Linux Networking Offload Infrastructure

Tom Herbert, SiPanda

1. Introduction

Ubiquitous hardware offloads in Linux networking has been the dream almost from the very beginning. While there have been many efforts over the years for offloading various networking features to hardware, very few offloads can be called an unqualified success. For a long time this didn't matter, networking developers were able to "paper over the problem" through various optimizations in kernel software. Offloads, with the exception of some simpler ones like checksum offload, RSS, and TSO, were considered "nice to have" features relegated to a few specialized use cases. But today the landscape has changed. There's not a lot of low hanging fruit left to reap in the software stack, and thanks to the end of Moore's Law and Dennard Scaling we can no longer rely on increases in CPU performance for the stack to keep pace with ever increasing performance requirements of demanding workloads like AI/ML. Hence, there is new motivation, and even urgency, to make offloads succeed in the Linux networking stack.

The road for offloads in Linux has been bumpy to say the least. There is no overall design or infrastructure for Linux offloads, driver APIs tend to be ad hoc and verbose, and each offload seems to be designed in isolation of other offloads. Probably the biggest impediment is disconnects between kernel software and hardware devices. The base assumption of an offload is that the device is offloading functionality that is otherwise in software that runs on the host CPU, so even slight discrepancies in behavior between the kernel and the offload device can be problematic. This is especially true when discrepancies manifest themselves at rare edge conditions; for instance, a data center operator doesn't want to find out that an offload fails under extreme load on Cyber Monday which is typically the day of the year with the most load on the Internet. Discrepancies arise since an offload device is not part of the kernel and is basically a "back box". Without visibility into the inner workings of the device, the kernel has no way to establish that the offload device does what the kernel wants it to do. Blindly using such an offload can become a risk potentially liability to the data center.

In this paper, we propose a solution to the Linux offload conundrum. There are two parts to the solution: 1) A consistent and simple API between the kernel and device drivers for offloads, 2) Mechanisms for the kernel to get visibility and assurances that the offload device implements functionality in the same manner that the kernel does. These two parts are the basis for a networking offload infrastructure in Linux. A key enabler for ubiquitous offloads is the recent emergence of programmable devices.

2. Networking Offloads

In this section we provide an overview of networking offloads and describe the issues in creating useful offloads.

2.1 Domain Specific Accelerators

Offloads are a form of Domain Specific Accelerators for networking. *Domain Specific Accelerators*, (*DSAs*) are specialized hardware components that accelerate certain workloads within a specific domain or application.

Domain Specific Accelerators can be characterized by three basic operational models:

- Offloads
- Acceleration instructions
- Accelerator engines

For each model, we can consider the programming model, hardware interaction, benefits and disadvantages, as well as opportunities for development. The three models are complementary and can be combined to work in tandem in a hybrid accelerator system model as shown in Figure 1.

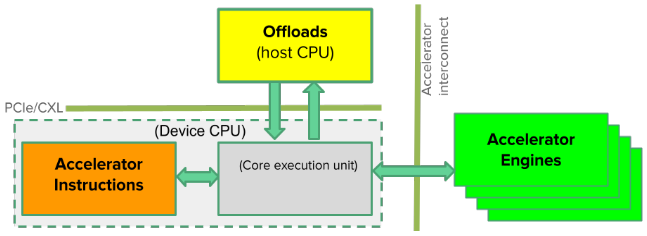

Figure 1: Example system architecture for accelerators. This diagram shows a hybrid accelerator model that employs offloads, accelerator instructions, and accelerator engines.

As suggested in Figure 1, offload is the host interface into a domain specific hardware device. The device may employ accelerator engines or specialized instructions to perform the backend offload processing. In the best case scenario, the offload device is programmable such that arbitrary offload functionality can be achieved by programming the device. Programmability also facilitates techniques for the kernel to be assured that the offload functionality matches that of the kernel; basically, this is accomplished if the offload device and kernel run the same program.

2.2 Offloads

Offload acceleration, or just *offloads*, is an early form of DSA. Almost every commercial NIC (Network Interface Card) in existence offers some form of offload. The basic idea is that functionality running in the host CPU is run in a hardware device instead. Use cases of offloads include checksum offload, TCP Segmentation Offload (TSO), Receive Side Coalescing (RSC), and TLS and TCP offload.

