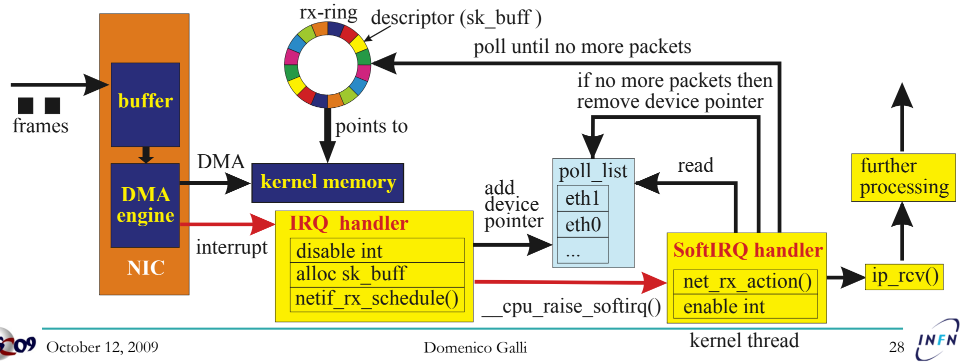
Packet Reception (III)

- If the **quota** is **reached**, but the NIC has **still packets to offer**:
  - Then the NIC is **put at the end of the poll-list**.
- If the **quota** is **reached**, but the NIC has **no more packets to offer**:
  - The NIC is **deleted from the poll-list** and the **IRQ reception** for that NIC is **enabled again**.



Packet Reception (IV)

- Reception mechanism, known as **NAPI** (New Network Application Program Interface):
  - Introduced in the **2.6 kernel** series.
- Main feature:
  - Converge to an **interrupt-driven mechanism** under **light network traffic**:
    - Reducing both latency and CPU load.
  - Converge to to a **poll mechanism** under **high network traffic**:
    - Avoiding **live-lock** conditions:
      - Packets are accepted only **as fast as the system is able process them.**



Setting the Process-to-CPU Affinity

- **Library calls**:
  - `#include <sched.h>`
  - `int` **`sched_setaffinity`** `(pid_t` **`tgid`**`, unsigned int` **`cpusetsize`**`, cpu_set_t *`**`mask`**`)`
  - `int` **`sched_getaffinity`** `(pid_t` **`tgid`**`, unsigned int` **`cpusetsize`**`, cpu_set_t *`**`mask`**`)`
- **Macro** to set/get the CPU mask:
  - `void` **`CPU_CLR`**`(int` **`cpu`**`, cpu_set_t *`**`mask`**`)`
  - `int` **`CPU_ISSET`**`(int` **`cpu`**`, cpu_set_t *`**`mask`**`)`
  - `void` **`CPU_SET`**`(int` **`cpu`**`, cpu_set_t *`**`mask`**`)`
  - `void` **`CPU_ZERO`**`(cpu_set_t *`**`mask`**`)`
- **Parameters**:
  - **`tgid`**: thread group identifier (was pid);
  - **`cpusetsize`**: length (in bytes) of the data pointed to by mask. Normally: `sizeof(cpu_set_t)`.
  - **`mask`**: CPU mask (structure).

Setting the Process-to-CPU Affinity

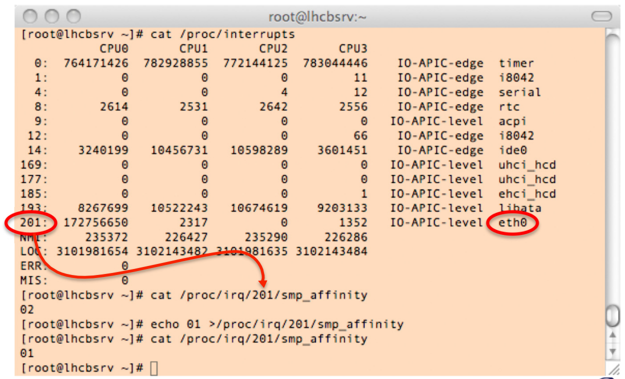
- **Shell commands**:
  - **`taskset`** `[mask] -- [command] [arguments]`
  - **`taskset`** `-p [tgid]`
  - **`taskset`** `-p [mask] [tgid]`
- **Parameters**:
  - **`tgid`**: thread group identifier (was pid);
  - **`mask`**: bitmask, with the lowest order bit corresponding to the first logical CPU and the highest order bit corresponding to the last logical CPU:
    - **`0x00000001`** is processor #0;
    - **`0x00000002`** is processor #1;
    - **`0x00000003`** is processors #0 and #1;
    - **`0x0000000f`** is processor #0 through #3;
    - **`0x000000f0`** is processors #4 through #7;
    - **`0xffffffff`** is all processors (#0 through #31).

Setting the Interrupt-to-CPU Affinity

- Usually **`irqbalance`** daemon running in Linux distributions:
  - **`irqbalance`** **automatically distributes interrupts** over the processors and cores;
  - Design goal of **`irqbalance`**: find a **balance** between **power savings** and **optimal performance**.
- To **manually** optimize network workload distribution among CPU core **`irqbalance`** has to be **switched off**:
  - **`service irqbalance status`**
  - **`service irqbalance stop`**

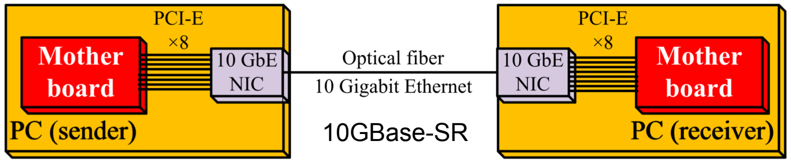
Setting the Interrupt-to-CPU Affinity (II)

- **To find IRQ #**:
  - **`cat /proc/interrupts`**
- **To set CPU Affinity for the handler of IRQ N**:
  - **`echo ⟨mask⟩ >/proc/irq/⟨N⟩/smp_affinity`**



