NETRONOME

BPF Hardware Offload Deep Dive

Jakub Kicinski

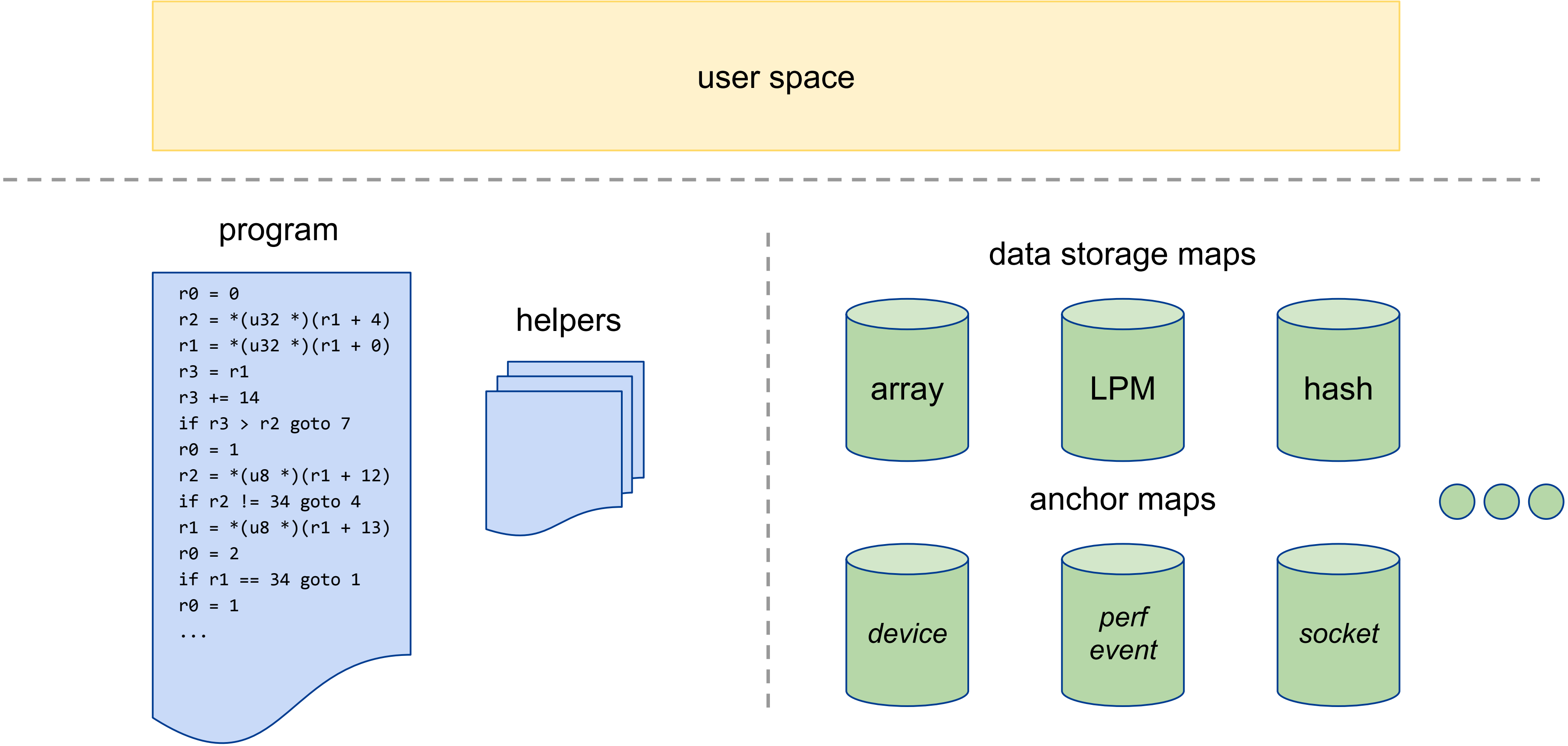
BPF Sandbox

- As a goal of BPF IR JITing of BPF IR to most RISC cores should be very easy
- BPF VM provides a simple and well understood execution environment
- Designed by Linux kernel-minded architects making sure there are no implementation details leaking into the definition of the VM and ABIs
- Unlike higher level languages BPF is a intermediate representation (IR) which provides binary compatibility, it is a mechanism
- BPF is extensible through helpers and maps allowing us to make use of special HW features (when gain justifies the effort)

BPF Ecosystem

- Kernel infrastructure improves, including verifier/analyzer, JIT compilers for all common host architectures and some common embedded architectures like ARM or x86
- Linux kernel community is very active in extending performance and improving BPF feature set, with AF_XDP being a most recent example
- Android APF targets smaller processors in mobile handsets for filtering wake ups from remote processors (most likely network interfaces) to improve battery life

