# Resource Management for Hardware Accelerated Linux Kernel Network Functions

**Andy Roulin, Shrijeet Mukherjee, David Ahern, Roopa Prabhu**
Cumulus Networks
{aroulin, shm, dsa, roopa}@cumulusnetworks.com

**Abstract**

The Linux kernel supports offloading of networking functions such as bridging and routing to switches (and NICs). These offloads occupy hardware resources that are difficult to track for the kernel as their representations can differ from kernel resources. In addition, hardware resources must be often traded off against each other if they are stored in the same shared memory.

Failure to offload a kernel resource, e.g., a routing entry, installed in the kernel puts the kernel and hardware device out of sync which can result in incorrect behavior, e.g., blackholed packets. Reverting to a software implementation is not always possible, especially if line rate processing of packets is a hard requirement of the network environment.

This paper presents the challenges of resource management involved in network function acceleration. We present models to manage complex resources offloads where kernel/hardware representation do not match. Resource availability is checked synchronously on the offload path to predict with a high degree of confidence that an offload will succeed while also providing synchronous feedback to users, e.g., routing daemons.

**Keywords**

Linux, ASIC, switch, offload.

## Introduction

The Linux kernel supports accelerating network functions by offloading to hardware. Typically, stateless decisions are offloaded, and more recently enhancements like FDB offload, FIB offload and eBPF offload support have been added. The situation however gets a little more complicated when offloading functions that affect multiple interfaces and need to be optimized and managed across these multiple interfaces. Further complications arise from mismatch in the view of resources between the kernel and hardware/driver. When offloaded to hardware, a single kernel resource may occupy more than one hardware resource: e.g. routing prefixes which are a complete entry in the kernel might be stored in TCAM while next-hops could be stored in a hash-based memory.

One important implementation challenge of hardware offloading is to keep the kernel state in sync with hardware when a network function cannot be offloaded due to lack of hardware resources. Let us consider the L3 routing case and how hardware offload failures can cause catastrophic problems. Failure to indicate a hardware offload error to a routing daemon, will result in the routing daemon advertising reachability of the failed prefix, which will result in peer routers sending traffic towards it, which will then get blackholed. The device driver may try to remedy the situation by disabling offloading and revert to software forwarding but this solution is not always practical and can actually even be harmful if forwarding packets at line rate is a strict requirement of the network infrastructure. A data center switch such as the Mellanox Spectrum ASIC can forward packets at 6.4Tb/s while software forwarding on the same ASIC delivers around 4Gb/s [1]. With such a mismatch in speeds many packets will be lost as the CPU cannot handle line rate traffic.

As said previously, hardware offloading failures can also result in incorrect route advertisements and potentially blackholing of traffic. Consider the network topology shown on Figure 1 with 3 routers and a Switch ASIC (DUT) acting as an additional router. All routers running BGP. Suppose the hardware FIB on the switch has a default route, e.g., 0.0.0.0/0, for packets and a new route, 13.0.0.0/24, is installed but its offloading fails without notifying the kernel or the routing daemon (assume no fallback to software forwarding). Packets for 13.0.0.0/24 will still be forwarded in hardware but will take the default route instead of the proper next-hop which will result in incorrect routing and blackholed packets. Details about this experiment can be found on github [9].