The programming model of offloads is a “tail call” from a program running in a host CPU to a device, or from the device to the host CPU. In the networking transmit path, instructions to request offloaded functions are specified in a transmit descriptor, the descriptor is then interpreted by the NIC and the requested offload processing is performed (for instance, computing and setting the TCP checksum). Receive offload works in reverse, the NIC autonomously performs offload processing on received packets and reports results in receive descriptors (for instance, the ones’ complement sum over the packet for verifying checksums).

The advantage of offloads is the simple programming model-- the details of an offload are easily abstracted from the programmer by a device driver.

A disadvantage of offloads is that there’s no means to leverage offload functionality outside of the networking path with any granularity. For instance, if a NIC has an encryption engine then it can only be used in the context of sending a packet, a CPU program can’t call the engine to encrypt an arbitrary block of host memory.

2.3 Offloads have been an underachiever

Deployment and ubiquity of offloads is underwhelming, and only a few simple offloads like checksum offload have been widely deployed. The impediment is that hardware offloads are only approximations of software functionality; even small discrepancies in functionality might lead to nondeterministic or incorrect behaviors. Discrepancies in functionality happen where there are disconnects between the kernel and the device or its driver.

2.4 API complexity in features

One challenge in offloads is expressing the offload functionality that a device supports. In Linux, supported offload functionality is expressed in a set of “netdev feature flags” for the device [8]. Even though there are only a few basic offloads, the number of feature flags for offloads has exploded due to so many variants of different offloads. In some regard, the current API for expressing offloads is a “mess” [11]. As an example, consider the features flags for indicating GSO or TSO support. Just for this one offload feature, there are nineteen feature flags:

NETIF_F_TSO, NETIF_F_GSO_ROBUST,
NETIF_F_TSO_ECN, NETIF_F_TSO_MANGLEID,
NETIF_F_TSO6, NETIF_F_FSO,
NETIF_F_GSO_GRE, NETIF_F_GSO_GRE_CSUM,
NETIF_F_GSO_IPXIP4, NETIF_F_GSO_IPXIP6,
NETIF_F_GSO_UDP_TUNNEL,
NETIF_F_GSO_UDP_TUNNEL_CSUM,
NETIF_F_GSO_PARTIAL,
NETIF_F_GSO_TUNNEL_REMCSUM,
NETIF_F_GSO_SCTP_BIT, NETIF_F_GSO_ESP,
NETIF_F_GSO_UDP, NETIF_F_GSO_UDP_L4,
NETIF_F_GSO_FRAGLIST

To make matters worse, there are several different sets of features flags: *features*, *vlan_features*, *hw_enc_features*, *mpls_features*. The cross product of the features bits and feature set gives a combinatorial number of possibilities. Note that this isn’t just TSO/GSO, other offloads have similar combinatorial properties in features.

2.5 Specifying offload requirements

One approach that could be considered is to specify offload functionality in normative requirements. This is what the “OCP NIC Core Features Specification” [1] endeavors to do. That document specifies requirements for checksum offload, segmentation offload, receive segment coalescing, and traffic shaping. While this is a noble effort, offloads don’t easily lend themselves to being defined by normative requirements. They are not protocols, and really not even something that could be standardized. Another problem is that to a large extent the offloads are being specified “after the fact”. Implementations have already deployed many of these offloads, so in some cases it’s more that the OCP specification is documenting what implementations already does rather than specifying forward looking requirements.

A good example of the difficulties of trying to specify requirements for offloads is in the requirements for Receive Segment Coalescing (RSC) in the “OCP NIC Core Features Specification”. That section describes Receive Segment Coalescing in normative requirements language. But for all the listed requirements, the documents ultimately states:

> *It takes the software Generic Receive Offload (GRO) in Linux v6.3 as ground truth. If the two disagree, that source code takes precedence.*

In other words, the source *is* the requirements document. Of course this is not unexpected, Receive Segment Coalescing offloads Generic Receive Offload so the expectation of the kernel is naturally that the offload provides the same functionality (i.e. no discrepancies between hardware and software functionality).

2.6 Buggy offloads

Because of the disconnects between software and hardware and the lack of visibility, networking offloads are prone to bugs. These are most likely to manifest themselves under edge conditions that may have not been well tested.