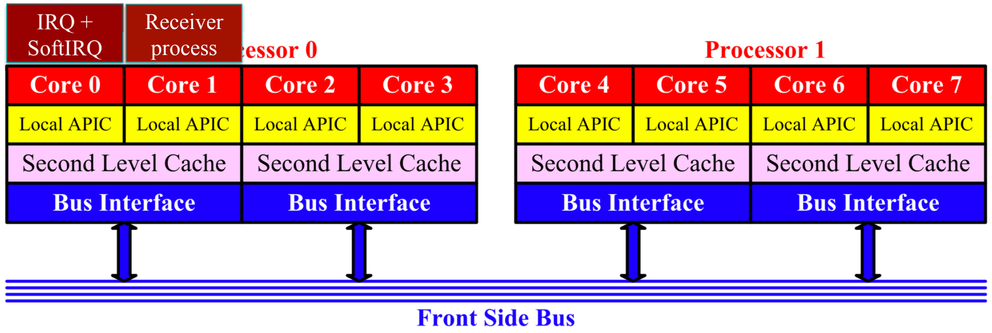
10-GbE Point-to-Point Throughput

- In real operating condition, maximum transfer rate **limited** not only by the capacity of the link itself, but also:
  - By the capacity of the data busses (PCI and FSB);
  - By the ability of the **CPUs** and of the **OS** to handle **packet processing** and **interrupt rates** raised by the network interface cards in due time.
- **Data throughput** & **CPU load** measures reported:
  - **NIC** mounted on the **PCI-E bus** of **commodity PCs** as transmitters and receivers.



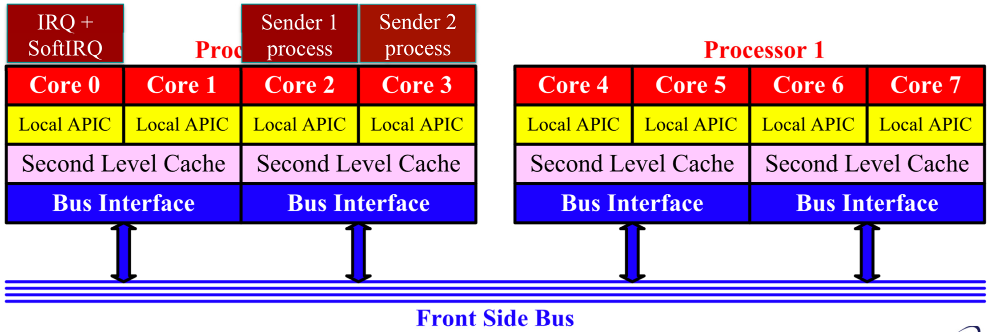
CPU Affinity Settings

10-GbE Receiver		
Core	**L2 Cache**	**Task**
0	0	(IRQ + softIRQ) from Ethernet NIC
1		Receiver process



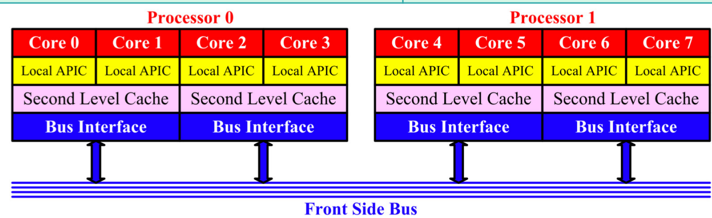
CPU Affinity Settings (II)

10-GbE Sender		
Core	**L2 Cache**	**Task**
0	0	(IRQ + softIRQ) from Ethernet NIC
1		
2	1	Sender process
3		Second sender process [2 sender tests]



Test Platform

Motherboard	IBM X3650
Processor type	Intel Xeon E5335
Procesors x cores x clock (GHz)	2 x 4 x 2.00
L2 cache (MiB)	8
L2 speed (GHz)	2.00
FSB speed (MHz)	1333
Chipset	Intel 5000P
RAM	4 GiB
NIC	Myricom 10G-PCIE-8A-S
NIC DMA Speed (Gbit/s) ro / wo /rw	10.44 / 14.54 / 19.07



Settings

net.core.rmem_max (B)	16777216
net.core.wmem_max (B)	16777216
net.ipv4.tcp_rmem (B)	4096 / 87380 / 16777216
net.ipv4.tcp_wmem (B)	4096 / 65536 / 16777216
net.core.netdev_max_backlog	250000
Interrupt Coalescence (μs)	25
PCI-E speed (Gbit/s)	2.5
PCI-E width	x8
Write Combining	enabled
Interrupt Type	MSI

UDP Data Transfer

- **User Datagram Protocol**:
  - **Connectionless**, **unreliable** messages (**datagrams**) of a fixed **maximum length of 64 KiB**.
  - What does UDP do:
    - **Simple interface to IP** protocol (fragmentation, routing, etc.);
    - **Demultiplexing** multiple processes using the **ports**.
  - What does **not** UDP do:
    - Retransmission upon receipt of a bad packet;
    - Flow control;
    - Error control;
    - Congestion control.

bits	0 - 15	16 - 31
0	Source Port	Destination Port
32	Length	Checksum
64	Data	

Why UDP?

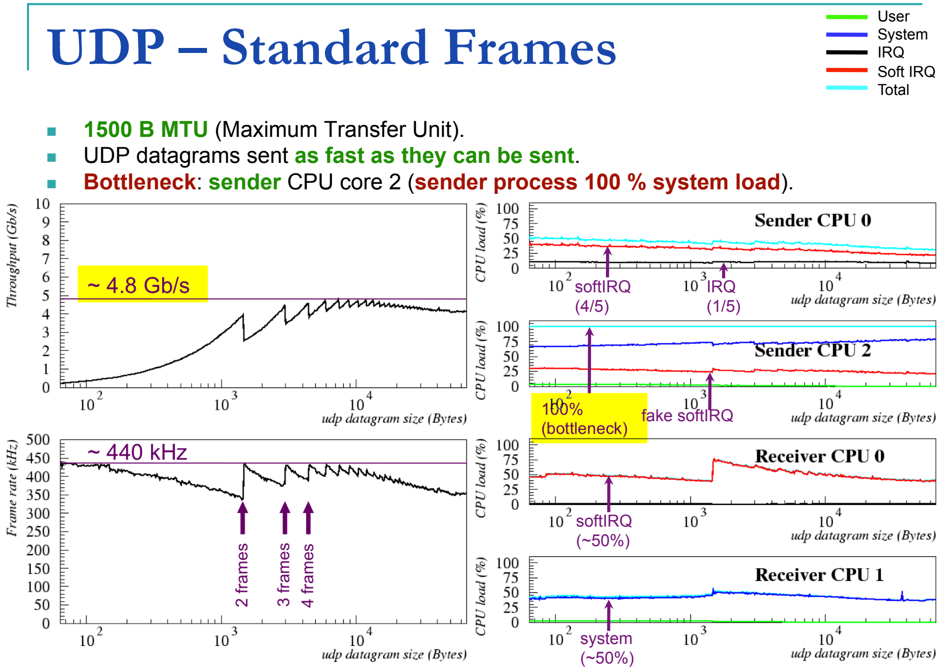
- TCP is optimized for **accurate** delivery rather than for **timely** delivery:
  - **Relatively long delays** (in the order of **seconds**) while waiting for **out-of-order** messages or **retransmissions** of lost messages.
- TCP not particularly suitable for **real-time applications**:
  - In time-sensitive applications, **dropping** packets is sometimes **preferable** to **waiting** for **delayed** or **retransmitted** packets.
  - UDP/RTP (Real-time Transport Protocol) preferred:
    - e.g. Voice over IP.