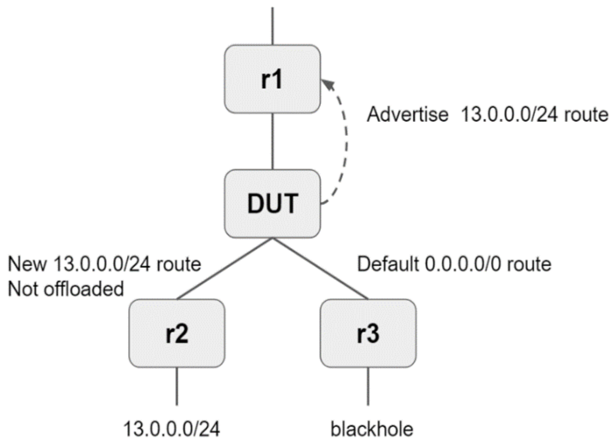


Figure 1: Lost packets if a route is not offloaded

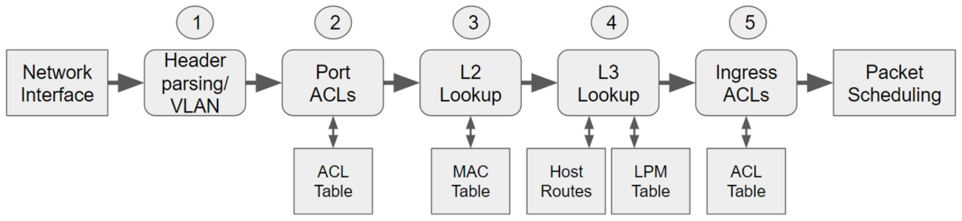


Figure 2: Switch ASIC Ingress Pipeline

Hardware implementation can differ drastically from the software implementation both in terms of processing and resource management. For example, IPv6/64 and IPv6/128 prefixes (routes with a /64 mask versus a /128 mask) typically use different hardware entries while the kernel does not distinguish between them. Hardware representation mismatches make it harder for the kernel to keep a precise and simple accounting of hardware resource occupancy. This is further complicated as different kernel resources can be mapped to the same shared hardware resources: e.g., if both routes and neighbor entries use the same shared memory on a device then adding more routes can decrease the number of available neighbor entries. This leads to problems where an IPv6 route add could result in decrementing the IPv4 address pool, which is an unexpected result.

Throughout this paper we use examples of Linux kernel networking functions offloaded to a switching ASIC. The main network function used to explain the problems and solutions is L3 routing. However, the same principles and problems apply for other network functions as well as other devices such as NICs.

The first three sections provide background information on switch ASIC pipelines, Linux kernel hardware acceleration and the current approaches to manage resources in the kernel. We then describe possible resource managers design to improve user feedback and avoid unhandled resource overload.

## Switch ASIC Pipeline Resources

Switch ASICs have different pipelines and resources, however main components can be abstracted as shown on Figure 2 and Figure 3 (showing respectively a packet's ingress and egress pipelines).

We discuss first the ingress path (Figure 2) with a focus on hardware resources used:

- **Port ACLs (2):** filtering of packets before forwarding, based on source port and VLAN; ACLs usually stored in TCAM.
- **MAC Address Table (3)**: exact-match table for (VLAN, destination MAC) tuples. It decides if a packet must be routed or bridged. In case of bridging, the table contains the egress physical port; usually stored in hash-based memory. Packets that match the "router MAC" are deemed to be routed.
- **Host Routes Table (4)**: Exact match for (VRF, IP) tuples. Given a destination IP address, it provides the directly connected port and next-hops which are /32 routes (or /128 for IPv6); usually stored in TCAM or hash-based memory.
- **LPM Table (4)**: Longest-Prefix Match on IP addresses for (VRF, destination IP). It is used mainly for remote routes as well as connected routes that have not been resolved and placed in the host table; usually stored in hash-based memory on modern silicon implementations.
- **Ingress ACLs (5)**: filtering of packets after forwarding, based on L2/L3/L4 fields; ACLs usually stored in TCAM

After having been buffered and scheduled, a packet goes through the egress pipeline, shown in Figure 3. The packet's header is being rewritten with new needed fields, e.g., MAC address, VLAN id might be added and egress ACLs are applied as egress filtering. Egress ACLs are stored usually in TCAM as well. Some ASIC architectures manage all different ACLs attach points (port, ingress and egress) through the same shared hardware table and thus induce tradeoffs on the overall ACL capacity.

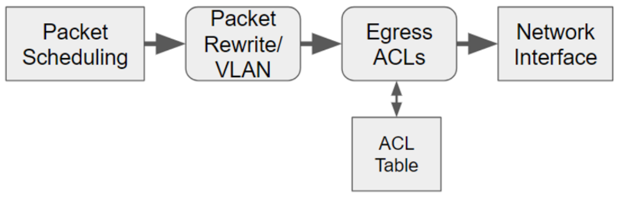


Figure 3: Switch ASIC Egress Pipeline

### Shared Hardware Resources

Switch ASICs resources are often shared between different types of objects. For example, the host routes table and MAC address table might use the same underlying memory, typically implemented with some form of a hash table. The entries and keys that use the table are defined in blocks that can be resized and thus allow routes versus mac capacity to be traded off against each other. Furthermore, different types of host route entries and LPM entries exist (e.g., IPv4 entries, IPv6/64 and IPv6/128 entries) which can also share the same memory and be traded off against each other.

