eBPF Hardware Offload to SmartNICs: cls_bpf and XDP

Abstract

This paper will lay out a method used to offload eBPF/XDP programs to SmartNICs which allows the general acceleration of any eBPF programs. We will concentrate on kernel infrastructure which has been developed and the in-kernel JIT/translator while covering the HW target architecture to the necessary degree.

Keywords

eBPF, XDP, offload, fully programmable hardware

Introduction

There have been previous attempts to promote general networking offloads within the Linux kernel. However in the past only offloads with a very limited scope have been successful. This is due to a combination of reasons:

Limited Hardware Capability: Currently most common network interface cards (NICs) trade off flexibility for performance and provide simple, mostly stateless, specific offloads.

CPU architectures have scaled: x86 and other general purpose CPUs have scaled well with networking requirements, supporting general stateful networking well.

Vendor Specific Solutions: Most accerlerated solutions have been vendor specific. To ensure general offload, infrastructure should be applicable to different hardware platforms.

It is only now becoming possible to combine stateful processing and performance; there are a number of NPU based SmartNICs with a pool of RISC workers. It is also becoming necessary due to the scaling of networking workloads beyond the capacity of CPUs. This combination of factors suggests that it is the right time to start adding a general offload framework to the kernel.

The are a number of reasons to first focus on eBPF as part of this offload:

- It is a well defined language and machine, with well constrained parameters, registers, memory use, instruction set and helpers.
- Implementation already takes into account running on parallel cores which is good for transparent offload to network specific HW.
- It is the programming method used in XDP and one of those used within TC, which are the most likely targets in the kernel for general offload at this time.

The hardware we are currently focusing on is the Corigine series of NFP based NICs, however it is hoped that this will be applied to other fully programmable NPU based NICs.

Background

This section is designed to give a brief background into the eBPF infrastructure and the NFP's architecture. It is not intended to be exhaustive. For those who require a more in depth reference for eBPF please either see the kernel documentation [3] or Daniel Borkmann's excellent paper [2]. For those who want more information on the hardware please see documentation on open-nfp.org [6].

The eBPF Infrastructure

The eBPF machine consists of **a)** eleven 64bit registers (including the stack pointer), see listing 1 for register descriptions **b)** a 512 byte stack for handling register spilling or general storage **c)** key-value maps without a strict size limit which can be read and written to from the kernel and user space. **d)** 4k of eBPF bytecode instructions, furthermore to this verifier enforces 64k 'complexity' limit - the amount of instructions traversed during verification.

Listing 1: description of the eBPF register set

```
R0 return value from in-kernel function, and exit value for
eBPF program
R1-R5 arguments from eBPF program to in-kernel function
R6-R9 callee saved registers that in-kernel function will
preserve
R10 read-only frame pointer to access stack
```

This well constrained machine is ideal for offloading to lightweight NPU general purpose cores. Another advantage of eBPF is its wide adoption, the simple to use and well defined surrounding infrastructure.

The use of the LLVM compiler ensures that it is relatively trivial to build eBPF programs in C, note this also exposes an interesting surface for future optimization of bytecode for specific targets (within eBPF spec). Combined with this, the existing verifier, which ensures that programs do not create loops through the use of depth first search (DFS) and carries per instruction state for the eBPF bytecode program, is a great candidate for reuse in hardware compatibility checks and program parsing/analysis. eBPF already presents the notion of multiple CPUs which is implicit within the NFP and TC has already been extended to enable transparent hardware offloads. This combination allows the reuse of significant amounts of kernel infrastructure along with ease of mapping to the NFP.

The NFP Architecture

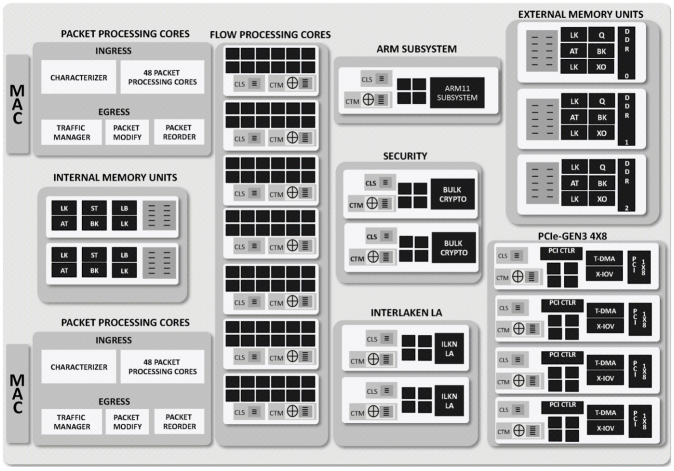

Figure 1: High level architecture of NFP 6xxx chip to indicate the scale and capability. Note ME islands in the centre

The NFP architecture consists of a series of hardware blocks for certain specialist functions (encryption, reordering, memory operation) combined with a group of fully programmable microengines. These are arranged into islands containing 12 microengines each (between 72 and120 MEs per chip) as well as SRAM cluster target memory (CTM) and cluster local scratch memory (CLS). The NIC also contains 2-8GB of DRAM memory and 8 MB of SRAM.