Why UDP in DAQ Chain?

- **High link occupancy** is **desirable**:
  - To maximize the physical event rate.
- The **data flow** is **driven** by **accelerator/detector** rates (**time-sensitive** application):
  - Independent on the PC which process data.
- Mechanisms which **slow down** data transmission are **not** appreciated:
  - E.g. in TCP: **slow start**, **congestion avoidance**, **flow control**.
- Mechanisms for **reliability** (retransmission) can be **useless** due to **latency limits**.
- **Retransmission** requires **additional bandwidth**, which is **stolen from the event bandwidth**:
  - If the **available bandwidth** is **limited**, retransmission will probably trigger a throttling system which discards physical events in any case.

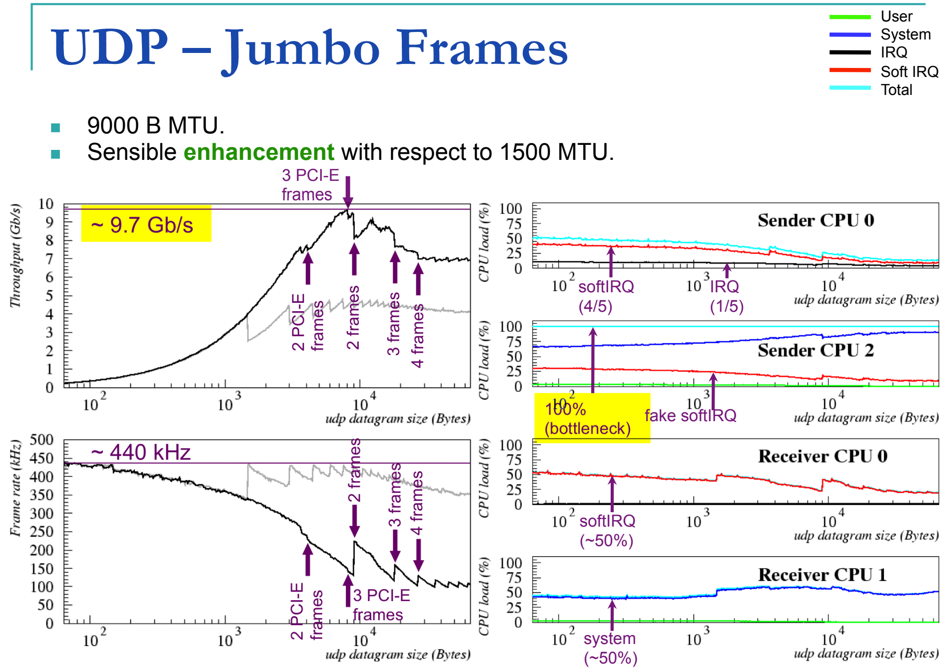
UDP – Standard Frames

- **1500 B MTU** (Maximum Transfer Unit).
- UDP datagrams sent **as fast as they can be sent**.
- **Bottleneck**: **sender** CPU core 2 (**sender process 100 % system load**).



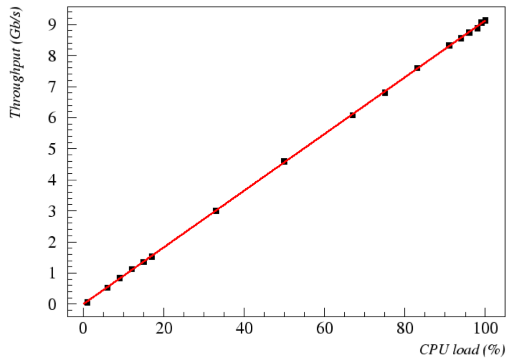
UDP – Jumbo Frames

- 9000 B MTU.
- Sensible **enhancement** with respect to 1500 MTU.



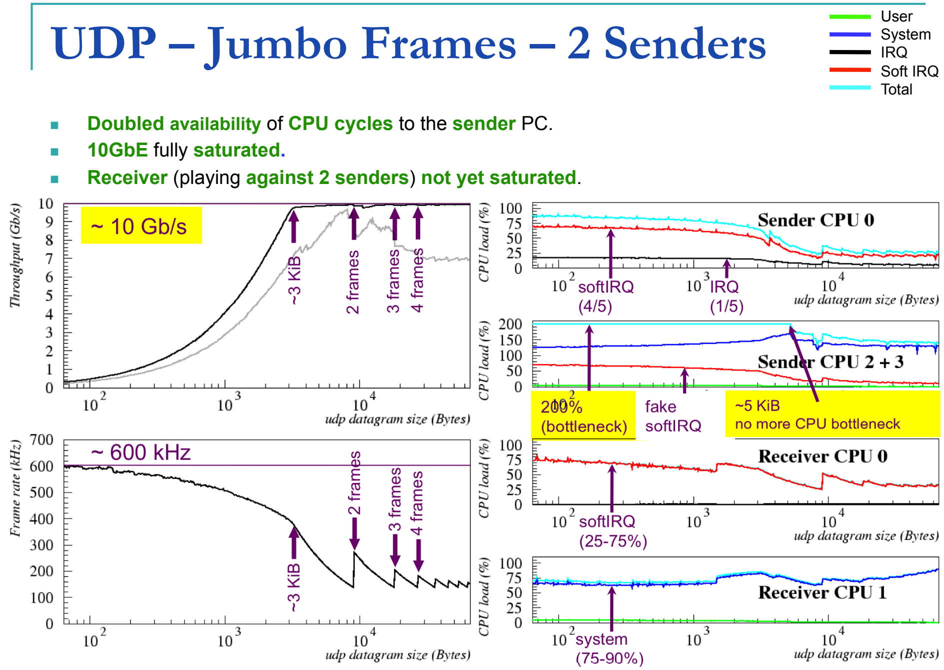
UDP – Jumbo Frames (II)