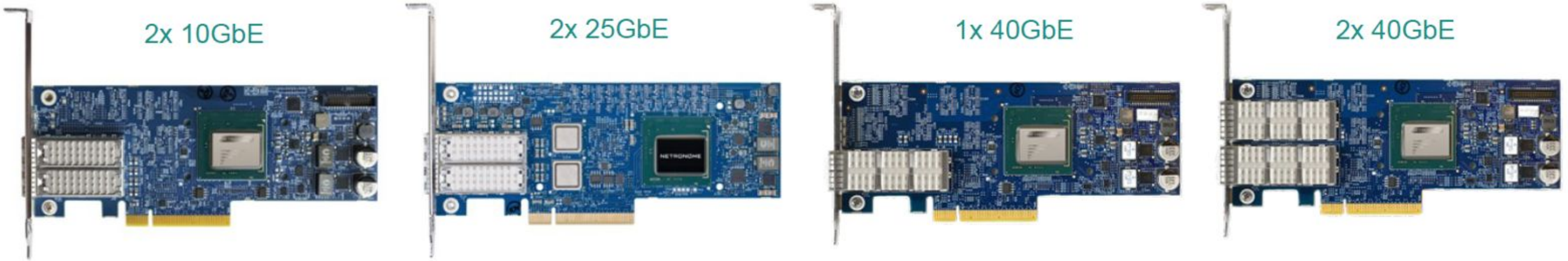
BPF Universe



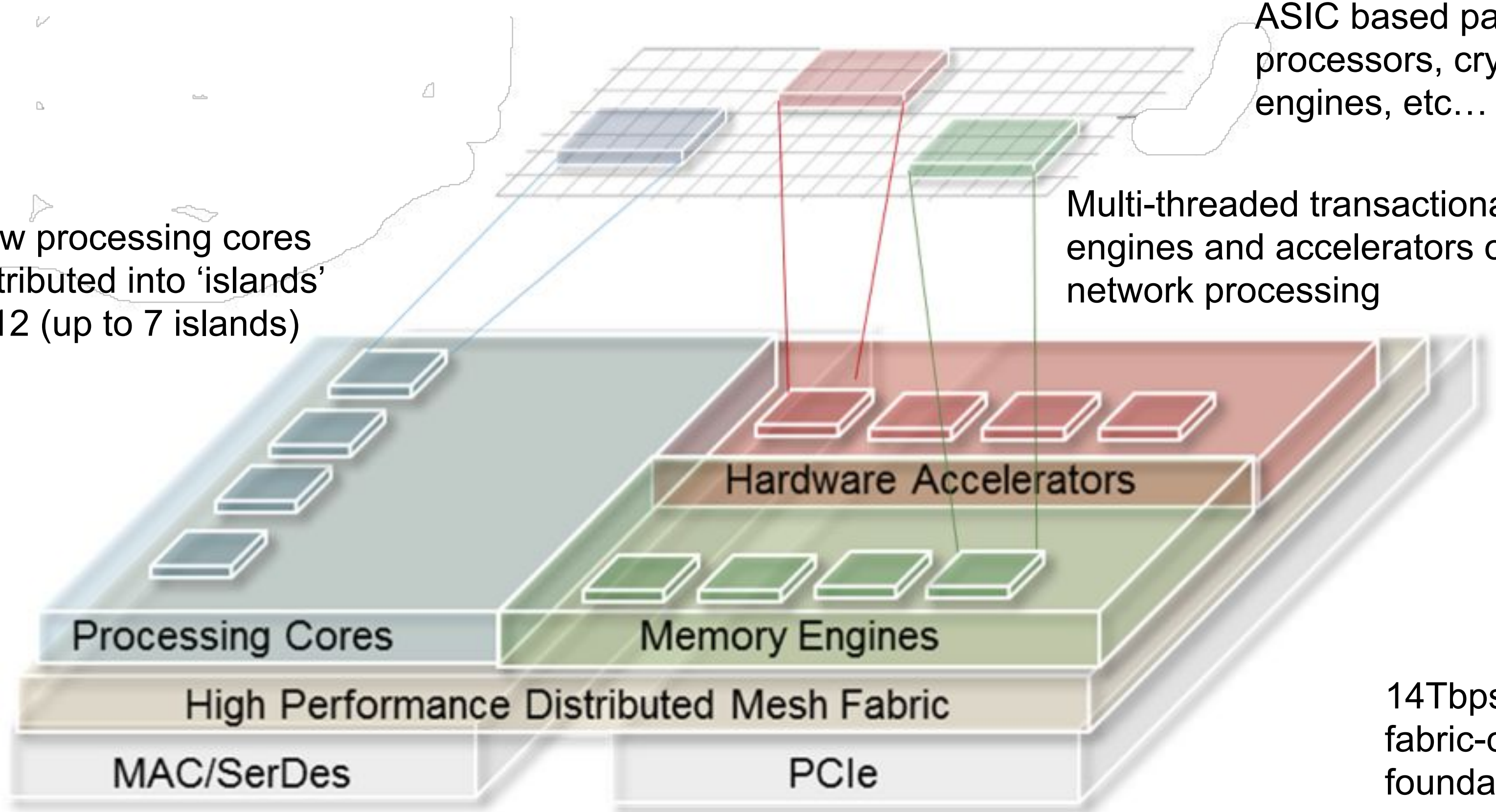
BPF Universe

- Translate the program code into device's native machine code
  - Use advanced instructions
  - Optimize instruction scheduling
  - Optimize I/O
- Provide device-specific implementation of the helpers
- Use hardware accelerators for maps
  - Use of richer memory architectures
  - Algorithmic lookup engines
  - TCAMs
- Filter packets directly in the NIC
- Handle advanced switching/routing
- Application-specific packet reception policies

