# BUSY POLLING
# Netdev 2.1

Past, Present, Future

# Presented by :

Eric Dumazet  @ Google

But many many thanks to all contributors over the years :

**Jesse Brandeburg, Eliezer Tamir, Alexander Duyck, Sridhar Samudrala, Willem de Bruijn, Paolo Abeni, Hannes Frederic Sowa, Zach Brown, Tom Herbert, David S. Miller and others**

# BUSY POLLING, What is that ?

As Jesse Brandeburg first stated in Linux Plumber Conference 2012 [1], busy polling has been first defined as the ability for user thread waiting for some incoming network message to directly poll the device ring buffer, instead of traditional way of waiting on a socket receive buffer being feeded by normal RX handling.

[1] http://www.linuxplumbersconf.org/2012/wp-content/uploads/2012/09/2012-lpc-Low-Latency-Sockets-slides-brandeburg.pdf

# Traditional Model, how does it work ?

Incoming network message, DMA from NIC to host memory.
<variable delay, caused by interrupt mitigation strategy>
Hard Interrupt.
<variable delay, caused by host scheduling constraints>
SoftIRQ (BH) NAPI poll(), IP/TCP stack processing.
Packet queued into socket receive buffer, wakeup of application.
<variable delay, caused by host scheduling other constraints>
Application reads the packet, process it, and eventually block again for the next one.

# Why is the Traditional Model hurting ?

TLDR; Throughput and Latencies are conflicting

The model was designed long ago, when fewer cpus were available on the hosts, and we had to carefully find a model that could achieve good performance for the available rates. On Uniprocessor host, cpu had to use a split between hard irq, soft irqs and process context cycles, and adding batches was the key for reasonable throughput.

We now reach more than 100 "cpus" per host, but the model have not fundamentally changed, maybe because it was actually quite good.

# Burn cycles to remove delays. ????

Busy Polling is really the way for some highly latency sensitive application to bypass the first stages of the Traditional Model,

**not waiting** for the Interrupts (Hard IRQ, Soft IRQ) and associated delays.

This costs one cpu per application thread, but can indeed be a significant win.

# Busy Polling History : The Past

Eliezer Tamir submitted first rounds of patches for linux-3.11 in 2013

The patch set demonstrated significant gains on selected hardware (Intel ixgbe) and was followed by few drivers changes to support the new driver method, initially called ndo_ll_poll() and quickly renamed into ndo_busy_poll() .

linux-3.11 also got bnx2x and mlx4 support

Later, myri10ge, be2net, ixgbevf, enic, sfc and cxgb4 support came.

Busy polling was tested for TCP and connected UDP sockets, using standard system calls : recv() and friends , poll() and select()

Results were magnified by quite high interrupt coalescing (ethtool -c) parameters that favored cpu cycles savings at expense of latencies.

# Some numbers, because.... why not ?

mlx4 for example has following defaults:

rx-usecs: 16

rx-frames: 44

TCP_RR (1 byte payload each way) on 10Gbit mlx4 would show **17500** transactions per second without Busy Polling, and **63000** with Busy Polling.

# API (sysctls)

Two global sysctls were added in µs units :

**/proc/sys/net/core/busy_read**

**/proc/sys/net/core/busy_poll**

Suggested settings are in the 30 to 100 µs range.
Their use is very limited, since they enforce busy polling for all sockets, which is not desirable. They provide quick and dirty way to test busy polling with legacy programs on dedicated hosts.

# API (socket options)

SO_BUSY_POLL is a socket option, that allows precise enabling of busy polling, although its use is restricted to CAP_NET_ADMIN capability.

This limitation came from initial busy polling design, since we were disabling (at that time) software interrupts (BH) for the duration of the busy polling enabled system call.

We might remove this limitation now that Busy Polling is a good citizen.

# Linux 4.5 changes

sk_busy_loop() was changed to let Soft IRQ (BH) being

serviced. Hint : Look at the back of the Netdev 2.1 t-shirt ;)

Main idea was that if we were burning cpu cycles, we could at the same time spend them for more useful things, that would have added extra latencies anyway right before returning from the system call.

Some drivers (eg mlx4) use different NAPI contexts for RX and TX, this change permitted to handle TX completions smoothly.

# Linux-4.5 changes (2)

Another step was to make ndo_busy_poll() optional, and use existing NAPI logic instead. mlx5 driver got busy polling support by this way.

Also note that we no longer had to disable GRO on interfaces to get lowest latencies, as first driver implementations did.

This is important on high speed NIC, since GRO is a key to decent performance of TCP stack.

# Linux-4.5 changes (3)

ndo_busy_poll() implementation in drivers required the use of an extra synchronization between the regular NAPI logic (hard interrupt > napi_schedule() > napi poll()) and the ndo_busy_poll() .

This extra synchronization required one or two extra atomic operations in the non-busy-polling fast path, which was unfortunate.

The driver implementations first used a dedicated spinlock, then Alexander Duyck used a cmpxchg()

# Linux-4.10 changes

We enabled busy polling for unconnected UDP sockets, in some cases.

We also changed napi_complete_done() to return a boolean, allowing a driver to not rearm interrupts if busy polling is controlling NAPI logic.

```
done = mlx4_en_process_rx_cq(dev, cq, budget);
if (done < budget &&
    napi_complete_done(napi, done))
        mlx4_en_arm_cq(priv, cq);
return done;
```

# Linux-4.11 changes

We finally got rid of all ndo_busy_poll() implementations in drivers and in core.

