XDP Hardware Offload: Current Work, Debugging and Edge Cases

Jakub Kicinski, Nicolaas Viljoen
Netronome Systems
Santa Clara, United States
jakub.kicinski@netronome.com, nick.viljoen@netronome.com

Abstract

This paper is designed to provide an update of the work ongoing with respect to XDP offload to hardware with a focus on the Netronome Network Flow Processor. This includes proposed kernel changes, updates to the driver and also the addition of bpftool for the introspection of programs and maps. Some proposed improvements to the offload infrastructure will be generally applicable, including bpftool, which provides additional functionality to non-offloaded programs.

Keywords

eBPF, XDP, offload, fully programmable hardware

Introduction

As discussed at netdev 1.2, the Netronome Flow Processor (NFP) architecture is well suited for the offload of eBPF. Since that time there has been extensive work done with regards to both the implementation in FW as well as mechanisms for offload within the kernel. This paper will provide an update of:

Infrastructure Enhancements: Rearchitecting the 2nd run of the verifier, map offload and improving the verifier to utilise 32-bit ALU instructions

Tools and Debugging: bpftool and its use for the introspection and testing of programs and maps

Driver Additions: Stack implementation, Minor updates

Future Work: Tails calls and partial offload

Overview of Previous Work

In the previous paper at netdev 1.2 the initial offload model was explained with the objective of transparency and simplicity stated. The offload was designed so that existing programs would work with a minimum of change compared with the implementation for the host. This was done with the model as shown in fig 1, which involves adding an nfp jit within the driver. This allows the user to compile the same program, whether running on the NFP or on the host, the location where program is loaded can be decided by the use of the xdpoffload or xdpdrv option in iproute2.

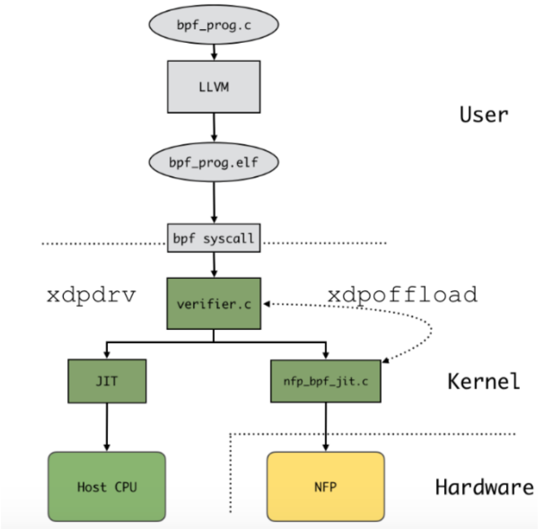


Figure 1: High level model for transparent XDP hardware offload

The current mechanism for offload is described in fig 2, whereby a network device driver passes the program through the verifier a second time to ensure the program is valid for offload, this is enabled by the callback in the verifier. If this check is passed, the instructions are optimised and jited to NFP assembler and loaded onto the NIC where the mapping of the elements of the BPF machine are fairly simple, fig 3. However, a key issue with this mechanism is that it is difficult for the driver to provide useful debug information in this configuration. The section below describes the proposed architecture that will allow the second pass of the verifier to provide this information.

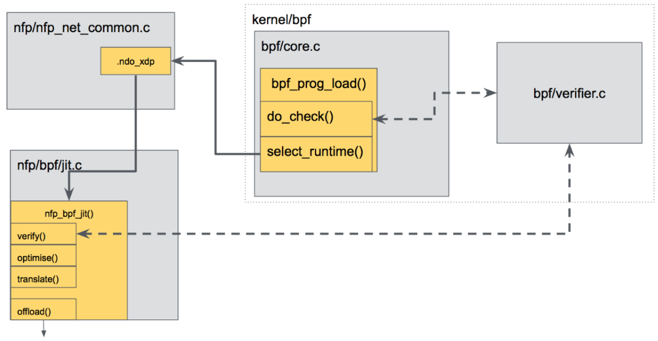


Figure 2: Current mechanism for the JITing of the BPF bytecode when offloaded, note the verifier callback is in the driver