## Linux Kernel Hardware Acceleration

This section provides a survey of network function elements currently being offloaded to a switchdev driver in the Linux kernel. Resources used in the offload are described as well as the current resource management scheme and the impacts of hardware resource overload.

## Address Lists offloaded to NIC e-switch

Unicast and multicast address lists can be offloaded to SR-IOV enabled NICs, e.g., ixgbe, for filtering and switching in hardware. MAC addresses are added through the netdev op ndo_fdb_add. Address lists can be directly offloaded to hardware without adding to the kernel (kernel bypass via NTF_SELF).

Drivers check the hardware limits before programming the address to the hardware and returns -ENOMEM when no resources are available and fall back to software instead. The cpu complexes in systems on which this is deployed are usually capable of handling line rate traffic therefore falling back to software is a viable option.

## L2 Bridging

Offloading of FDB and MDB entries enables switching of packets at line rate based on destination MAC addresses. If learning is done in hardware, hardware learnt entries are pushed to kernel FDB. Software learnt or managed FDB entries are offloaded to hardware through via switchdev notifiers.

In case of failure due to the absence of an FDB/MDB entry, hardware or kernel will flood the packet to all ports in the broadcast domain and functionality is still preserved. While this preserves functionality flooding is suboptimal in a L2 domain as it takes up unnecessary network bandwidth and CPU cycles.

## L3 Routing

Hardware drivers learn of changes to Layer 3 configuration via notifiers. For example, when a network interface is enslaved to a VRF, a netdev notifier is called or if an address is added or removed on a network interface an address notifier is called. The driver also uses that notifier to check on the fly if any additional hardware resources are needed with these interfaces. For example, a RIF might need to be allocated in hardware or a virtual router id needs to be assigned for a VRF. If the maximum number of such resources has been reached, the driver can synchronously fail the request and the error is returned to the user.

Similarly, switchdev drivers learn of changes to kernel FIB entries (adds, deletes, and modifications) via FIB notifiers. Because of the potential overhead in offloading FIB entries, the work needed to program the change in hardware is done asynchronously from the user request (e.g., via a work queue). This means the change is made to the kernel FIB, and assuming no errors adding the work to the queue, the kernel returns success to the user which can continue with more FIB changes. Programming the hardware is done later which means if hardware capacity is exceeded there is no way to pass an error back to the user. Since the FIB entry exists in the kernel but not in hardware, the two forwarding paths are now out of sync. One response to such overload scenarios is to abort all FIB offloads and fall back to software based forwarding. As described earlier this situation simply does not always result in acceptable system behavior.

## ACL Offload

ACL's are rules that specify whether traffic matching the criteria is forwarded or discarded. In Linux ACL's can be configured via netfilter or tc. An API to offload tc rules to hardware is a netdev operation named ndo_setup_tc. Drivers implement this netdev operation if they can support tc rule offload. In most cases hardware is not capable of offloading all ACL rules that software can support. For example, the driver can choose to offload flower and u32 classifiers and not the many others that tc can support in software. The Linux kernel does not fail an acl offload if the hardware does not support the ACL today. For a switch ASIC this means, on an ACL offload failure, the ACL is active only on the packets punted to the CPU and not on the packets forwarded in hardware resulting in inconsistent and unpredictable ACL enforcement. In environments where strict security policies are audited and signed off on, this would be a violation.

## Offload Policies

Having seen the previous examples, offload policies in the kernel can be generally classified in three different models:

- **Sync Model:** hardware and kernel state are kept in sync. In case of insufficient hardware resources, fall back entirely to software, e.g., routing, ACL's

- **Bypass Model:** hardware offload is managed separately from the kernel and skips the kernel, e.g., address filters on NICs.