- Additional dummy **ps**, **bound** to the **same core** of the **tx ps** (CPU 2), **wasting CPU resources**.
- CPU available for tx process **trimmed** using relative **priority**.
- The **perfect linearity** confirms that the **system CPU load** @ **sender side** was actually **the main bottleneck**.
- **2 GHz ➔ 3 GHz CPU** (same architecture):
  - Potential increase of 50% in the maximum throughput:
    - Provided that bottlenecks of other kinds do not show up before such increase is reached.



UDP – Jumbo Frames – 2 Senders

- **Doubled availability** of **CPU cycles** to the **sender** PC.
- **10GbE** fully **saturated.**
- **Receiver** (playing **against 2 senders**) **not yet saturated**.



TCP Data Transfer

- **Transmission Control Protocol**:
  - Provides a **reliable** end-to-end **byte stream** over an unreliable network.
  - Designed to **dynamically adapt** to properties of the internetwork and to be **robust** in the face of many kinds of failures.
- TCP **breaks** outgoing data **streams** into pieces (**segments**) which usually **fit** in a single network **frame** and which are sent as **separate IP datagrams**.

TCP Header

<table>
<tr><th>Bit offset</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td colspan="16">Source port</td><td colspan="16">Destination port</td></tr>
<tr><td>32</td><td colspan="32">Sequence number</td></tr>
<tr><td>64</td><td colspan="32">Acknowledgment number</td></tr>
<tr><td>96</td><td colspan="4">Data offset</td><td colspan="4">Reserved</td><td>CWR</td><td>ECE</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td colspan="16">Window Size</td></tr>
<tr><td>128</td><td colspan="16">Checksum</td><td colspan="16">Urgent pointer</td></tr>
<tr><td>160
...</td><td colspan="32">Options (if Data Offset > 5)
...</td></tr>
</table>

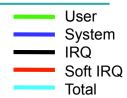
TCP Data Transfer (II)

- TCP key feature:
  - **Ordered data transfer**:
    - The destination host rearranges segments according to **sequence number**.
  - **Retransmission of lost packets**:
    - Any cumulative stream **not acknowledged** will be retransmitted.
  - **Discarding duplicate packets**.
  - **Error-free data transfer**:
    - Checksum.
  - **Flow control** (sliding windows):
    - **Limits the rate** a sender transfers data to guarantee reliable delivery;
    - The receiver specifies in the receive **window** field the amount of additional received data (in bytes) that it is willing to buffer for the connection;
    - When the receiving host's buffer fills, the next acknowledgement contains a 0 in the window size, to stop transfer and allow the data in the buffer to be processed.
  - **Congestion avoidance**:
    - Avoid congestion collapse.

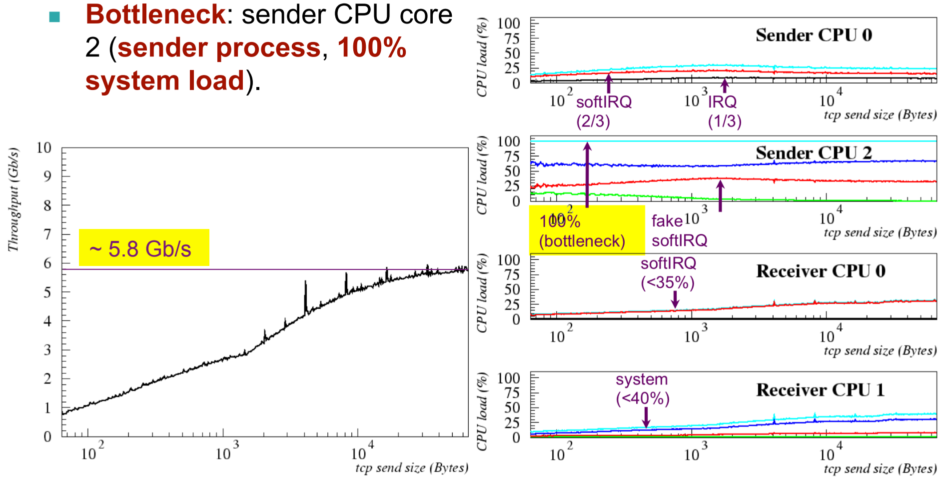
TCP Data Transfer (III)

- TCP provides many **additional control mechanisms**:
  - **Selective acknowledgments**;
    - Allows the receiver to acknowledge discontiguous blocks of packets that were received correctly.
  - **Nagle's algorithm**:
    - To cope with the small packet problem.
  - **Clark's solution**:
    - To cope with the silly window sindrome.
  - **Slow-start**, **congestion avoidance**, **fast retransmit**, and **fast recovery**:
    - Which cooperate to congestion control.
  - **Retransmission timeout**:
    - **Karn's algorithm**, **TCP timestamps**, **Jacobson's algorithm** for evaluating round-trip time.