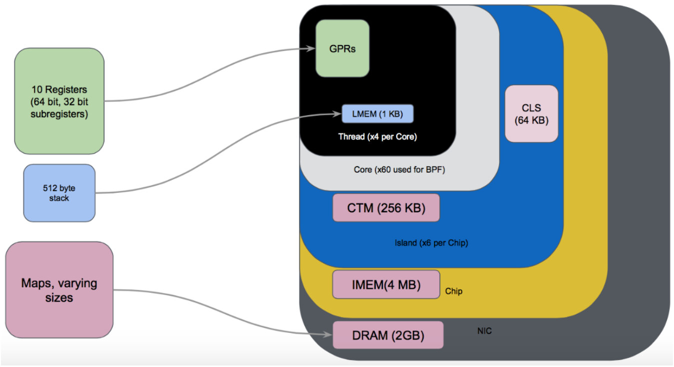


Figure 3: Mapping of the BPF machine to the NFP's cores

Current Work

Enhancements of XDP Offload Architecture

There are a number of updates to the general XDP hardware offload architecture currently underway.

Verifier 2nd Pass: Moving the JIT logic closer to the load time through a device specific hook, this would ensure that translation mechanisms and debugging are both significantly simplified

Map Offload: How to manage the RCU lock in the offload case, the proposal would be to use flags to optimise this

LLVM and 32 Bit: Ensuring that 32-bit architectures are able to utilise recent changes in LLVM through verifier compatibility with 32-bit ALU instructions

Re-Architecting the Second Verifier Pass Currently the mechanism for offload is as described in the introduction, whereby the second pass of the verifier occurs within the driver. The current proposal would be to move the JIT logic (verify, optimise and JIT) to close to the load point through a device-specific hook. This provides significant advantages due to the fact that it will improve debugging and will also allow significant simplification of the translation mechanism since the original, unmodified eBPF bytecode is still available, context accesses, function calls and map pointer loads are still not translated. There is also code generation which happens after the first verifier pass like function 'inlining' or prolog generation. Allowing the device-specific JIT to operate in the first pass simplifies the work, and make it easier for users to understand why offload fails ('using feature X causes some code generation inside the kernel' may not be entirely intuitive to newcomers). The device JIT would also be able to integrate at this point with the existing verifier log infrastructure. Accessing the device machine code could then be accomplished through the existing interface for retrieving JITed image from the kernel (provided program info is extended with the 'JITed for device X' attribute).

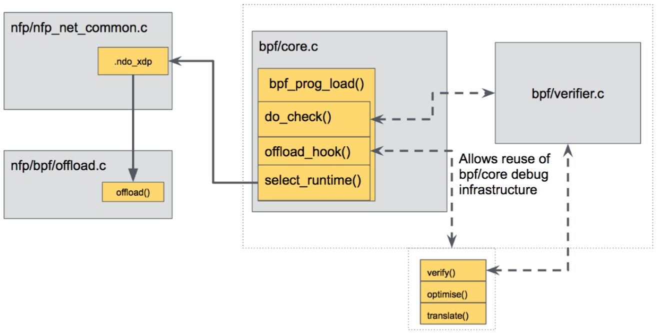


Figure 4: Proposed mechanism for the JITing of the BPF bytecode when offloaded, note the verifier callback done via a device-specific hook

Map Offload The map offload requires diverting the map operations into the driver. One of the challenges is that the map operations are currently invoked under the RCU lock. The current way that map calls are made is that they take the RCU lock before checking the specific callback is invoked, then once they are done they release the lock. We currently work around that by busy-waiting for the FW to reply, but it would be preferable to make the RCU lock optional before upstreaming our PoC code.

32 Bit Optimisation-LLVM and the Verifier Through recent patches added to LLVM, there is now the ability to generate 32-bit ALU instructions, as opposed to the ALU64 instructions that were previously generated by LLVM in all cases. This is possible through the use of the subregisters that are defined within the BPF architecture and ensure the semantics of the original 32-bit ALU operations are preserved. These do not break compatibility with 64-bit architectures as it is well defined that these sub-registers zero extend to 64 bit values when being written to.

Note however that this does not mean that the entire program is being converted to 32-bit, if pointers are still 64-bits wide as there would otherwise be a requirement to define a register-pair ABI to be handled by the JIT. This would put the complexity onus on the host JIT. The NFP JIT handles this by using predefined static register pairs which are handled in a well defined manner to reduce complexity of any combination, movement or splitting operations.

To make this work will require some small modifications in the verifier-such as the use of BPF_W as well as BPF_DW checks in a few locations. It may also become useful for there to be more formal data-flow analysis in the verifier as well as a formal control flow graph, as this will allow further improvements in code generation. An example of this would be tracking the upper half of the 64-bit registers when tracking their liveness to ensure they are correctly nulled.