The Microengines Each ME has the ability to run 4 or 8 cooperatively multiplexed threads with a shallow pipeline, this ensures that cycles are used effectively. Lockless transactional memory architecture allows memory to be less of a bottleneck than it would otherwise be in a many core architecture. Each ME has 256 32bit general purpose registers, divided into A and B banks. As in the figure 2 these are divided up between the threads, leaving each thread with 16 A and 16 B registers when running in 8 context mode (32 each in 4 context). A and B registers can only interact with registers in the other bank, not with other registers in their own bank. There are also 256 32bit transfer registers (128 read, 128 write) and 128 32bit next neighbour registers. Each ME also has 4KB of local SRAM memory.

Memory System There are four different types of memory, three on chip and one off chip. As shown in figure 2 each island has its own SRAM consisting of CLS and CTM, these are 64KB and 256KB in size. CTM is used to store the first portion (configurable, but currently 2k on receive) of packets being processed-enabling low latency processing of packet headers. CLS is used as a local state overflow for MEs. Global chip memory consists of 8MB of on chip IMEM (SRAM) and 2-8GB of off chip EMEM. IMEM mainly contains packet payloads and EMEM is where large data structures tend to be stored, however, some caching may be used in lower latency memory.

Mapping an eBPF machine onto the NFP

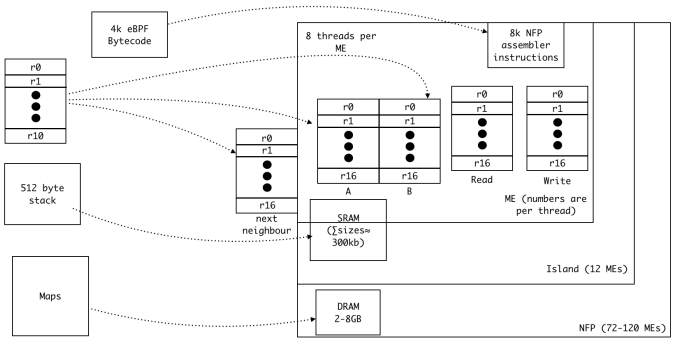

Figure 2: Illustration of the conceptual mapping of the eBPF machine to a ME thread

Figure 2 shows how the eBPF machine can be mapped to the NFP. Maps up to a certain size can be placed in DRAM, with some potential caching features improving access speed and multi-thread time multiplexing being used to hide any latency.

The stack can be placed in a combination of per ME SRAM and per island SRAM, depending on size. The NFP instruction store can hold up to 8000 NFP assembler instructions, however multiple MEs can be tied together to increase the size of the potential instruction store.

Finally through a combination of general purpose A,B registers with next neighbour combined with transfer registers the 10 64bit general purpose eBPF registers can be mapped. If the NFP is running in 4 context mode, this is trivial due to the large amount of registers allocated to each thread. However in 8 context some thought is required to optimise register use, for example data can be stored directly in the transfer registers if the only purpose is to transfer it later.

The Programming Model

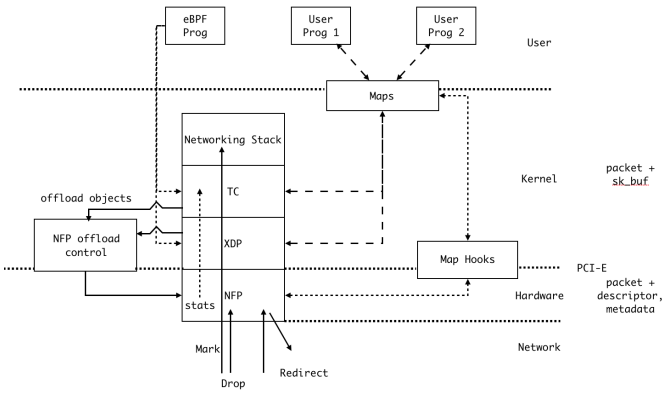

Figure 3: The high level architecture of the eBPF offload model as used on either TC or XDP hooks

Figure 3 is designed to show how the transparent offload of eBPF programs fits into the networking stack. eBPF programs are attached to the XDP or TC hooks in the standard manner. TC and XDP infrastructure form offload objects which are handed over to the in-driver verifier and translator, nfp_bpf_jit.c (see figure 4) which is used to convert the eBPF bytecode to the programmable hardware's machine code. Maps may also be offloaded as the hardware contains 2-8GB DRAM. The firmware is then able to return processed frames along with metadata and statistics to the host.