One area in which has seen a lot of bugs is *protocol specific checksum offload* [7]. In a protocol specific checksum offload, the device takes responsibility for computing and setting the TCP or UDP checksum on transmit, and reports that the TCP or UDP checksum is validated on receive. Protocol specific checksum offload requires that the device is able to parse and process specific network and transport layer protocols. Protocol specific checksum offload can be contrasted with *protocol agnostic checksum offload,* where the device doesn’t need to understand any specific network or transport protocols. On transmit, the device computes the ones' complement sum from some *csum_offset* start point in a packet through the end of the packet and sets the value at a *csum_offset* in the packet, and on receive the device just returns the ones' complement sum over the packet and the stack can use that to verify any number of transport layer checksums in a packet. Protocol agnostic checksum offload has long been the preferred method, and in fact protocol specific checksum offload is obsoleted per comments in skbuff.h [9], and [10] presents the arguments why protocol agnostic checksum offload is recommended.

Protocol specific checksum offload is prone to bugs since the device must parse packets on both transmit and receive. Problems occur when the device encounters a packet with protocols it doesn't understand. On receive, the most likely failure case is that the device is unable to validate a checksum and so the host must do it. However on transmit, if a device is unable to correctly parse the packet then the packet might be sent with an incorrect checksum and the packet will be dropped.

As an example, consider that a TCP packet is sent with an SRv6 routing header [2]. In this case, the destination IP address in the packet is not the final destination which is used in the pseudo header for computing the TCP or UDP checksum. If a packet with a router header is sent with transmit protocol specific checksum offload then an incorrect checksum is produced and the packet is dropped because the device uses the incorrect address in the pseudo header. This has been verified to happen with several NICs.

3. The Fundamental Offload Requirement

The problems with offloads motivates us to define the "fundamental requirement of offloads":

The functionality of a hardware offload must be ***exactly*** *the same as that in the CPU software being offloaded*

This is a generalization of the OCP requirement [1] for Receive Side Coalescing that the requirements implied by the software are the ground truth. Unlike the OCP case, this requirement is independent of any specific version of software. The requirements for an offload are derived from the software being offloaded at *runtime*. Additionally, this requirement isn't specific to just Receive Side Coalescing, but applies to *all offloads*.

While the fundamental requirement is easily stated, that begs the obvious question: How can this requirement be met? Our answer is:

Run the same code in the CPU and the offload target

Programmable devices are an enabler of this. We can infer an offload is viable if the same program code runs in both the host CPU and the device. Specifically, we would generate two images from the same base source code, one that runs in the CPU and one that runs in the offload device. An offload is considered viable if the program in the device matches that of the one being ran in the kernel.

4. Design for an offload infrastructure

In this section we present the design for an offload infrastructure in Linux. Requirements are:

- Move complexity out of core stack
- Assume programmable devices, compilers
- Simplify host/device interfaces
- Method for host CPU to query device to see what programs are supported
- Deal with resource limits in device

4.1 Moving complexity from the stack to devices

Our base design philosophy is that the core kernel stack offloads functionality *to the NIC device driver*.

In a receive offload the device needs to autonomously decide if an offload can be performed, this entails parsing and processing at least some of the received packet. The role of the driver is to read the offload results of a packet from the receive descriptor and convey them to stack via skbuff fields [9].

For a transmit offload, it's up to the device driver how to implement the required functionality of the requested offload. The driver may offload to the device, or may invoke a software helper function. The driver can make this determination based on whether the device has sufficient capabilities to perform the offload-- that is determined by inspecting the metadata in an skbuff or by parsing the packet if necessary. To assist drivers when processing cannot be offloaded, the core stack can provide helper functions. One example is *skb_checksum_help* that is called by several drivers when they determine that a device cannot properly offload the TCP or UDP checksum in a transmit packet.

By moving the impetus for determining whether an offload to a device is possible to the driver, we can greatly simplify the offload APIs and the core stack. For instance, the nineteen feature flags for TSO/GSO could conceivably be replaced by just one flag: NETIF_F_GSO. When a driver advertising that flag sees a packet marked with GSO, it can inspect the skbuff fields and perhaps parse the packet to determine if offload to the device is supported; if it's not then the driver would invoke helper functions for GSO.