- **Hybrid Model:** Partial offload to hardware either through user flags (skip_sw, skip_hw) or when no more hardware resources are available, fall back to software for additional entries, e.g., ACLs.

This classification is not consistently respected in the kernel and each network function develops its own custom resource management. It also sometimes differs between drivers for the same function.

While each of the solutions and approaches works for building solutions to specific problems, there is no consistent platform expectation that a userspace process can have that spans all solutions.

## Approaches for Resource Management

### Driver Resource Profiles

Driver resource profiles partition the available memories in advance between the different hardware entries. The partitioning guarantees a given number of entries (e.g., 50K MAC addresses) and is thus conservative: the unused space for MAC addresses cannot be used for, e.g., more IPv4 routes. Moreover, resource overruns are left as an exercise for the consumer of those resources and there is no simple mechanism that prevents a user space daemon from oversubscribing. As we have noted in the earlier sections, this can lead to catastrophic and undetectable networking failures

### Simple accounting for Dedicated Resources

Resources like RIFs or VRFs have static limits where the device has guaranteed support for a specific number of such resources. They are more easily tracked for example using simple counters and allow for synchronous failures when the limit is reached.

### Try and Abort for Complex Resources

FIB entries, MAC entries and next-hops share underlying memories and the kernel representation can be different from the underlying device representation (e..g, IPv6/64 and IPv6/128 addresses use different hardware entries). If driver resource profile is not used for these resources (ie., guaranteed limits) and with no means for querying current resource utilization, userspace has no way to know if FIB changes need to be limited. A routing daemon can only push the request to the kernel and hope for the best -- that the change does not exceed hardware capacity and trigger an offload abort.

## Problem Definition

The introduction and background sections motivated the need for:

- a clear offload failure error path to the user or protocol daemon;
- better communication between kernel and drivers/hardware to account for resource usage and availability.
- A resource manager model or resource management algorithm for drivers. A resource manager allows shared memories to be used more efficiently and with more flexibility than the strict partitioning of resources used in driver profiles.

We discuss in the rest of the paper different models for drivers to manage hardware resources. The main goal of these models is to fail synchronously to the protocol daemons in case of insufficient resources.

Considering the L3 lookup case again, three main loops are involved in a typical workflow as shown in Figure 4. Loop (1) represents users, e.g., routing protocol daemons, adding routes. The kernel checks with the device driver that the device still has sufficient resources for a route in Loop (2), one route at a time. Resources are only reserved at that point as drivers typically defer the actual offload to the device through a work queue for latency reasons. Loop (3) represents the driver reserving hardware resources through communication with the device (PCIe reads/writes).

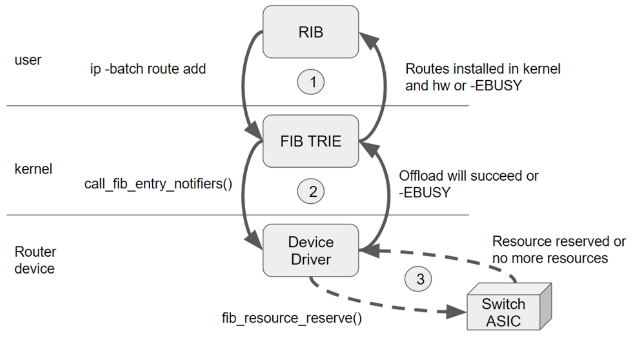

user
ip -batch route add
RIB
1
Routes installed in kernel and hw or -EBUSY
kernel
call_fib_entry_notifiers()
FIB TRIE
2
Offload will succeed or -EBUSY
Router device
Device Driver
3
Resource reserved or no more resources
fib_resource_reserve()
Switch ASIC

Figure 4: Route Add Workflow/Error Path

Loop (1) and (2) are synchronous, (1) representing the batch add of routes through rtnetlink while (2) is reserving resources needed one route at a time. Loop (3) can take three different forms:

1. synchronous, install route synchronously: **lockstep implementation**
2. synchronous, prefetch resources for future routes: **synchronous prefetching**
3. asynchronous, prefetch resources for future routes: **credit-based implementation**

The synchronous error flow in case of insufficient resources is described in more details below:

