What's Happened to the World of Networking Hardware Offloads?

Jesse Brandeburg, Anjali Singhai Jain

Intel Corporation
Hillsboro, Oregon, USA
jesse.brandeburg@intel.com
anjali.singhai@intel.com

Abstract

Over the last 10 years the world has seen NICs go from single port, single netdev devices, to multi-port, hardware switching, CPU/NFP having, FPGA carrying, hundreds of attached netdevs providing, behemoths. This presentation will begin with an overview of the current state of filtering and scheduling, and the evolution of the kernel and networking hardware interfaces. (HINT: it's a bit of a jungle we've helped grow!) We'll summarize the different kinds of networking products available from different vendors, and show the workflows of how a user can use the network hardware offloads/accelerations available and where there are still gaps. Of particular interest to us is how to have a useful, generic hardware offload supporting infrastructure (with seamless software fallback!) within the kernel, and we'll explain the differences between deploying an eBPF program that can run in software, and one that can be offloaded by a programmable ASIC based NIC. We will discuss our analysis of the cost of an offload, and when it may not be a great idea to do so, as hardware offload is most useful when it achieves the desired speed and requires no special software (kernel changes.) Some other topics we will touch on: the programmability exposed by smart NICs is more than that of a data plane packet processing engine and hence any packet processing programming language such as eBPF or P4 will require certain extensions to take advantage of the device capabilities in a holistic way. We'll provide a look into the future and how we think our customers will use the interfaces we want to provide both from our hardware, and from the kernel. We will also go over the matrix of most important parameters that are shaping our hardware designs and why.

Introduction

The networking stack's support of hardware offloads has been built over time with each feature generally serving the needs of the moment, now we have several interfaces and multiple different hardware models, sometimes for the same feature. The feature space of offloads is only going to grow, and we (as a community) need to plan ahead and anticipate the growth to provide some structure and direction.

Evolution

How we got here

Back when networking was young, devices used to be one netdev, one external port. Things were easier and features of drivers included; one skb equals one packet on the wire. Since then, networking has evolved and the stack with it, usually one feature at a time. Development originally started with enabling hardware offload of some features of the stack, but often those features were fairly simple and stateless, particularly checksum offload of both transmit and receive, helped alleviate the CPU from having to burn cycles on every packet. Those features were generally expressed with a single offload flag programmed into the netdev->features (see Documentation/networking/netdev-features.txt.) After that we started moving on to more complicated offloads like Transmit Segmentation Offload (TSO.) The TSO feature was first committed back in linux-

2.5.33[1]. The TSO feature itself has been rewritten several times, yielding what we have today, but the segmentation offload has become more complicated as more features get added to the stack, including tunnel offloads as well as more protocols. These changes yield an ever more complicated Network Controller, more complicated validation plans, and lots more room for bugs.

Network Interface Controllers have continued the technology march, moving more complexity into the NIC, including offload of protocol stacks (like RDMA over Ethernet,) and offloading eBPF programs, which has led some vendors to provide fully flexible "Smart NICs".

Over time as the development has continued, each vendor adds a feature (small or large) to the kernel as hardware becomes available that can support it. That said, a lot of configurability and options for the user are often left un-implemented or implemented out-of-tree due to needing to provide a cross vendor generic implementation when the feature is upstreamed.

Once the pattern was established with each new feature being added a little bit at a time, it seems to the authors that we are just continuing along a path of "a little bit here, a little bit there" and we are not looking at the big picture or even just stating where we want to be in a few years.

The Benevolent Corporation?
For years, and pretty much still to this day, Microsoft specified everything having to do with hardware offload and then supported it in the OS interfaces. It could be argued that the only reason we have consistent stateless hardware offloads is because **some entity**, in this case Microsoft, was defining for networking vendors how everything should work, letting the vendor implement, and providing a check (certification) to make sure that the expectations documented in the specification were met. Microsoft did this for their own best interest, but the end result was a consistent hardware ecosystem that the Linux community (and others) benefitted from.

Current hardware

Basic network interface controllers (NICs)
A NIC that optionally provides basic stateless offloads and doesn't support SR-IOV. Generally, this kind of NIC only supports one or slower speeds. Definitely commodity, but the features from the Advanced NICs category below are always becoming "normal" and displacing NICs in this category.