Tooling and Debugging

A key aspect of ensuring that BPF offload is production ready is the tooling and debugging. This however was also a larger problem within BPF. In particular the introspection of maps and programs. This led to the development of the bpftool, which allows the user to show which maps are present within the system as well as inspect them, delete them and update them. It is also possible to dump the programs which are present.

Listing 1: Currently available commands within bpftool

```
        bpftool map show   [MAP]
        bpftool map dump    MAP
        bpftool map update  MAP
key BYTES value VALUE [UPDATE_FLAGS]
        bpftool map lookup  MAP  key BYTES
        bpftool map getnext MAP [key BYTES]
        bpftool map delete  MAP  key BYTES
        bpftool map pin     MAP  FILE
        bpftool map help

        MAP := { id MAP_ID | mapid MAP_ID | pinned FILE }
        PROGRAM := { id PROG_ID | progid PROG_ID | pinned FILE | tag PROG_TAG }
        VALUE := { BYTES | MAP | PROGRAM }
        UPDATE_FLAGS := { any | exist | noexist }

        bpftool program show [PROGRAM]
        bpftool program dump xlated PROGRAM  file FILE
        bpftool program dump jited
PROGRAM [file FILE] [opcodes]
        bpftool program pin   PROGRAM FILE
        bpftool program help

        PROGRAM := { id PROG_ID | progid PROG_ID | pinned FILE | tag PROG_TAG }
```

This tool allows the user to then have significant visibility into what is being offloaded in the case of the use of the use of a programmable NIC. It is our belief that others will need this infrastructure also to be able to handle debugging when offloading BPF

As alluded to above in the previous section, other Future Plans include the ability to dump outputs in the second pass of the verifier.

Driver Updates

Stack Offload The stack offload is one of the more challenging aspects for the JIT to deal with in terms of translation to the NFP cores, as the processing cores on the NFP do not contain a stack. This means that the local memory on the NFP core is used to create a stack. There are a couple of difficulties presented by this:

1. LMEM is accessed indirectly, a LMEM pointer has to be moved around to be able to address the correct place in the stack if the stack is larger than 64 bytes
2. The LMEM is word based, this means that there needs to be state tracking to ensure the translator is aware of which byte within the word is currently being pointed at.

We split the stack accesses into 5 cases:

a) Stack access within first 32/64 bytes of the stack to a constant address - one pointer is always pointing at the base of the stack to allow quicker access, it can be used in shift operations to access up to 32 bytes (unaligned accesses) or in move operations to access up to 64 bytes (aligned accesses)

b) For unaligned access in the 32 - 64 byte window, the base pointer can still be used, but it is required to load the words into a GPR before shifting them

c) Accesses to a known offset beyond 64th byte require loading a LMEM pointer register with the address, once the pointer is loaded (which costs us up to 5 cycles) we are back to cases (a) and (b)

d) Accesses which cross the 64 byte boundary have to be treated specially, because LMEM pointers don?t allow offsetting across those. This requires the loading of the LMEM pointer to the actual address and instead of using the offsetting mode, thereafter an ‘iteration’ mode is used where the exact value under the pointer can be read and then the pointer can be incremented or decremented to access next LMEM word.

e) For stack addresses which are not known at compilation time (different paths through the program may use the instruction to access different words) it is important to make sure that all accesses have the same (mis)alignment to 4 bytes (word size) so that the correct series of shifts can be emitted, thereafter the LMEM pointer with the exact address computed in the BPF code is loaded and the ‘iteration’ mode of the LMEM pointer is used to access whichever word of data the pointer is referring to.

Once this is handled, implementing the stack is trivial, the key part of the translation is reusing the verifier state tracking to understand when registers contain pointers to the stack and to which offset they are referring. This is an example of why the offload requires access to the verifier data via callback.