**Loop (3)**: If there are not enough hardware resources to offload a kernel resource, the device firmware will return a driver-specific error to the device driver. Reserving resources for an IPv4 route can fail, e.g., because there is no more IPv4 prefix entries available on the device.
**Loop (2)**: Reserving hardware resources for a given offload thus fails and the driver returns -EBUSY to the kernel core. In the routing case, *fib_table_insert* cannot reserve needed hardware resources on devices and does not install the route in the kernel. Routes that have been previously offloaded as part of the same rtnetlink batch are removed for consistency.
**Loop (1):** Routing daemons receive the errno indicating no more resources available on the device. If a batch add fails, the entire batch fails and no resources are actually offloaded.

An alternative solution that gets rid of Loop (3) is presented as the **predictive solution**.

### Lockstep Implementation

In the lockstep solution, routes are programmed synchronously to the hardware when right after they are added to the kernel. This solution introduces latency (PCIe reads/writes to configure resources in the hardware) on the control path for installing a route. We measured the latency on a Mellanox Spectrum ASIC (SN2100) to be **50-55µs** for install of an IPv4 route with a single nexthop. When installing many routes, this additional latency increases route convergence time, e.g., if a link goes down and routes need to be recomputed and added to the kernel and hardware.

In case of offload failure, routes are removed from the kernel to keep hardware/kernel states in sync and an error is returned synchronously to the user.

```
inet_rtm_newroute() // called via rtnetlink
        fib_table_insert()
               router_fib4_insert()
               // program hardware with new route
              // fail synchronously in case of offload failure
```

Listing 1: Lockstep Workflow

Patch to the spectrum driver used to measure the latency when installing a route inline can be found on github [7].

### Synchronous Prefetching

With synchronous prefetching, the driver reserves N hardware route resources in advance. The next (N-1) offloads will be able to reserve routes and return immediately without any hardware access. Routes are installed to hardware asynchronously (route install is deferred using a work queue) but the offload will succeed as resources have been reserved.

The critical path of installing a route still incurs the additional I/O latency every N routes (worst-case scenario) which can still impact route convergence time. Thus, the prefetching should be removed from the control path which we discuss in the next section.

```
inet_rtm_newroute() // called via rtnetlink
        fib_table_insert()
               router_fib4_prefetch()
              // prefetches synchronously new
              // resources if needed, returns immediately o/w
```

Listing 2: Synchronous Prefetching Workflow

### Credit-based Solution

The credit-based solution hides the latency of the synchronous solution while still providing an immediate feedback to the user. The driver periodically refills, on the side, a "bucket" of N route resources. This allows for modelling the hardware's capacity and rate of intake accurately which is yet another error that can happen on the hardware front. The size of the bucket can be adaptive: starting with a bigger bucket size and progressively reducing its size as hardware resources decrease. The driver refills the bucket asynchronously when it reaches a low threshold such that it can always reply synchronously and without any involving hardware access to a route add request.

```
inet_rtm_newroute() // called via rtnetlink
        fib_table_insert()
               router_fib4_reserve()
               // triggers async refill if bucket reaches
               // low threshold

router_fib4_refill()
              // refills asynchronously bucket with new
              // reserved resources
```

Listing 3: Credit-based Workflow

We simulated this implementation (hardware accesses to reserve resources are simulated) with the Mellanox switchdev driver (mlxsw) to show that the additional latency from the lockstep implementation is hidden. The source code for the credit-based solution can be found on github [8].

## Comparing Lockstep & Credit-based

We compared the performance of the lockstep implementation and the credit-based solution for FIB updates using two metrics: coherent control plane latency and dataplane latency. We used a Mellanox SN2100 Open Ethernet Switch (Spectrum ASIC) with net-next kernel. The Spectrum driver was modified to install routes inline (lockstep implementation) and alternatively to use a bucket refill system to hide the lockstep latency (credit-based solution). The experimental setup is shown in Figure 5.