Advanced NICs
This NIC definitely provides stateless offloads, possibly has support for multiple speeds and / or PHYs (physical layer,) has internal resources to support high speed networking; including larger FIFOs, stateless offloads, and interrupt moderation. Supports multiple queues for multi-core load balancing, yielding a big step forward in terms of scalability at higher speeds.

Switch capable NICs and drivers
This category builds on the above, but these have a switch in them, mostly to enable SR-IOV, but the usage cases for the switch continue to grow in this space.

Switching chips
These chips provide many ports of network connectivity with hardware connectivity between ports such that the data plane can be almost completely autonomous once set up. These are the kind of chips that are typically in a home router, or larger versions of them in chassis or Top-of-rack switches. They usually run an embedded OS. Today these are also sometimes run by switchdev drivers.

System on a chip (SoC / SmartNIC)
These NICs likely have CPU cores, local RAM (sometimes Gigabytes,) and sometimes full switch chips. These SoC ideas are not new, but they are somewhat new to the NIC world in the last couple years.

System on a chip with FPGA (SmartNIC + FPGA)
In addition to all the previously mentioned features, these NICs can have fully programmable FPGAs, giving the user lots of capability, but can take a lot of work to deploy, and is not typically useful to a single instance deployment.

Offloads and Workflows

How different are the same offloads from one vendor to another?
Every feature and offload that a NIC device supports can be tweaked with many configuration parameters. Not every vendor supports every possible customization when they support a given offload. There is nothing that binds them to provide all the knobs and there is no way for Linux as an operating system to impose a bare minimum or check the minimum.

---

1 commit 9d9cfb15585, Alexey Kuznetsov <kuznet@ms2.inr.ac.ru>, Date: Wed Aug 28 11:57:33 2002 -0700, **[NET]: Add segmentation offload support to TCP.**

In the following, we will summarize a few of the key features, and how every vendor can differ in their implementation.

Receive Side Scaling (RSS)
RSS is a mechanism to spread traffic to multiple receive queues using a hash over certain packet header fields. RSS can support the following configurations in the hardware depending on the NIC's capability:

- Number of RX queues that RSS can direct traffic towards
- The algorithm used for hashing
- The key used for the hash
- Number of hash buckets used (indirection table size)
- What packet header fields to hash together in order to get enough entropy
- Packet types that can be hashed
- Assigning queues to hash buckets
- RSS for pass-through VFs and their configurability

Depending on the NIC vendor, not all of the above is configurable from software and in some cases even if the hardware allows configurability, the software may not (or maybe cannot) expose all of it to the user, sometimes leading to very different results in terms of how much load balancing across queues is achieved for different types of traffic.

Large Receive Offload (LRO,) hardware GRO
Sometimes hardware LRO is known as Receive Side Coalescing aka RSC. LRO is a mechanism to coalesce, in hardware, several received packets from a flow (TCP/IP or UDP flow identified by a flow rule match) and then indicate that "super packet" to the software stack above the driver. The software stack can use less CPU cycles overall due to processing fewer packets. Configuration of LRO is done using the following configuration options (once again depending on NIC's capability):

- Size of largest coalesce
- Number of flows that can be coalesced in parallel
- Flow types that can be coalesced: TCP/IPV4, TCP/IPv6, UDP/IPv4, UDP/IPv6, etc.
- Mechanism to enable/disable LRO per Queue
- Mechanism to enable/disable LRO per flow
- LRO for pass-through VFs and their configurability

Vendors can differ in this space by modifying the above parameters, no one has the same default, and there is no larger specification in Linux saying how things should work.