TCP – Standard Frames

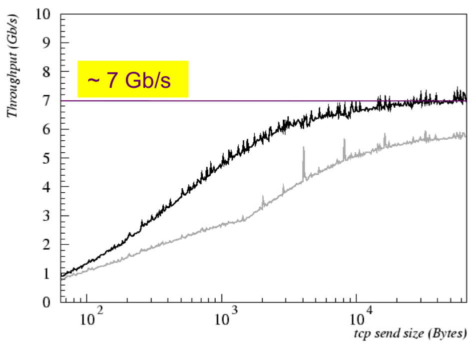


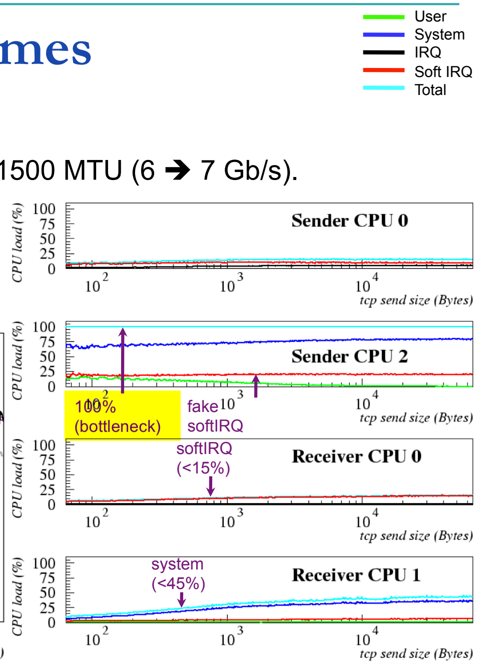
- **1500 B MTU** (Maximum Transfer Unit).
- TCP segments sent **as fast as they can be sent**.
- **Bottleneck**: sender CPU core 2 (**sender process**, **100% system load**).



TCP – Jumbo Frames

- **9000 B MTU**.
- **Enhancement** with respect to 1500 MTU (6 ➔ 7 Gb/s).
- **Bottleneck**: sender CPU core 2 (**sender process**, **100% system load**).





Nagle's Algorithm

- Nagle's algorithm **active by default** when using **TCP-streamed transfers**.
- Introduced in the TCP/IP stack (RFC 896) in order to solve the so called **small packet problem**.
  - An application repeatedly emits data in **small chunks**, frequently only 1 byte in size. Since TCP packets have a 40 byte header (20 bytes for TCP, 20 bytes for IPv4), this results in a 41 byte packet for 1 byte of useful information, a **huge overhead**.
  - This situation often occurs in **`telnet` sessions**, where most key-presses generate a single byte of data which is transmitted immediately.
    - Worse, over slow links, many such packets can be in transit at the same time, potentially leading to **congestion collapse**.
- The **Nagle's algorithm** automatically **concatenates** a number of small data packets in order to **increase the efficiency** of a network application system, i.e. **reducing the number of physical packets** that must be sent.

Nagle's Algorithm (II)

- When there are **few bytes to send**, but **not a full packet's worth**, and there are **some unacknowledged data in flight**:
  - Then the Nagle's algorithm **waits**, **keeping data buffered**, until:
    - Either **the application provides more data**:
      - Enough to make **another full-sized TCP segment** or **half of the TCP window size**;
    - Or **the other end acknowledges all the outstanding data**, so that there are **no** longer any **data in flight**.

Linux Settings on Nagle's Algorithm

- The Linux operating system provides two options to **disable** the Nagle's algorithm in **two opposite ways**, which can be set by means of the **`setsockopt()`** system call:
- **`TCP_NODELAY`**
  - The OS always **send segments as soon as possible**:
    - Even if there is only a small amount of data.
  - The **behavior** of **TCP** transfers is expected to **match more closely** that of **UDP** ones:
    - Since **no** small packet **aggregation** at the sender side is performed.
- **`TCP_CORK`**
  - The OS **does not send out partial frames at all** until the application provides more data:
    - Even if the other end acknowledges all the outstanding data.
  - Only full frames can be sent out:
    - If an application does not fill the last frame of a transmission, the system will delay sending the last packet forever.

Linux Settings on Nagle's Algorithm (II)

cork.c (~/Documents/didattica/ESC09) – VIM1

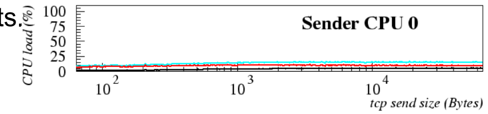
```c
#include <sys/socket.h>
int main()
{
  int mySock;
  int opt=1;
  ......................................
  mySock = socket(AF_INET,SOCK_STREAM,0);
  ret_code = setsockopt(mySock, IPPROTO_TCP, TCP_CORK, (char*)&opt, sizeof(opt));
  ......................................
}
```

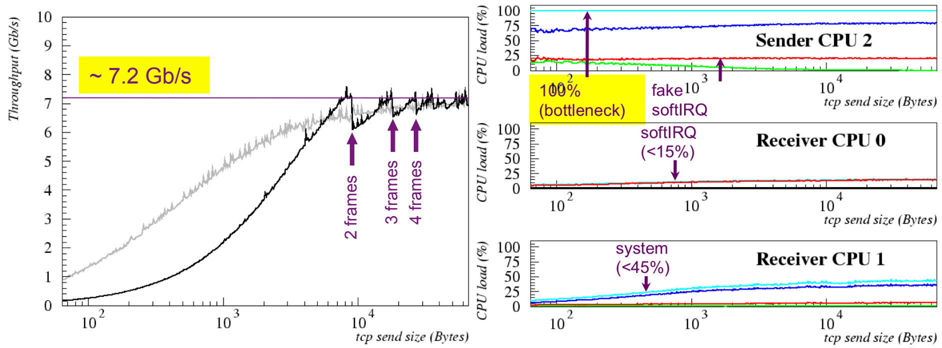
nodelay.c (~/Documents/didattica/ESC09) – VIM

```c
#include <sys/socket.h>
int main()
{
  int mySock;
  int opt=1;
  ......................................
  mySock = socket(AF_INET,SOCK_STREAM,0);
  ret_code = setsockopt(mySock, IPPROTO_TCP, TCP_NODELAY, (char*)&opt, sizeof(opt));
  ......................................
}
```


TCP – Jumbo – TCP_NODELAY

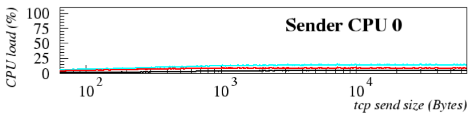
- **Nagle's algorithm disabled**. **Segments** are always **sent as soon as possible**, even if there is only a small amount of data.
- **Small data packets no longer concatenated.**
- **Discontinuities** of the UDP tests.
- **UDP throughput not reached**, due to the latency overhead of the TCP protocol.