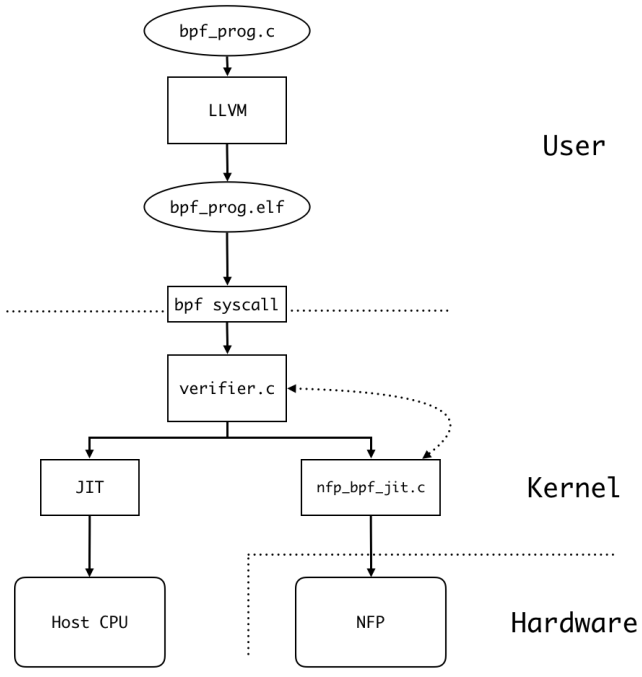

Figure 4: Programming model showing HW-specific JIT

Note that the offload mechanism described above is transparent, the user is not required to make any changes to their applications. There is devicewide offload on/off control via ethtool, this is similar to the mechanism used for standard offloads such as LSO, checksum offload and vlan acceleration. Ethtool configuration is combined with per program flags for fine grained control allowing forcing offload or non-offload. For more detail on this see the Kernel Infrastructure section.

Kernel Infrastructure

In this section we will describe changes made to different subsystems of the kernel as part of our implementation. Although the TC and XDP hooks are described separately the kernel infrastructure required for this type of offload is quite similar in both cases (and the driver/translator handles both of them). The area in which differences between hooks have the biggest impact on implementation is how fine grained communication is with the relevant section of the stack, XDP being fine grained enough to easily implement a fallback path for partial offload.

Traffic Control (TC)

The TC subsystem has already been extended for transparent hardware offloads [1, 4]. Standard set of flags has been introduced for u32 and is used by other classifiers such as flower (e.g. TCA_CLS_FLAGS_SKIP_HW). The team responsible for flower offload added an ability for drivers to populate statistics associated with TC actions via the tcf_action_stats_update method [5].

cls_bpf has two distinct modes of operation that need to be supported, the traditional mode of calling the TC actions, and the direct action mode as described by Daniel Borkmann at Netdev 1.1 [2].

Direct action mode involves simply offloading the TC program as the actions are expressed through program's return codes, whereas when TC actions are called, it is also required to offload the actions that are being taken (if they are supported) and later updating their statistics.

If a cls_bpf program is added to TC with the TCA_CLS_FLAGS_SKIP_SW flag, then the program will only be run in hardware, see figure 5. Likewise if the TCA_CLS_FLAGS_SKIP_HW is used, the program will only be run inside the kernel. If a program has the TCA_CLS_FLAGS_SKIP_SW flag set but it is determined by the hardware specific JIT not to be supported by the target, an error will be returned to the user space control application.

In the case of no flag being set, the NFP offload control will try to run the program in the hardware as well as it being run in the software, as shown in figure 5. Note that if the program modifies the packet, there is the possibility of a packet being modified twice.

There is also the possibility of state inconsistency due to programs being run in different orders than intended. This however applies across flower, U32 etc and is not specific to eBPF and is future work for all the offloads within TC.