Transmit Segmentation Offload (TSO aka LSO)
This is the reverse of LRO, but for transmit. Instead of the software breaking a packet (see Generic Segment Offload - GSO) into MTU sized segments, and sending each packet separately (header and data in one skb) to the driver to send it on the wire, hardware can do this segmentation with help from software. When implemented in a generic fashion in the hardware, TSO can be used to accomplish TCP segmentation offload and UDP segmentation offload for tunneled and non-tunneled packets. Depending on the NIC's implementation it may or may not support the following configurations for a segmentation offload:

- IPv6 options or a plethora of them
- TCP vs UDP segmentation offload
- Tunneled vs Non-tunneled packet segmentation offload, particularly if the outer headers need L4 checksum calculation in the hardware (Geneve and VxLAN)
- Parallel support for TSO Contexts (across the PF or chip)
- Ability to enable/disable TSO per queue
- Maximum size of a single descriptor
- Maximum number of descriptors
- Maximum bytes in a single offload

It should be noted that the Linux kernel networking stack has an API for drivers to 'opt-out' of offloading a particular packet, using the .ndo_features_check netdev op. This works ok, but is a relatively high-overhead thing to do for each and every packet, especially because there is no memory in the stack of the previous path for a packet that hit the exception for some reason.

Receive Checksum Offload
Receive checksum offload usually consists of hardware recognizing a packet that it should compute a checksum on, and indicating that checksum or the status of the checksum calculation (pass/fail) in the receive metadata for the packet. There is a lot of variability between vendors with respect to receive checksum offload. A device can either provide a checksum pass/fail for L3/L4 checksums (CHECKSUM_UNNECCESARY) or provide a raw checksum over the entire packet (CHECKSUM_COMPLETE) and then let the stack validate the packet checksum. Differences in support by vendor:

- Checksum validate vs deliver raw checksum

- Checksum for tunneled vs non-tunneled packets
- In case of pEdit offloads with tc, ability to fix the checksum of delivered packets

Transmit Checksum Offload
Transmit checksum offload usually consists of computing and inserting a protocol specific checksum at a particular offset within a packet. Hardware generally knows how to compute checksums only for a certain set of protocols, and likely will not know how or where to insert a checksum for every possible kind of packet that the stack might support (new protocols can be added too!)

Some of the flags exist in the kernel today for device drivers to communicate the limits of the hardware with respect to these offloads. The bad news is that even for a simple feature such as transmit checksum offload, the following is output from ethtool -k, demonstrating the complexity of expression for even this simple feature:

- tx-checksumming: on
- tx-checksum-ipv4: on
- tx-checksum-ip-generic: off [fixed]
- tx-checksum-ipv6: on
- tx-checksum-fcoe-crc: off [fixed]
- tx-checksum-sctp: on

Even in the case of transmit checksum offloads, drivers cannot easily specify all the existing offload capability of even this interface. The default ethtool interface doesn't cleanly express checksum offload support for tunneled protocols, like FOU[2], Geneve, VxLAN, STT[3], etc.

Flow Classification (aRFS, Flow director, ntuple rules)
There are many different ways that a NIC vendor can support flow classification offload, some are to achieve locality of where the packets get delivered to where the application is running for better performance for example:

aRFS - Adaptive Receive Flow Steering
Adaptive receive flow steering is a feature to allow the kernel to direct packets to a particular receive queue using hardware steering. The kernel uses the ntuple interface (see below) to program rules, noticing when the application's socket reads miss the receiving CPU, and programming new rules for each flow. Once again there is no clarity on how many flows can be steered this way by the NIC, the programming is best effort and the usage experience can vary quite drastically between vendors and even between hardware from the same vendor.

Ntuple - the rule programming interface used by ethtool
Ntuple rules from ethtool provides an interface that could potentially match on any field for a given flow type, and the matching fields could differ in terms of mask etc., without priority specified. The NIC vendors may implement this using CAMs or TCAMs, resulting in limitations of how flexible the match can be for a flow type. This interface to ethtool has gone through several iterations, giving users quite a bit of confusion about how they should use it. Gaps here include the inability to query how many rules can be programmed, and what kind of rules will work (users just have to try some to see if they will work.) One (admittedly unhappy) example for the Intel NICs is that ixgbe and i40e drivers have different programming interfaces (support different kind of matches) when using the ethtool interface because the underlying hardware is significantly different.

tc-u32 - match a 4-byte field anywhere in the packet and do an action, and others...
While several types of hardware support offloading u32 rules, there is a whole lot of software configurability in u32 including multiple tables, hash tables, and chained matches which are likely not fully implemented by anyone's hardware. What the user ends up with in this case is a vendor defined, extremely limited hardware offload implementation that might work, but only if you format and limit your rules for u32 to a very specific set. When the hardware offload is unavailable, a rule can be processed/executed by the software stack. Each vendor ends up implementing an "individual vendor" set of offloads, and may say that they support u32, but you can only find out if a particular rule would be offloaded by actually trying to program it on a NIC, or by reading and understanding a lot of driver code.