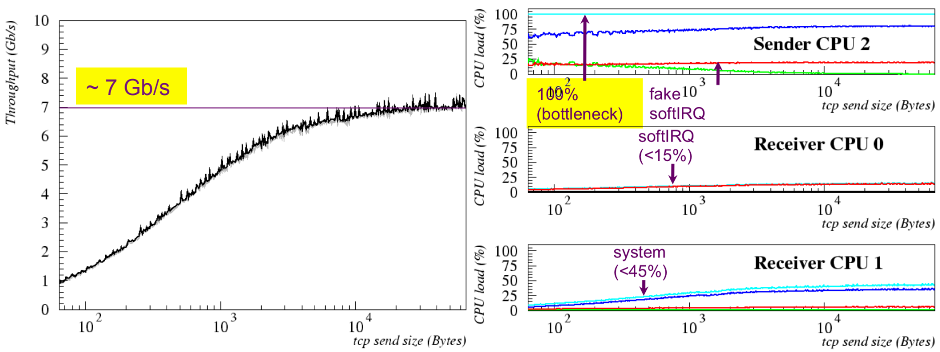




TCP – Jumbo – TCP_CORK

- **Nagle's algorithm disabled**. OS **does not send out partial frames at all** until the application provides more data, even if the other end acknowledges all the outstanding data.
- **No relevant differences**.





TCP – Zero Copy

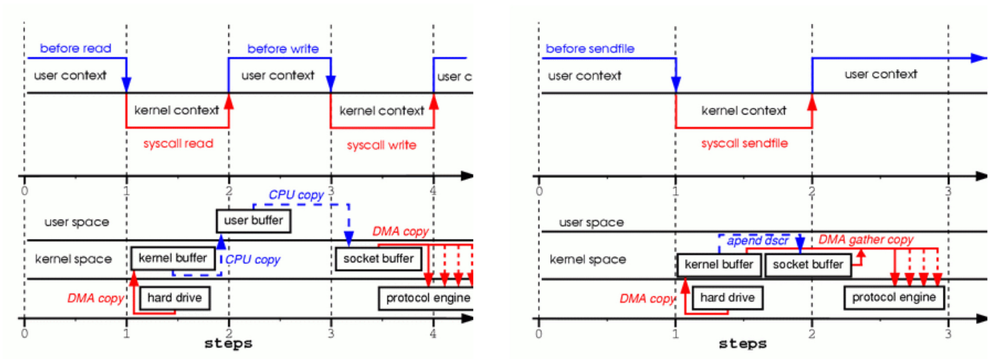
- The `send()` system call is used to send data stored in a **buffer** in the **user space** to the network through a TCP socket.
  - This requires the copy of the data from the user space to the kernel space on transmission.
- The `sendfile()` system call allows to send data read from a **file descriptor** to the network through a TCP socket.
  - Since both the **network** and the **file** are **accessible** from **kernel mode**, any time-expensive copy from user space to kernel space can be avoided.

TCP – Zero Copy (II)

- `#include <sys/sendfile.h>`
- `ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);`

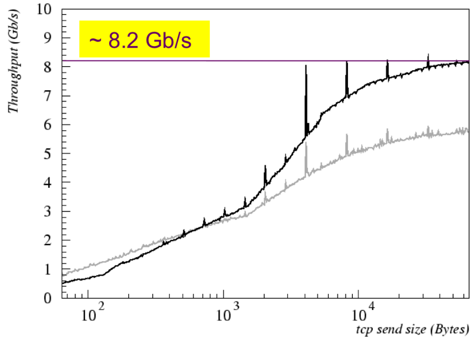
- `out_fd`: file descriptor of the output socket;
- `in_fd`: file descriptor of the open file;
- `offset`: start position in file;
- `count`: number of Bytes to be copied.

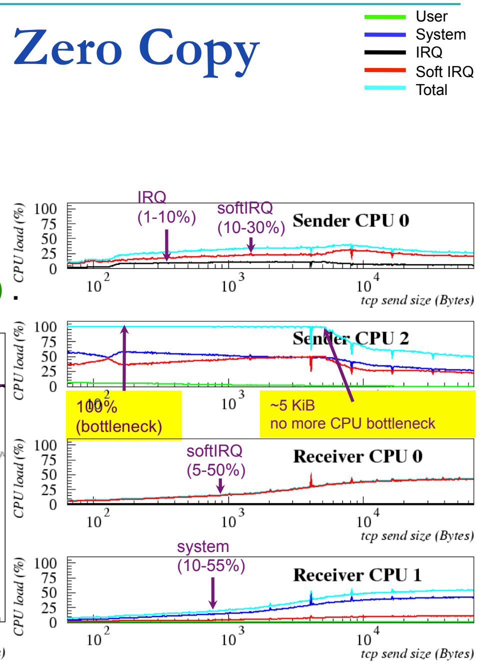
`read()+send()` **`sendfile()`**



TCP – Standard – Zero Copy

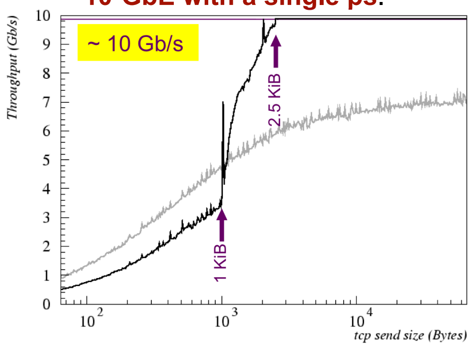
- **1500 B** MTU.
- **Significant increase in throughput**
  - with respect to `send()`.

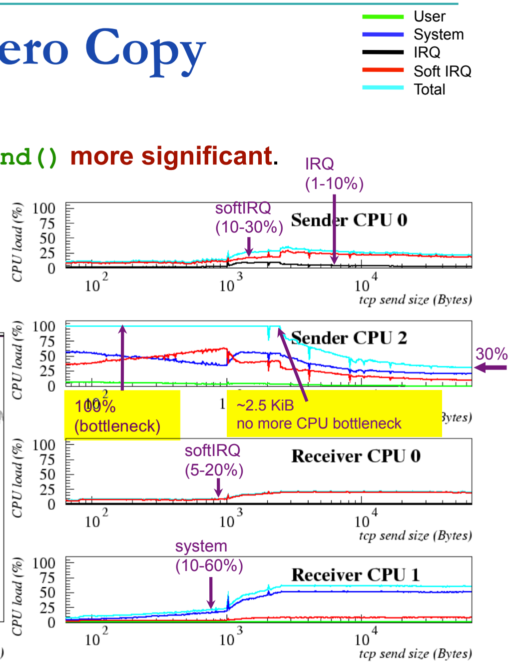




TCP – Jumbo – Zero Copy

- `sendfile()` system call.
- **Improvement** with respect to `send()` **more significant**.
- For send size > 2.5 KiB:
  - Throughput = 10 Gbit/s
  - Sender CPU 2 load < 100%:
    - down to 30%.
- **Only test able to saturate 10-GbE with a single ps.**





TCP Hardware Offload

- Modern network adapters usually implement various kinds of **hardware offload functionalities**:
  - The **kernel can delegate** heavy parts of its tasks **to the adapter**.
  - This is one of the most effective means available to improve the performance and reduce the CPU utilization.