XDP

Due to XDP's the tighter integration with the driver, it is easier to make use of the *bpf applied flag* within the packet de-

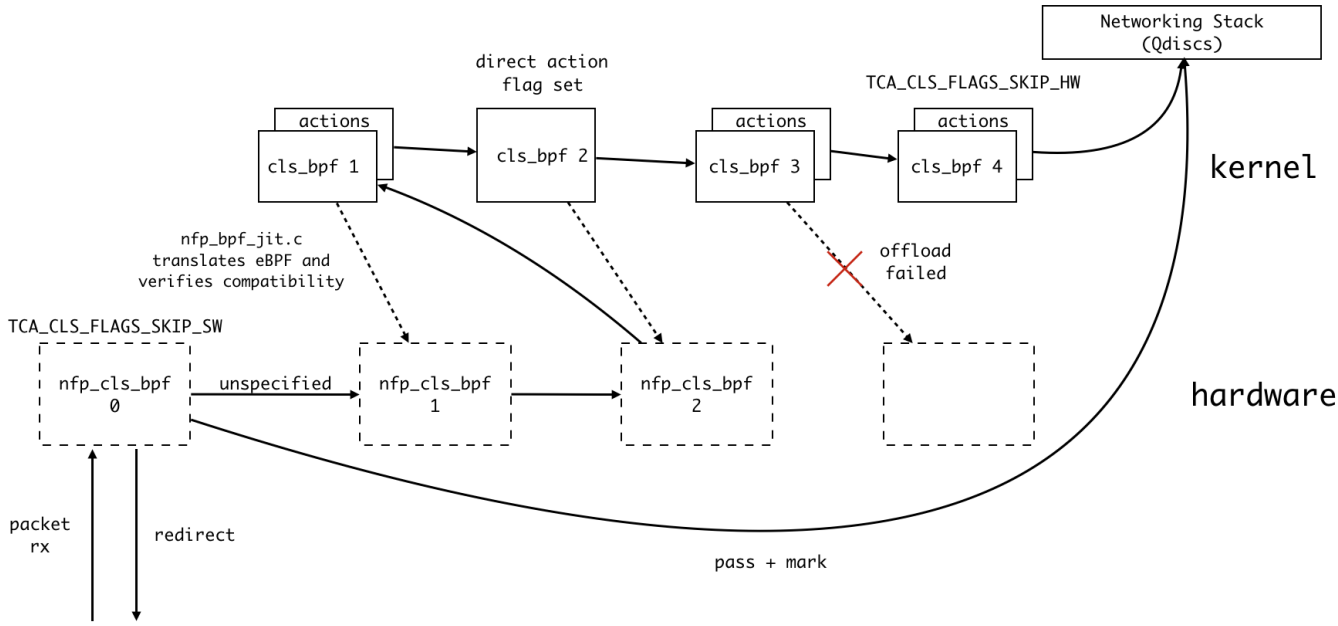

Figure 5: The flow of a packet to indicate what a series of cls_bpf programs would look like when offloaded. Note that due to space constraints only the full return codes to the first program are shown, otherwise unspecified path is only illustrated

scriptor. This allows a per packet decision about rerunning the program in the driver. Which in turn simplifies the implementation of an effective fallback path in the driver. The ability to create effective fallback paths allows 'optimistic offload', i.e the offload of programs which we may not be able to execute within the NIC (e.g due to unsupported instructions on some of the paths) because we can be assured of the ability to enact fallback path, see figure 6.This can be used as the basis for the partial offload of programs, if a program could be split into parts, a subset could be offloaded, with the rest contained in a fallback path.

A potential improvement to the XDP API that could be useful would be a set of flags to control offload in a similar way to TCA_CLS_FLAGS_SKIP_SW in cls_bpf. Finally due to the simpler set of return codes, XDP lends itself better to hardware offload.

Verifier

There are certain operations that the verifier may accept as they are compatible with the host architecture, which may not be supported by the offload target. Examples of this include certain return codes relating to actions (as shown on figure 3 we only support pass, drop and redirect in TC)[1].

The mixing of pointers is a difficult case, for example, it may be appropriate in the host to use a single instruction to dereference a packet pointer in one flow and a stack pointer in another flow. However the offload target may use different memories and require different instructions for access.

[1]Daniel Borkmann has suggested during discussions a way of reporting return codes from the driver to TC, which may solve this problem using the tc_verd field of sk_buff.