tc-flower - match a complicated flow specification, src-ip, dst-ip, protocol, src-port, dst-port, and do an action on the match
From the man page for tc-flower: The flower filter matches flows to the set of keys specified and assigns an arbitrarily chosen class ID to packets belonging to them. Additionally (or alternatively) an action from the generic action framework may be called. This interface is a vast

[2] "Foo over UDP [LWN.net]." 1 Oct. 2014, https://lwn.net/Articles/614348/. Accessed 4 Nov. 2018.

[3] "What is GENEVE? - Red Hat." 22 Jun. 2017, https://www.redhat.com/en/blog/what-geneve. Accessed 4 Nov. 2018.

improvement from what we had before, but even simple things like querying a piece of hardware to figure out what kind of tc-flower filters can be offloaded is not possible. The user is left either attempting to read driver code, or pouring over vendor provided manuals to find out if what they want to do can be offloaded. We've also ended up with workarounds like skip_hw and skip_sw directives in the interface to try to address these problems, which are a good idea, but are not automatic, obvious or programmatically discoverable.

Can users express a rule set for tc-flower (using skip_sw, which forces hardware offload of a rule,) pick those rules up from one NIC, and try to apply that same rule set to another NIC?

vSwitch Offload
vSwitch offload is advertised by many vendors, but there is no defined way to express what this means to the kernel, nor are there tools to describe and configure things that hardware can offload. vSwitch offload could mean a simple L2 offload, or could mean supporting overlay networks that get terminated in the hardware. vSwitch offload could mean there are meters and policers applied in hardware, per virtual port. It could mean flow tracking and flow eviction done by hardware. vSwitch offload could also mean a complete pass through using SR-IOV or just an assist in terms of Encap/Decap.

The point is a vendor can say they support vSwitch offload and no one really knows what it means.

Tunnel Offloads
Tunnel offloads are a particularly thorny area of implementation in the Linux kernel. There are several good summaries[4] of the problems in this space, but the takeaway is that examining capabilities and configuring the offloads around tunnel offloads is complex, ill defined, and implemented differently in every piece of hardware.

eBPF In Hardware vs Software
The extended Berkeley Packet Filter provides a nice Linux kernel and user space interface to filter packets and apply simple filtering programs to them, as well as take an action based on that program. Some companies have shown that offloading eBPF programs in hardware is possible, but it takes a very specialized NIC with some very hefty hardware behind it to be able to keep up with any decent rate of incoming packets (millions or more packets per second.) In the case of Intel hardware, the eBPF programs have a tendency to be hard for us to offload directly because we can't separate out from an eBPF program what is offloadable by our hardware, and what actions can/should be taken by our hardware. Also, see the paper/presentation from Linux Plumbers Networking Track 2018 from Waskiewicz, et al. on eBPF metadata.

Cost of an Offload
Many times, hardware vendors like Intel are forced to make business decisions (go / no-go) around perceived use and value of a feature vs the hardware cost to produce. Often an offload can be extremely expensive to implement in hardware. A good example of this is hardware based LRO. In order to implement this in hardware, the NIC must be able to track flows, track state, note ACK/FIN events, have reasonable timeouts for unclosed flows, have a large amount of resources dedicated to each flow (likely 16-64 bytes of hardware memory,), not add unnecessary latency, and not least of all, have no bugs in parsing or handling packets that require a spin of the silicon. Without some caution, offloads can also cause ossification if we as developers and a community are not constantly looking out for problems created by offloading workloads.

Massive Programmability of Smart NICs
With the advent of data-center based computing, many companies are buying large amounts of server machines, installing high speed networking and managing the entirety of their network under one schema. In this case the extra control and flexibility being offered by a SmartNIC[5] becomes desirable. These offloads and functionality are very useful to a consumer with resources to develop configurations for them, but the authors wonder what will happen when the data center configuration is not one size fits all (vendors.)