Setting the Hardware Offload

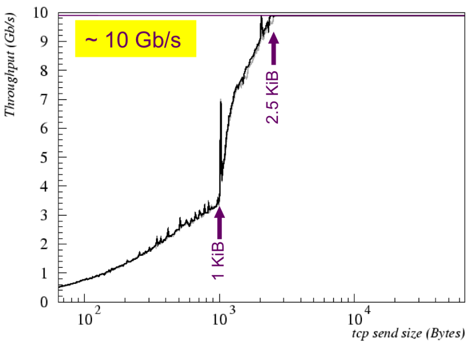
- Print offload functionalities:
  - `ethtool -k ethX`
- Set offload functionalities:
- `ethtool -K ethX [rx on|off] [tx on|off] [sg on|off] [tso on|off] [ufo on|off] [gso on|off] [gro on|off] [lro on|off]`
  - **rx**: receiving checksumming;
  - **tx**: transmitting checksumming;
  - **sg**: scatter-gather I/O;
  - **tso**: TCP segmentation offload;
  - **ufo**: UDP fragmentation offload;
  - **gso**: generic segmentation offload;
  - **gro**: generic receive offload;
  - **lro**: large receive offload.

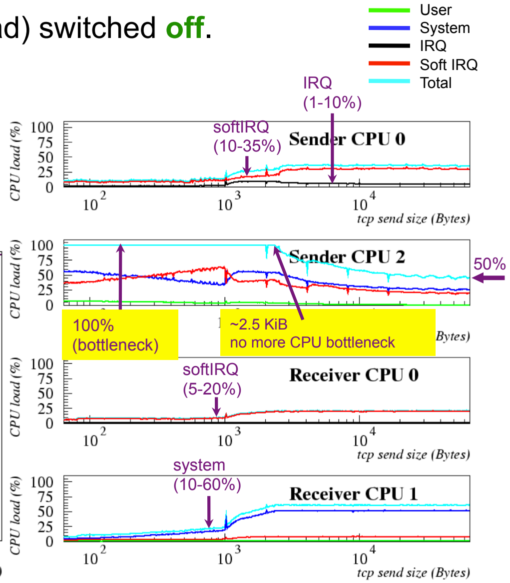
TCP Segmentation Offload (TSO)

- When a **data packet larger than the MTU** is sent by the kernel to the network adapter, the data must first be **sub-divided into MTU-sized packets** (**segmentation**).
- With **old adapters**, this task was commonly performed at the **kernel level**, by the **TCP layer** of the TCP/IP stack.
- In contrast, when **TSO is supported** and active, the host **CPU is offloaded** from such a segmentation task, and it can **pass** segments **larger than one MTU** (**up to 64 KiB**) **to the NIC** in a single transmit request.

TCP – Jumbo – Zero Copy – No TSO

- **TSO** (TCP Segmentation Offload) switched **off**.
- **No differences in throughput**:
  - 10-GbE link already **saturated** (size > 2.5 KiB).
- **CPU load reduced** by TSO (size > 2.5 KiB).



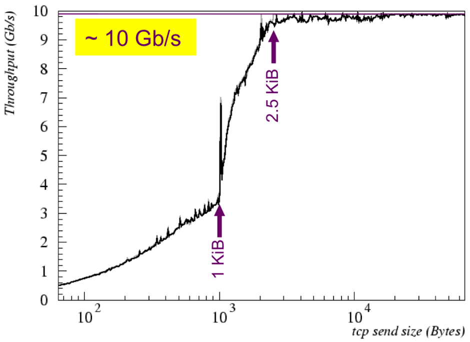


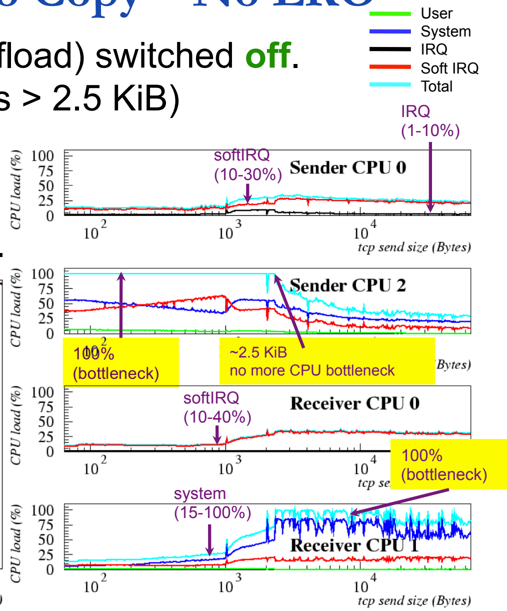
Large Receive Offload (LRO)

- Assists the **receiving host** in **processing incoming TCP packets**:
  - By **aggregating** them at the **NIC level** into fewer larger packets;
  - It may **reduce** considerably the number of physical packets actually processed by the kernel;
  - Hence offloading it in a significant way.

TCP – Jumbo – Zero Copy – No LRO

- **LRO** (Large Receive Offload) switched **off**.
- The **performance** (sizes > 2.5 KiB) **slightly worse**;
- Total **load** of the CPU 1 receiver sensibly **increased**, **up to 100%**.