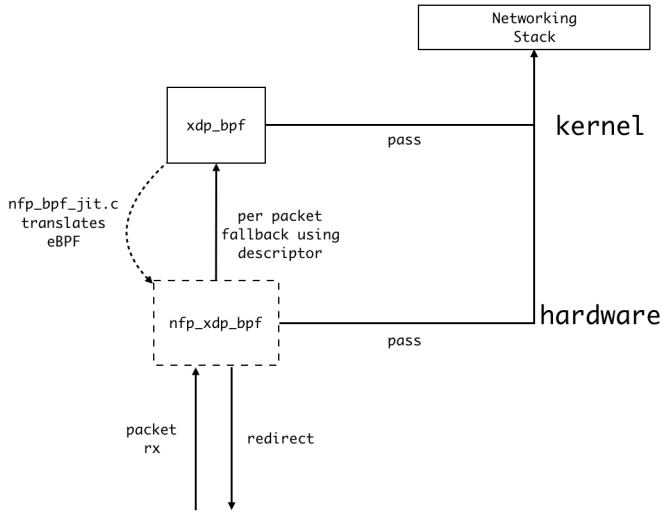

Figure 6: Flow of packet through XDP eBPF program, note the presence of a per-packet falback path due to per-packet descriptor field

Given the requirement to do more verification per instruction, a simple user defined callback was inserted into the kernel eBPF verifier and the verifier state was exposed in a header file. Drivers are now enabled to rerun the validation while performing their own checks. Note that the final full

set of verifier callbacks could be expanded to include two per instruction calls (pre and post verification) and a state compatibility check during path pruning.

Maps

We identified three types of map offload cases based on type of accesses the offloaded program is performing. Most basic case is where the map is read only and populated by other programs or from the user space. Second case is where map is populated by other programs or from the user space but the offloaded program performs atomic add operations on existing elements (most likely gathering statistics). Third case is where program has read and write access to the map.

From the verification of update calls it is possible to determine whether a particular program interacts with the map in a read only, write only or read/write manner. Simple hooks on read and write paths in map infrastructure should allow us to reflect written values to maps on the device and provide read results from them. In read only and write only access cases there should be another copy of the map in kernel space. Write only maps may require gathering of results from multiple sources and combining them.

Read/write maps are the most complex case since most offload targets are not close enough to the CPU memories to make it possible to achieve coherency between maps in main memory/CPU caches and on the device at reasonable update speeds speeds. Therefore we anticipate that the read/write maps will have to be explicitly "claimed" by the offloaded program and subsequent attempts by other non-offloaded programs to bind to such map must fail. Note that this includes a case where an offloaded program is being replaced by a non-offloaded one since the existence of the two will necessarily overlap. This is because replace is not atomic across CPUs, even if it was, however, we would still most likely not be able to evict the map from hardware to CPU memory fast enough to provide expected replace speed.

Given the limitation and potential unexpected user-visible behaviour we should only attempt offload of read/write maps for programs which are explicitly marked as hardware only (using relevant TC/XDP flags). Hardware only maps will be marked as such and any other program trying to attach to them later will fail.

NFP Infrastructure

Bytecode to NFP Assembler Conversion and Register Allocation

Translation of eBPF instructions to the hardware machine code is done after the verifier collects relevant instruction state. Due to the worker CPU cores on the offload target being 32 bit as opposed to the 64 bit registers mandated by eBPF there is an optimisation opportunity in identifying 32 bit instructions. This shall significantly improve performance due to more efficient translation of instructions (less code) and less register contention. As can be seen in the unoptimised example in listings 2 and 3, there is a danger that 32 bit explosion counts can increase significantly when translating 64 bit instructions.

Listing 2: Simple eBPF Program

```
BPF_ALU64 | BPF_MOV | BPF_X,6,1,0,0
BPF_LD | BPF_ABS | BPF_W,0,0,0,0x0e
BPF_ALU | BPF_MOV | BPF_K,1,0,0,0xffff
BPF_ALU64 | BPF_AND | BPF_X,0,1,0,0
BPF_EXIT | BPF_K | BPF_JMP,0,0,0,0
```

Listing 3: Corresponding NFP Assembler

```
#Check pkt length
alu[--,n$reg_3,-,0x12]
bcc[.1011]
#Read packet from memory
mem[read8,$xfer_0,gprA_15,0xe,4],ctx_swap[sig1]
#Load read values into GPRs and zero extend
alu[gprA_0,--,B,$xfer_0],gpr_wrboth
immed[gprA_1,0x0],gpr_wrboth
#Load constant (and zero extend)
immed[gprA_2,0xffff],gpr_wrboth
immed[gprA_3,0x0],gpr_wrboth
#Perform AND
alu[gprA_0,gprA_0,AND,gprB_2],gpr_wrboth
immed[gprA_1,0x0],gpr_wrboth
#Exit
br[.1014]
```

Through the use of standard compiler techiniques of dataflow analysis it should be possible to track which operations need to be performed on full 64 bit values and which require only 32 bit results. Attempts to extend the kernel verifier/analyzer to help with performing this task have already started. First result of this work will be the introduction of full liveness analysis in the verifier which is hoped to be published soon.