A Look to the Future

What's coming? It's here already. System on a Chip (SoC) implementations of network cards with CPUs, FPGAs, firmware, local storage and memory, and even internal OSes. Customers want a networking object installed in the

[4] "OVN – Geneve vs VXLAN, Does it Matter? | Russell Bryant." 30 May. 2017, https://blog.russellbryant.net/2017/05/30/ovn-geneve-vs-vxlan-does-it-matter/. Accessed 4 Nov. 2018.

[5] "Azure Accelerated Networking: SmartNICs in the Public ... - Microsoft." https://www.microsoft.com/en-us/research/uploads/prod/2018/03/Azure_SmartNIC_NSDI_2018.pdf. Accessed 4 Nov. 2018.

server that can allow renting **the whole server's CPU and system resources**, but still maintain some control of that machine's network from outside. We need a way to express this device's capabilities and control it.

The market has fragmented a bit, customers want three things from NICs with offloads, and they are somewhat orthogonal.

- A speeds and feeds (high speed with many ports) ASIC, but still want offloads and maybe even SR-IOV/Scalable-IOV.
- A full Switch on a NIC, with flow rules, visibility into switch config, port representors, etc., basically a NIC with switchdev enabled to represent all the virtual ports hosted by the hardware.
- A full SoC / computer running on my NIC, maybe even with data-plane independence, maybe including port to port forwarding and on-board flow management, via a control plane running locally to the NIC or coordinating with an external orchestrator.

Proposals

How do we get to a generic way to express offload capabilities? How can a user find and/or make a program to use them? We recommend defining and developing a common offload infrastructure, which would include a user-space library for programs to attach to and query, as well as the kernel implementation of the common pieces among vendors. Each implementation could be slightly different, but the method of defining each feature and what it does would be generic.

An interface much like the devlink dpipe[6] interface is possible, we envision something like this for hardware offloads is required. As it turns out, since this paper was proposed, devlink gained some useful code in this space, the 'devlink dev param' option. This is a possible way forward that needs investigation and we are curious if the community will support a lot more use of this option, but it does seem limited since devlink is aimed at configuring chip-wide options (its current granularity is associated with a PCIe device/function) and is possibly missing some functionality.

We also believe that the kernel can benefit from adding support for and separating configuration of the parsing pipeline (the kind of packets that can be filtered on) in the hardware from the actual filtering rules and actions. It should be noted that the P4 language[7] has the ability to both express the pipeline and programs that use that pipeline. We aren't encouraging the kernel to adopt P4, but it is worthwhile to know what other elements in the ecosystem are using so we can work on kernel compatible solutions. Currently if you use P4 to generate eBPF programs, it's not clear how the pipeline can be configured.

Conclusion

Every vendor is implementing more offloads and features to try to differentiate, and during the implementation of each feature and the associated software support, each vendor causes more fragmentation of the user interfaces as well as the user experience.

Sometimes today, you can even find a driver that says it supports feature X, but it's implementation of that feature could be limited or even completely different to the point that it doesn't work the same or even use the existing interfaces the same way as other vendor's drivers. We see this today when each vendor is having to create unique (to that vendor) documentation for how to use supposedly "common" interfaces. This causes a less than optimal user experience, we think we (as the community) can do better.

We believe our correct way forward is to figure out a generic way to advertise and show the current pipeline of a device, similar to devlink dpipe, and add more offload configuration in a very similar way to devlink dev param, but possibly with a better granularity of device specification, and with a more concrete plan for how the infrastructure and code should be developed going forward.

As a general method going forward, the authors also believe the community and kernel can benefit from asking for specifications / expectations of an interface to be written down and stored in the kernel as documentation, not just as code. If we as a community of reviews and

[6] "dpipe - NetDev conference." https://www.netdevconf.org/2.1/session.html?sharshevsky . Accessed 4 Nov. 2018.

[7] "P4 Language Specification - P4.org." https://p4.org/p4-spec/. Accessed 4 Nov. 2018.

maintainers ask for designs that not only include an initial implementation, but as well include some thinking in the relevant space about where the design can go in the future, it will clearly benefit all of us. As well, we believe there is benefit to implementing some sort of check (in zero-day tests or somewhere else like LTP) to make sure specifications are being met, and offloads are working as intended.

Acknowledgements