4.2 Programmable devices

As mentioned in Section 3, the most direct way for the kernel to be assured that an offload is doing what it wants is to run the same program in both the kernel and the offload device. The idea is that a developer implements the offload functionality in generic source code (not hardware or software specific code). The code is then compiled twice-- once to run in the kernel, for instance in XDP/eBPF, and once to run in the hardware device, for instance into P4 backend binary. Each target image can then be loaded and run in their perspective environments (Figure 2).

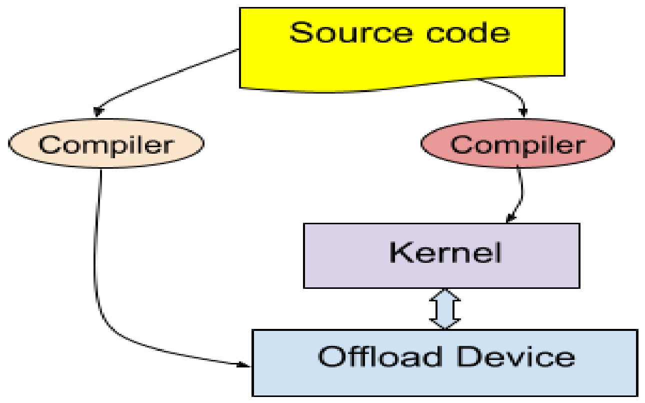

Figure 2: Offloading from a common program. A common source program is compiled twice: once for running in the kernel (or host), and once for running in an offload hardware device.

4.3 Querying offload supported

When both the kernel and offload device are running the same program then the offload is viable. To establish this equivalency, the kernel queries the device about what programs have been loaded. If a loaded program is the same as that used in the kernel then the offload is allowed.

To check for equivalent functionality, we propose that programs are identified by a hash of the original source code or an intermediate representation (IR). As part of the compiling process, a hash, SHA-1 for instance, is computed over the input source code or IR. The resultant hash value is saved in each target image.

When the kernel wishes to offload some functionality, it queries the driver for what programs have been loaded for the function. The driver either queries the device or consults a table that lists each of the programs that are loaded in the device. The driver returns the hash value that is attached to the program image. When a hash is returned, the kernel compares the value to that of the one that is running in the kernel (for instance, the hash associated with a loaded eBPF program). If the hashes exactly match, then the kernel can assume that the device offloads exactly the same functionality running in the kernel.

4.4 Resources in an offload device

The hash method is sufficient to validate that an offload device implements the same functionality as the kernel, however we also need to consider the available resources of an offload device.

In many cases, offload devices are resource limited. A device will often have fewer resources than the the host. This is especially true in local memory as a device may have orders of magnitude less memory. When the kernel is managing an offload, it needs to take resource limits into account. We especially need to consider stateful offloads, or offloads that involve table lookups-- the offload device may support fewer table entries or state entries.

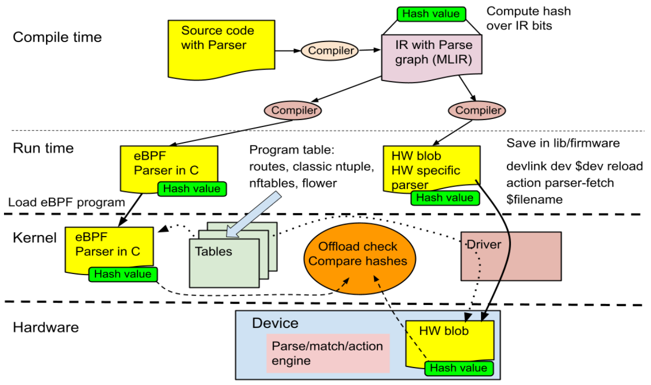
Compile time
Source code with Parser
Compiler
Hash value
IR with Parse graph (MLIR)
Compute hash over IR bits
Compiler
Compiler
Run time
eBPF Parser in C
Hash value
Program table: routes, classic ntuple, nftables, flower
HW blob HW specific parser
Hash value
Save in lib/firmware
devlink dev $dev reload action parser-fetch $filename
Load eBPF program
Kernel
eBPF Parser in C
Hash value
Tables
Offload check Compare hashes
Driver
Hardware
Device
Parse/match/action engine
HW blob
Hash value

Figure 3: Offload procedures. This diagram shows the procedures for offloading some functionality to a device.