The alternative approach is to move the 32 bit optimization to the LLVM compiler by introducing a 32 bit eBPF machine subtype. Exploiting the user space compiler tools has the obvious advantages; however, it takes away some of the transparency and it remains to be seen whether most programs can even be complied to such a subarchitecture (due to eBPF 32 bit instruction set not being able to express 64 bit instructions).

There are optimization possibilities stemming from the fact that the target instruction set allows some instructions to be merged while performing instruction translation. The NFP can do bit operations along with shift/mask operations in a single instruction, as shown in listings 4 and 5. Also the NFP assembler specifies destination register independently from source operands (similarly to ARM but unlike x86) therefore helping to reduce the amount of register to register moves.

Listing 4: eBPF Shift Mask Snippet

```
BPF_ALU | BPF_SHR | BPF_K,0,0,0,5
BPF_ALU | BPF_AND | BPF_K,0,0,0,0xff
BPF_ALU | BPF_MOV | BPF_X,1,0,0,0
```

Listing 5: Corresponding Line of NFP Assembler

```
alu_shf[gpr1, 0xff, AND, gpr0, >>5]
```

Another area of possible future work is improving register allocation to A,B banks (which is an NP-hard problem in

itself). Currently the most optimal brute force method for register allocation is to run in 4 context mode, allowing the NFP to duplicate the eBPF registers in both the A and B banks. This is acceptable due to the current implementation requiring minimal latency hiding. Over time this may become a more significant issue and we have an number of proposed optimizations to the firmware that would address this.

Stack

Placement of the stack may be significantly different within the NFP depending on the size of required the stack and the amount of threads used (4 or 8). Also there may be optimizations around using the stack slot type to determine placement, e.g prioritising STACK_SPILL over other bpf_stack_slot_type's. Implementing the stack is one of the next steps in driver and firmware implementation.

Maps

The implementation of maps will partially rely on hardware and firmware infrastructure which already exists for the target. This implies that the hashing and lookup algorithms may differ from those used in the kernel, it is hoped to be an acceptable approach. The offloaded maps will be accessible via firmware control messages.

The map implementation will be reused for some internal needs like mapping *ifindexes* for redirect actions to hardware port IDs[2].

Conclusion

cls_bpf and XDP are fast and efficient classifiers, however as time goes on, efficient use of CPU will become more important as we move through 10 to 25 or even 100 Gbps. To ensure that networking is able to cope without an explosion in CPU usage requires the implementation of an efficient and transparent general offload infrastructure in the kernel. We believe this is the first step in that direction.

This paper is designed to outline our initial proposals and prompt feedback; it is by no means a *fait accompli*. As time goes on we believe this coprocessor model may become important to drive more general datapath accelerations as there are similarities between some of the problems that need to be solved and various datapath offloads such as OVS and Connection Tracking. This makes it important that this work is driven in a direction which can be generally applied by other offloads as well as other hardware. This paper has outlined some of the challenges (instruction translation, map sharing, code verification) and implemented some solutions to the required problems, while ensuring that the infrastructure is reusable. However there is plenty of work to do before the final goal is in sight.

References

[1] Almesberger, W., Linux Network Traffic Control-Implementation Overview, https://www.almesberger.net/cv/papers/tcio8.pdf.

[2] Borkmann, D., On Getting the TC Classifier Fully Programmable with cls_bpf, *NetDev 1.1*.

[3] Starovoitov, A et al., Linux Socket Filtering aka Berkeley Packet Filter (BPF) *Linux Kernel Documentation*.

[4] Salim, J.H., Linux Traffic Control Classifier-Action Subsystem Architecture *Netdev 0.1*.

[5] Salim, J.H, Bates, L ., The CLASHoFIRES: Who's Got Your Back? *Netdev 1.1*.

[6] OpenNFP.org

[2]Mapping *ifindexes* to port IDs is required for the cls_bpf direct action mode where redirection is setup by a function, parameters of which may be computed at runtime or taken from a map.