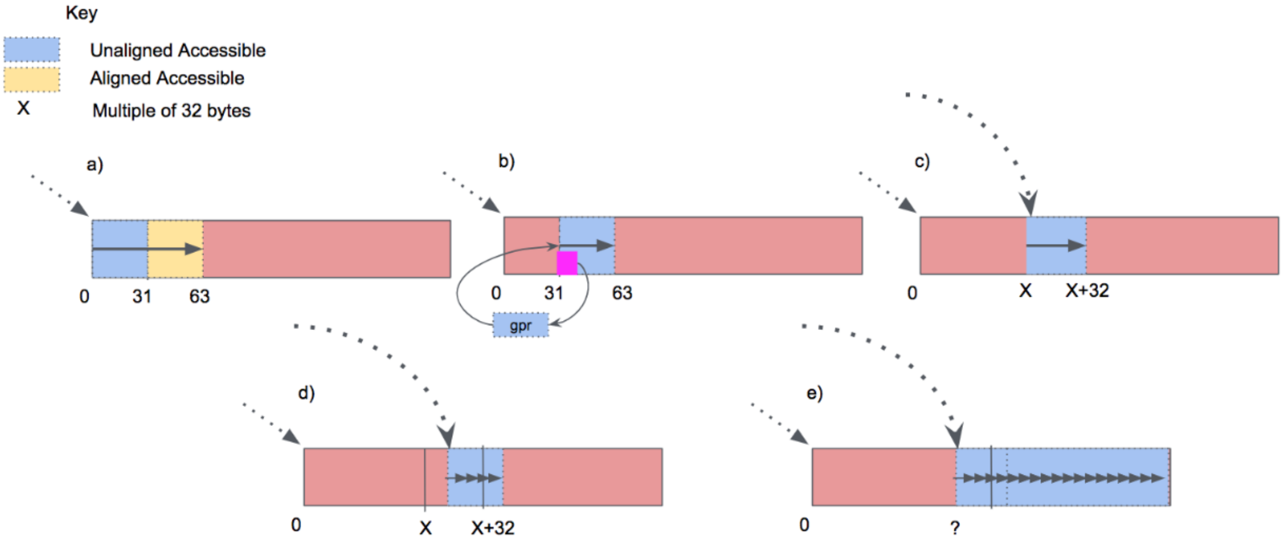


Figure 5: The 5 cases to be handled by LMEM when implementing the BPF stack

Minor Updates Since NetDev 1.2 there have been a number of small updates to the driver which are worth noting

1. New FW: The amount of BPF processing cores has jumped to include up to 60/70 cores depending on the silicon used (previously 40/56)
   (a) To accommodate this new production FW, a new ABI version number has been added to the driver. This will ensure that only the correct FW will attempt BPF offload
2. Direct Packet Access: This has been upstreamed-currently this accesses packets in an area of memory known as Cluster Target memory (CTM)

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

Future Work

Handling Tail Calls

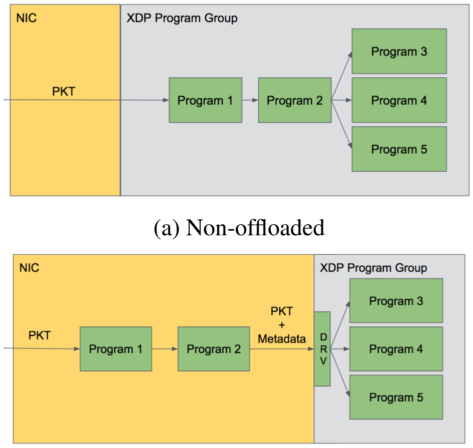
Tails Calls will be handled fairly easily by the system, this will be done by simply adding the programs to the already JITed image. If a series of programs is jitted and the length becomes too long for a single code store to handle, it will be possible to link multiple code stores to increase the amount of instructions available (each code store is 8k insns).

Partial Offloading

There may be scenarios where certain programs are not desired to be offloaded for a number of reasons

- Instructions used not yet fully supported in NFP, e.g Map type that is not yet supported
- Semantically preferred, e.g you would prefer to optimize resources on NFP for other XDP programs
- Resource Requirement e.g Maps may need more than the possible 2, 8 or 24GB available on the NIC

It is important that the offload is able to handle these smoothly, which is why a model of partial offload is important. The suggestions made for reusing the XDP metadata at netdev 2.2 may be very useful for this purpose. Another possibility would be using the driver metadata for this purpose and injecting the packet at the correct XDP program in the list in case of tailcalls.



(a) Non-offloaded

(b) Offloaded

Figure 6: Proposed model for partial offload through the use of metadata to inject packets into correct XDP program

Summary

This paper has shown the current state of XDP offload to HW, as the model is coming close to production implementations, visibility and the ability to inspect the offloaded programs and maps is essential. As well as this, being able to practically do partial offload in situations where that is applicable ensures that XDP offload to HW will be available to more users and ensure the applicability to more situations where XDP is used.