4.5 Offload procedures

The procedures for a network offload are (Figure 3) [6]:

1. User compiles source code with parser into intermediate representation (IR) (Figure 4)
   - IR could be LLVM IR with parser in MLIR using a "parser dialect"
   - IR for parser could be Common Parser Representation (.json)
2. Compute hash, SHA1, over the IR bits. Save with the IR file
3. Invoke backend compilers to compile IR into targets. Attach hash to each target image file
   - Compile to executable for running in kernel (e.g. to XDP/eBPF)
   - Hardware "blob" for executing in hardware target
4. User puts the HW blob in /lib/firmware
5. devlink dev $dev reload action parser-fetch $filename
6. devlink loads the file, parses it and passes the blob to the driver
   - driver/fw reinitializes the HW parser
   - User can inspect the graph by dumping the common parser representation
7. The parser tables are annotated with Linux offload targets (routes, classic ntuple, nftables, flower etc.)
8. ethtool ntuple is extended to support insertion of arbitrary rules into the "raw" tables
9. To enable an offload, kernel compares the hash of the program loaded in the kernel to that of the program in the device
   - ○ Call ndo function to request loaded programs and their hashes from driver
   - ○ If hashes are equal then offload is permitted

5. Retrofitting the five basic offloads

In this section we discuss work for retrofitting the five most basic offloads for the Linux Offload Infrastructure.

5.1 Transmit checksum

The goal of transmit checksum work is to remove protocol specific checksum offload from the core stack. Remove features NETIF_F_IPCSUM, and NETIF_F_IPV6CSUM, and set NETIF_F_HWCSUM in all drivers supporting checksum offload. For a legacy device that only supports protocol specific checksum offload, the driver can inspect each packet to determine if the checksum is offloadable to its device; if it is not offloadable then the driver can call *skb_checksum_help* to compute the checksum.

There are few patches needed to support this:

- Prerequisites patch sets
    - drivers: Fix drivers doing TX csum offload with EH (*ipv6_skip_exthdr_no_rthdr*)
    - crc-offload: Split RX CRC offload from csum offload
    - Flow dissector: Parse into UDP encapsulations
- Convert drivers to NETIF_F_HWCSUM
    - Add helper function: *skb_csum_hwoffload_legacy_check*
    - Fairly minor change to most drivers

5.2 Receive checksum

The goal of receive checksum work is to remove protocol specific checksum offload for the core stack. This entails removing the feature flag CHECKSUM_UNNECESSARY and having all drivers just use CHECKSUM_COMPLETE to report an offloaded checksum. For legacy devices that only support protocol specific checksum offload, a helper function is provided that does a generic *csum-unnecessary* to *csum-complete* conversion.

Patches to support this include:

- Convert drivers to CHECKSUM_UNNECESSARY
    - Add helper function: *skb_csum_rx_legacy_convert_unnecessary*
    - Change legacy drivers to call *skb_csum_rx_legacy_convert_unnecessary*
    - Fairly minor change to most drivers

5.3 RSS (and aRFS)

We can consider RSS [12] to be an offload of Receive Packet Steering (RPS), and similarly aRFS is an offload of Receive Flow Steering [3]. For these, the biggest need is a programmable parser (in both software and hardware) [4].

The parser that underlies RPS is flow dissector. Flow dissector is not programmable, however we propose moving flow dissector to eBPF in [5].

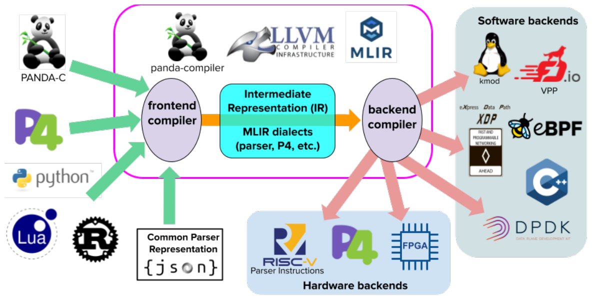

Figure 4: Compiling for offloads. A common source code can be compiled to different hardware and software targets.