Busy Polling is now a core NAPI infrastructure, requiring no special support from NAPI drivers.

Performance gradually increased, at least on mlx4, going from **63000** on linux-3.11 (when first LLS patches were merged) to **77000** TCP_RR transactions per second.

# Linux-4.12 changes

epoll() support was added by Sridhar Samudrala and Alexander Duyck, with the assumption that an application using epoll() and busy polling would first make sure that it would classify sockets based on their receive queue (NAPI ID), and use at least one epoll fd per receive queue.

SO_INCOMING_NAPI_ID was added as a new socket option to retrieve this information, instead of relying on other mechanisms (CPU or NUMA identifications).

# linux-4.12 changes (cont)

Ideally, we should add eBPF support so that SO_REUSEPORT enabled listeners can choose the appropriate silo (per RX queue listener) directly at SYN time, using an appropriate SO_ATTACH_REUSEPORT_EBPF program.

Same eBPF filter would apply for UDP traffic.

# Lessons learned

Since LLS effort came from Intel, we accepted quite invasive code in drivers to demonstrate possible gains. Years passed before coming to a core implementation and remove the leftovers, mostly because of lack of interest or time.

Another point is that Busy Polling was not really deployed in production.

If too many threads/applications want to simultaneously use Busy Polling, then process scheduler takes over and has to arbitrate among all these cpu-hungry threads, adding back the jitter issues.

# Busy Polling : What's next ?

Zach Brown asked in an email sent to netdev if there

was a way to get NAPI poll all the time.

https://lkml.org/lkml/2016/10/21/784

While this is doable by having a dummy application looping on a recv() system call on properly setup socket(s), we probably can implement something better.
This mail, plus some discussions we had in netdev 1.2 (Tokyo 2016) made me work again on Busy Polling (starting in linux-4.10 as mentioned above).

# Look Ma, there is a pipeline !

Program flow, traverse all layers from the application to the hardware access.

Example at transmit :

- sendmsg()
- fd lookup
- tcp_sendmsg(), skb allocation, copy user data to kernel space
- tcp_write_xmit() (sk->sk_wmem_alloc)
- IP layer (skb->dst->__refcnt), neighbour layer
- qdisc enqueue
- qdisc dequeue (qdisc_run())
- grab device lock, dev_queue_xmit()
- ndo_start_xmit()
- roll back all the way down to user application

# Too many cpus in networking stacks hit a wall

With SMP, we added spinlocks and atomics in order to protect the data structures and communication channels.

Multi Queue NIC and spinlock contention avoidance naturally lead us to use one queue per cpu, or a significant number of queues.

Increasing number of queues had the side effect of reducing NAPI batching.

Experts complain about enormous amount of cpu cycles spent in softirq handlers. It has been shown that driving a 40Gbit NIC with small packets on 100,000 TCP sockets could consume 25 % of cpu cycles o hosts with 44 queues with high tail latencies.

# CPU L1/L2 caches are too small for this model

Number of core/threads is increasing, but L1/L2 caches sizes are not changing.

Typical L1 are 32KB, and L2 are 256 KB.

When all cpus are potentially running kernel networking code paths, their caches are constantly overwritten by kernel text/data.

# Break the pipe !

Busy Polling next-gen would run some of the pipeline stages using dedicated and provisioned cores.

This is commonly used in NPU architectures (Network Processor Units).
Main difference here is that would be optional only.
( Most linux driven hosts would not use this mode )

We could more easily bound cpu cycles used by parts of IP/TCP/UDP stacks, and not interrupt anymore cpus that would run application code, with higher cpu caches utilization.

# Busy Polling CPU group(s)

We need to create groups of cpus, preferably per NUMA node.

Then attach cpus to groups or detach them.

It is probable we need cooperation from process scheduler, because we do not want these cpus being part of a normal scheduling domains.

The group would have the ability of parking cpus to low power mode, and activate them only on demand.

On low load, only one cpu per group would be busy polling.

# RX or TX path

We need to expose each NAPI in the system in sysfs, so that it can be added to a CPU group (or removed)

The operation would grab the NAPI_STATE_SCHED bit forever, automatically disabling the device interrupts.

Available cores in the CPU group would then service the NAPI poll.

Some device drivers use one NAPI per queue, handling both RX and TX. Others use separate NAPI structures (eg mlx4 driver)

# RX path

When driver napi->poll() is called from the Busy Poller group,

it would naturally handle incoming packets, delivering them to another queues, being sockets or a qdisc/device.

XDP, if enabled, would be transparently be handled.

No change should be needed in drivers.

Note that the present busy polling infra would continue to work, since the application would still spinning on its receive queue, or event poll queue.

# TX path

When a napi->poll() handles TX completions from Busy Polling group, we ideally would permanently grab qdisc->running (cf qdisc_run_begin() ). The dequeues from qdisc and calls to dev_queue_xmit() and ndo_start_xmit() would then no longer be done by application threads.

qdisc_run() is a **well known source of latencies**, as a victim thread (even a Real Time one) might be trapped in its loop, de-queueing packets queued by other applications.

# Challenges

Normal network stacks uses timers, RCU callbacks, work queues, software irqs in general. In particular, RFS could still be used.

Cpus in Busy Polling groups still need to service softirqs, and potentially yield to other threads.

They will be implemented as kernel threads, bounded to cpus.

To keep cpu caches really hot, we could dryrun code paths even if no packet has to be processed.