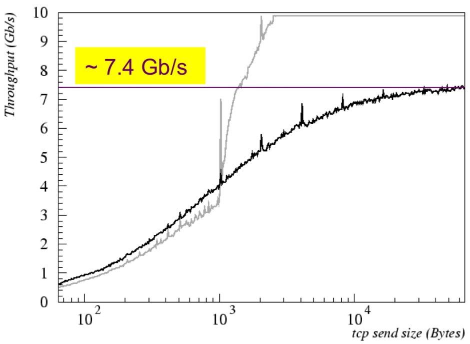


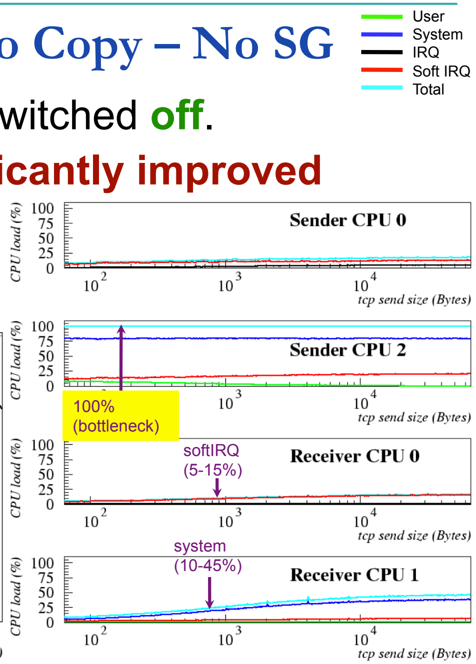
Scatter-Gather I/O (SG)

- The process of **creating a segment** ready to be transmitted through the network, starting **from the transmission requests** coming from the TCP layer, in general requires **data buffering**:
  - In order to assemble **packets** of **optimal size**, to evaluate **checksums** and to add the TCP, IP and Ethernet **headers**.
- This procedure can require a fair amount of **data copying** into a new buffer:
  - To make the **final linear packet**, stored in **contiguous memory locations**.
- However, if the NIC that has to transmit the packet can perform **SG I/O**, the **packet does not need to be assembled into a single linear chunk**:
  - Since the NIC is able to **retrieve through DMA** the **fragments** stored in **non-contiguous memory locations**.
  - This hardware optimization **offloads the kernel** from such a **linearization duty**, hence improving performance.

TCP – Jumbo – Zero Copy – No SG

- **Scatter-gather I/O** switched **off**.
- **Performance significantly improved** by SG I/O.



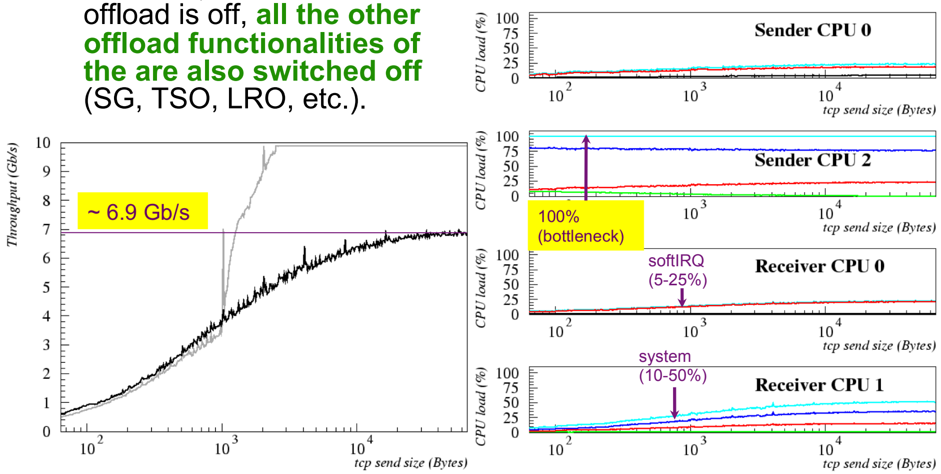


Checksum Offload (CO)

- IP/TCP/UDP **checksum** is performed to make sure that the packet is correctly transferred:
  - by **comparing**, at the receiver side, the value of the checksum field in the **packet header** (set by the sender) with the **value calculated** by the receiver from the packet payload.
- The task of evaluating the TCP checksum can be **offloaded to the NIC** thanks to the so-called Checksum Offload.

TCP – Jumbo – Zero Copy – No CO

- **Checksum offload** switched **off**.
- **Performance is significantly improved**.
- However, when the checksum offload is off, **all the other offload functionalities of the are also switched off** (SG, TSO, LRO, etc.).



Summary

- Main **bottleneck**:
  - **CPU** utilization at the **sender** side:
    - **System load** of the **transmitter process**.
- **Optimization**:
  - CPU **workload** can be **distributed** among **2 CPU** cores by **separating** the sender/receiver **process** from the IRQ/SoftIRQ **handlers**.
  - **Jumbo** frames in fact **mandatory** for 10-GbE.
  - In **TCP transmission**:
    - Improvement can be obtained by **zero-copy** (`sendfile()`);
    - **Scatter-Gather** functionality sensibly improves the performance;
    - The **TSO** functionality helps the sender CPU.
    - The **LRO** functionality helps the receiver CPU.
- **Performances**: review of data transfer via 10-GbE links at full speed:
  - Using either the **UDP** or the **TCP** protocol;
  - By varying the **MTU** and the **packet send size**;
  - **2 UDP sender** needed to **saturate** the link:
    - **1 receiver can play against 2 senders**;
  - Using **TCP+zero-copy+offload**, **1 sender is enough** to saturate the link;
  - **Packet size** crucial:
    - Using 10-GbE you could transfer data at 200 Mb/s maximum!

More Details

Inaccuracy in SoftIRQ Accounting

- Inaccuracy in the kernel accounting (2.6.9-78.0.1, SLC 4.7) can lead to a **wrong assignment of jiffy counts to the SoftIRQ partition**.
- The CPU tick is accredited to **SoftIRQ** if the **softirq_count() macro** returns a value **≠0**.
- This **value** is **incremented** by `__local_bh_disable()` function:
  - **Usually** called by the `__do_softirq()` function, i.e. the one which actually executes the **SoftIRQ** code.
- Problem: `__local_bh_disable()` **also called by** `local_bh_disable()`:
  - In turn **called by other functions in the kernel**.
- If a **timer interrupt** employed for accumulating kernel accounting statistics happens when the kernel is executing a function where `local_bh_disable()` has been called, the tick is incorrectly accredited to **SoftIRQ** time, while it **was indeed a synchronous code** and had to be accounted to **System**.