Given a programmable flow dissector, we can then offload it using the techniques described in section 4.6.

5.4 TSO (GSO)

TCP Segmentation Offload (TSO) is an offload of Generic Segmentation Offload (GSO). Similar to offloading flow dissector, the first step would be to rewrite the GSO logic in eBPF. That establishes the standalone program that can be offloaded to a device using techniques in Section 4.

The GSO operation does not require parsing the packet. In order to maintain this property, the GSO related fields in an skbuff could be conveyed to the device in a transmit descriptor (*gso_size*, *gso_segs*, *gso_type*, etc.).

For legacy devices, a helper function *skb_gso_helper* could be created. If a driver sees a packet for GSO offload that cannot be offloaded to the device, then the driver can call the helper function to perform the GSO operation.

A side effect of this design is that most of the NETIF_F_GSO_* flags can be eliminated and just one flag, NETIF_F_GSO, would be sufficient.

5.5 RSC (GRO)

Receive Side Coalescing (RSC) is an offload of Generic Segmentation Offload. Similar to GSO, for offloading GRO the kernel implementation of GRO could be rewritten in eBPF, and that could serve as the common program that is offloaded to a device.

In its nature, RSC and GRO require packets to be parsed. Parsing could be done by invoking a programmable parser (either in the hardware device or a programmable flow dissector in software).

6. Conclusion and status

Implementing the Linux Offload Infrastructure is a fairly significant effort. The full project can be broken down into smaller milestones. Initially, efforts will be focused on checksum offload and fixing bugs in that. Subsequently, replacing flow dissector with eBPF is probably the next milestone, followed by support for RSS, TSO, and GRO.

Once the basic offloads are supported by the infrastructure, then more advanced offloads can be supported including TC flower offload, P4-TC offload, and TCP offload.

REFERENCES

[1] Dan Daly, Jakub Kicinski, and Willem de Brujin. August 9, 2023. Open Compute Project – NIC Core Features Specificiation. *Open Compute* Project. https://www.opencompute.org/documents/ocp-server-nic-core-features-specification-ocp-spec-format-1-1-pdf

[2] Filsfils, C., Ed., Dukes, D., Ed., Previdi, S., Leddy, J., Matsushima, S., and D. Voyer, "IPv6 Segment Routing Header (SRH)", RFC 8754, DOI 10.17487/RFC8754, March 2020, https://www.rfc-editor.org/info/rfc8754

[3] Tom Herbert and Willem de Brujin. May 2014. Scaling in the Linux Stack. *Linux Kernel Documentation.* https://docs.kernel.org/networking/scaling.html

[4] Tom Herbert, Pratyush Khan, and Aravind Buduri. 2022. High Performance Programmable Parser. Slides from *Netdev 0x16* conference. Slides 9-17. https://netdevconf.info/0x16/papers/11/High%20Performance%20Programmable%20Parsers.pdf

[5] Tom Herbert and Pedro Tammela. May 2021. Replacing Flow Dissector with PANDA Parser. *Netdev* 0x15. Scaling in the Linux Stack. https://netdevconf.info/0x15/slides/16/Flow%20dissector_PANDA%20parser.pdf

[6] Jakub Kicinski, Extracted form email sent to Netdev list with subject "[PATCH net-next v16 00/15] Introducing P4TC (series 1)", June 11, 2024.

[7] Linux kernel documentation. Checksum offloads. https://docs.kernel.org/networking/checksum-offloads.html

[8] Linux kernel netdev_feautures.h. https://github.com/torvalds/linux/blob/master/include/linux/netdev_features.h

[9] Linux kernel skbuff.h. https://github.com/torvalds/linux/blob/master/include/linux/skbuff.h

[10] David S. Miller. November 2015. Hardware Checksumming: Less is More. *Netdev 1.1 conference*. Scaling in the Linux Stack. *Linux Kernel Documentation.* https://www.netdevconf.org/1.1/keynote-hardware-checksumming-less-more-david-s-miller.html

[11] Michal Miroslaw. May 2014. Netdev features mess and how to get out from it alive. *Linux Kernel Documentation.* https://docs.kernel.org/networking/netdev-features.html

[12] A. Viviano. 2023. Introduction to Receive Side Scaling. Microsoft. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling