Busy Polling: Past, Present, Future

Eric Dumazet
Google
edumazet@google.com
netdev 2.1 Montreal 2017

Abstract

Traditional model for networking in linux and other unixes is based on interrupts being generated by devices, and being serviced from different layers depending on various constraints.

While this gave good results years ago when CPU resources were scarce, we very often hit pathological behaviors needing painful tuning.

Linux NAPI model added a generic layer helping both throughput and fairness among devices, at the cost of jitter.

Busy Polling was added in 2013 as an alternative model where user application thread was opportunistically going to poll the device, burning cycles and potentially avoiding the interrupts latencies.

As of today (linux-4.12), Busy Polling is still an application choice, that has practical limitations.

In this paper I will present Busy Polling history and its limitations.

Then I will propose ideas to shift busy polling decision in the hands of system administrators, allowing to precisely budget cpu cycles burnt by Busy Polling and allow for its wider use.

Busy Polling in a nutshell

Jesse Brandeburg presented initial ideas in LPC 2012[2].
Initial name was LLS (Low Latency Sockets)
Principal sources of latencies/jitter are :

- Scheduling / Context Switching
- Interrupt Moderation
- Interrupt Affinity
- Power Management
- Resources sharing (caches, bus)

By no longer waiting for device interrupts being generated/handled, and polling driver/device queues, we can avoid context switches, keep CPU in C0 state, and immediately react to packet arrival, on the proper cpu (regardless of CPU IRQ affinities)
Idea was to let the application thread calling a `recv()` system call or any other socket call that would normally have to wait for incoming messages directly call a new device driver method and pull packets. This would be done in a loop, bounded by a variable time budget.

Busy Polling History

Eliezer Tamir submitted first rounds of patches[11] for linux-3.11.

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()`.

Dmitry Kravkov added bnx2x support in linux-3.11

Amir Vadai added mlx4 support in linux-3.11

Hyong-Youb Kim added myri10ge support in linux-3.12

Sathya Perla added be2net support in linux-3.13

Jacob Keller added ixgbevf support in linux-3.13

Govindarajulu Varadarajan added enic support in linux-3.17

Alexandre Rames added sfc support in linux-3.17

Hariprasad Shenai added cxgb4 support in linux-4.0

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.

mlx4 for example had following defaults:

```
rx-usecs: 16
rx-frames: 44
```

TCP_RR (1 byte payload each way) on 10Gbit NIC would show 17500 transactions per second without busy polling, and 63000 with busy polling.

Even reducing rx-usecs to 1 and rx-frames to 1, we would only reach 37000 transactions per second.

Socket Interface

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 50 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.

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 software interrupts (BH) for the duration of the busy polling enabled system call. We might relax this limitation since we now have proper scheduling points in busy polling.

linux-4.5 changes

In linux-4.5, sk_busy_loop() was changed to let BH being serviced[5]. Main idea was that if we were burning cpu cycles, we could at the same time spend them for more useful stuff, 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.

Another step was to make ndo_busy_poll() optional, and use existing NAPI logic instead. mlx5[4] 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.

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 operation in the non-busy-polling fast path, which was unfortunate. The naive implementations first used a dedicated spinlock, then Alexander Duyck used a cmpxchg()[7]

linux-4.10 changes

In linux-4.10, we enabled busy polling for unconnected UDP sockets, in some cases.

We also changed napi_complete_done() to return a boolean as shown in Figure 1, allowing a driver to not rearm interrupts if busy polling is controlling NAPI logic.

linux-4.11 changes

In linux-4.11, we finally got rid of all ndo_busy_poll() implementations in drivers and in core[6].
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 applied) to 77000 TCP_RR transactions per second.

```
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;
```

Figure 1: Using napi_complete_done() return value

linux-4.12 changes

In linux-4.12, epoll()[9] 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).
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. It took years to come up to a core implementation and remove the leftovers, mostly because of lack of interest or time.

Another big problem is that Busy Polling was not really deployed in production, because it works well when having no more than one thread per NIC RX queue.

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

What's next

Paolo Abeni and Hannes Frederic Sowa presented[1] a patch introducing kthreads to handle a device NAPI poll in a tight loop, in an attempt to cope with softirq/ksoftirq defaults.

Zach Brown asked in an email[3] sent to netdev if there was a way to get NAPI poll all the time.

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 made me work again on Busy Polling (starting in linux-4.10 as mentioned above).

Various techniques are used to speed up networking stacks.

- RPS, RFS, XPS
- XDP
- special memory allocation paths
- page recycling in RX path.
- replacing table driven decisions by ePBF

But in all cases, we still have the traditional model where the current cpu, 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)
qdisc enqueue
qdisc dequeue (qdisc_run())
grab device lock.
dev_queue_xmit()
ndo_start_xmit()
roll back all the way down to user application
```

With SMP, we added many spinlocks and atomics in order to protect the data structures and communication channels, since any point in the kernel could be reached by any number of cpus. 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. Number of packets per hard interrupt decreased, and many 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 on hosts with 44 queues with high tail latencies.

It is worth noting that 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 all kernel networking code paths, their caches are constantly overwritten by kernel text/data.

Break the pipe !

General idea is to break the pipeline and run some of the 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 cache utilization.

Busy Polling CPU group

We need to create groups of cpus, preferably by their NUMA locality.

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 domain. 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)

Admins should be aware to attach properly all relevant NAPI.

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 current busy polling infra would continue to work, since the application would still spinning on its receive queue, or event poll queue, if could not grab `NAPI_STATE_SCHED`.

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 thread (even a Real Time one) might be trapped in its loop, dequeueing packets queued by other applications.

With Busy Polling, we could eventually always defer the doorbell, regardless of xmit_more[8] status, knowing that we will soon either provide another packet or send the doorbell after few empty rounds. This last part would require a change in a driver `ndo_start_xmit()`.

Challenges

Normal network stacks uses timers, RCU callbacks, work queues, software irqs in general. In particular, RFS[10] 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. For example, preallocating or touching the `napi->skb` could avoid some cache evictions.

Thanks

Many thanks to 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 for their work and ideas.

References

[1] Abeni, P. 2016. net: implement threaded-able napi poll loop support. https://patchwork.ozlabs.org/patch/620657/.

[2] Brandeburg, J. 2012. A way towards lower latency and jitter. Retrieved from http://www.linuxplumbersconf.org/2012/wp-content/uploads/2012/09/2012-lpc-Low-Latency-Sockets-slides-brandeburg.pdf.

[3] Brown, Z. 2016. Pure polling mode for netdevices. Archived at https://lkml.org/lkml/2016/10/21/784.

[4] Dumazet, E. 2015a. mlx5: add busy polling support. commit 7ae92ae588c9f78006c106bb3398d50274c5d7de.

[5] Dumazet, E. 2015b. net: allow bh servicing in sk_busy_loop(). commit 2a028ecb76497d05e5cd4e3e8b09d965cac2e3f1.

[6] Dumazet, E. 2017. net: remove support for per driver ndo_busy_poll(). commit 79e7fff47b7bb4124ef970a13eac4fdeddd1fc25.

[7] Duyck, A. 2014. ixgbe: Refactor busy poll socket code to address multiple issues. commit adc810900a703ee78fe88fd65e086d359fec04b2.

[8] Miller, D. S. 2014. Bulk network packet transmission. https://lwn.net/Articles/615238/.

[9] Samudrala, S. 2017. epoll: Add busy poll support to epoll with socket fds. commit bf3b9f6372c45b0fbf24d86b8794910d20170017.

[10] Scaling in the linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[11] Tamir, E. 2013. Merge branch 'll_poll'. commit 0a4db187a999c4a715bf56b8ab6c4705b524e4bb.