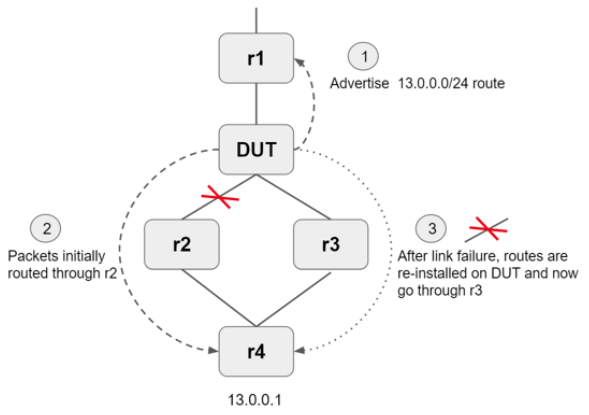

r1
1
Advertise 13.0.0.0/24 route
DUT
2
Packets initially routed through r2
r2
r3
3
After link failure, routes are re-installed on DUT and now go through r3
r4
13.0.0.1

Figure 5: Setup for 10,000 routes being updated from FRR to hardware

The coherent control plane latency represents the round-trip time (from userspace and back to userspace) to install new routes in the kernel. It is coherent as it includes the time to update hardware tables as well in order for the kernel and hardware to be in sync at the end of the updates.

We measured coherent control plane latency using the Unix *time* command (focusing on system time) when installing 10,000 routes with *ip -batch add*. We checked that the results are coherent using perf tracepoints measuring hardware access latency. For the lockstep implementation, we measured the control plane latency to be **600ms**, which gives a latency per route of **60us**; the credit-based system install the 10,000 routes in **480ms,** giving a speedup of **1.25**.

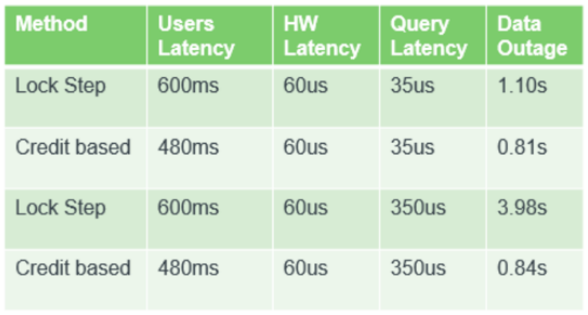
We also measure dataplane latency, i.e., time for packets to start flowing after 10,000 routes are being installed. We measured the worst-case latency, i.e., when the last route being installed is the one needed for the packets to start being routed. We ran *ping* with an interval of 0.01s between packets and look at the icmp sequence number of the first packet successfully transmitted. Route installation is started at the same time as the *ping* command. We measured dataplane latency to be **1.10s** when installing 10,000 routes (110 icmp packets were lost) for the lockstep implementation.

With the credit-based solution, dataplane latency goes down to **0.81s (**80 icmp packets lost) when installing the same 10,000 routes. Packets thus start flowing **~1.3x** faster when using the credit-based solution.

The results are summarized in Table 1:

- **Users Latency** represents the consistent control plane latency;
- **HW Latency** is the constant latency to access hardware when installing a route, hidden in the credit-based implementation.
- **Query Latency** measures the actual time to query hardware, increased by 10x in the experiments (up to 350us) to show that the credit-based solution would not suffer if querying hardware about resources was costly.
- **Data Outage** shows the dataplane latency;

The setup for the experiments (including BGP configuration) can be found on github [8].



| Method | Users Latency | HW Latency | Query Latency | Data Outage |
|---|---|---|---|---|
| Lock Step | 600ms | 60us | 35us | 1.10s |
| Credit based | 480ms | 60us | 35us | 0.81s |
| Lock Step | 600ms | 60us | 350us | 3.98s |
| Credit based | 480ms | 60us | 350us | 0.84s |

Table 1: Measurements for 10,000 routes being updated from FRR to hardware

## Predictive Solution

The predictive solution computes the current occupancy of hardware resources (overestimates if necessary) and predicts with a high degree of confidence that an offload will succeed.

### Flat Model: Simple, Hard Limits for Resources

This solution is equivalent to driver resource profiles: simplistic assumptions on capacity and usage for each object type are chosen which allow for fast in-line checks of a configuration change with a high degree of confidence that hardware offload will succeed. This model always works but never pushes hardware to full utilization because of the simplifying and conservative assumptions.

### Complex Model of Resources and Usage

With enough details about the hardware, a more complex resource usage predictor can be built: kernel resources are mapped to their corresponding hardware resources. The dependencies between hardware tables and shared memories is known by the driver which can then compute the current usage and remaining entries given the current kernel state. Current kernel accounting/models of resources could be modified to better align with typical hardware resources and ease the hardware resource accounting.

## Related Work

The DPIPE interface [5] provides visibility for the user into the hardware pipeline and [4] extends this interface to provide information about available hardware memories, including shared memories and memories' current usage. [1] presented the same issues involved in hardware overload and discussed the need for synchronous feedback in case of hardware failure.

## Future Work

Feedback from the conference will help confirm and direct the work on a resource manager. The credit-based solution will be extended and studied in more details to model rate mismatches from user down to hardware as well as resource representation mismatch. We expect higher speedups with the credit-based solution than found in this paper (using a workqueue item for the refill thread is not the most efficient).

In addition, kernel data structures and representations of offloaded data can be improved to better align with hardware, for example to simplify accounting.

## Conclusion

Hardware acceleration of network functions offloads kernel resources on hardware devices. These kernel resources occupy hardware resources and the current expectations in case of insufficient hardware resources vary from cases to cases. For L3 routing, the expectation is fall back to the software implementation but reverting to software can be harmful if line rate processing of packets is required for a given network infrastructure. Other network functions such as ACLs, address lists on NICs and L2 bridging develop their own custom solutions for resource overload which similarly might not be efficient enough and do not always provide synchronous feedback to the user.

This paper motivated the need for synchronous feedback to the user in case of offload failure in order to avoid incorrect or harmful function behaviors. Different models were presented to deal with the representation mismatch between hardware and kernel resources. A credit-based solution with hardware resources prefetching helps managing complex resources involved in, e.g., routing, where routing resources end up in shared hardware resources and have to be traded against other resources. A predictive model is harder to engineer if resources are shared and vary greatly between devices but can be used in simpler instances.

## Acronyms

| | |
|---|---|
| ACL | Access Control List |
| ASIC | Application-Specific Integrated Circuit |
| FDB | Forwarding Database |
| FIB | Forwarding Information Base |
| IP | Internet Protocol |
| LPM | Longest-Prefix Match |
| MAC | Media Access Control |
| MDB | Multicast Group Database |
| RIF | Router Interface |
| TCAM | Ternary Content-Addressable Memory |
| UFT | Unified Forwarding Table |
| VRF | Virtual Routing and Forwarding |

## References

[1] Jiri Pirko, Ido Schimmel, Matty Kadosh, 2016, Switchdev BoF, Netdev 1.2
https://netdevconf.org/1.2/slides/oct5/08_switchdev-BOF.pdf

[2] Cisco Nexus 9300-EX Platform Switches Architecture
https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-739134.pdf

[3] Arista 7050X Switch Architecture
https://people.ucsc.edu/~warner/Bufs/Arista_7050X_Switch_Architecture.pdf

[4 ]Arkadi Sharshevsky, Driver profiles RFC
https://www.mail-archive.com/netdev@vger.kernel.org/msg181492.html

[5] Arkadi Sharshevsky, ASIC Pipeline Debug API
https://netdevconf.org/2.1/papers/dpipe_netdev_2_1.odt

[6] Roopa Prabhu, Wilson Kok, Hardware accelerating Linux network functions,
https://people.netfilter.org/pablo/netdev0.1/slides/Hardware-accelerating-Linux-network-functions.pdf

[7] Changes to spectrum driver to offload FIB entries inline with user request
https://github.com/CumulusNetworks/net-next/tree/res-mgmt/sync-ipv4-fib

[8] Changes to spectrum driver to simulate the credit-based solution for inline synchronous resource checks.
https://github.com/CumulusNetworks/net-next/tree/res-mgmt/credit-based-fib

[9] Changes to spectrum driver to simulate offload failure resulting in blackholed packets
https://github.com/CumulusNetworks/net-next/tree/res-mgmt/fib-offload-blackhole