- Optimized for standard server based cloud data centers
- Based on the Netronome Network Flow Processor 4xxx line
- Low profile, half length PCIe form factor for all versions
- Memory: 2GB DRAM
- <25W Power, typical 15-20W



SoC Architecture-Conceptual Components



ASIC based packet processors, crypto engines, etc…

Flow processing cores distributed into 'islands' of 12 (up to 7 islands)

Multi-threaded transactional memory engines and accelerators optimized for network processing

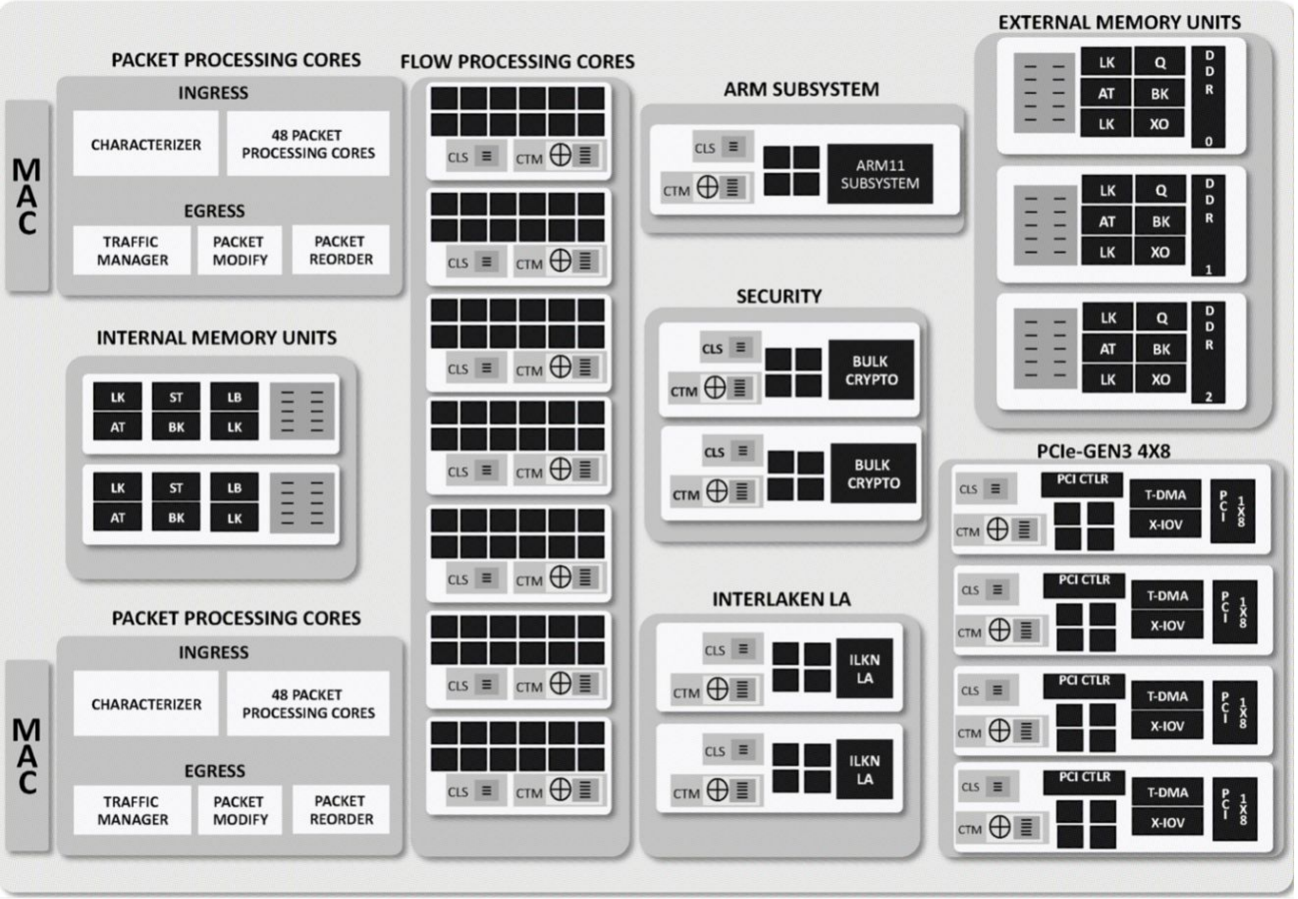
14Tbps distributed fabric-crucial foundation for many core architectures

Up to 100 Gbps

4x PCIe Gen 3x8

NFP SoC Architecture

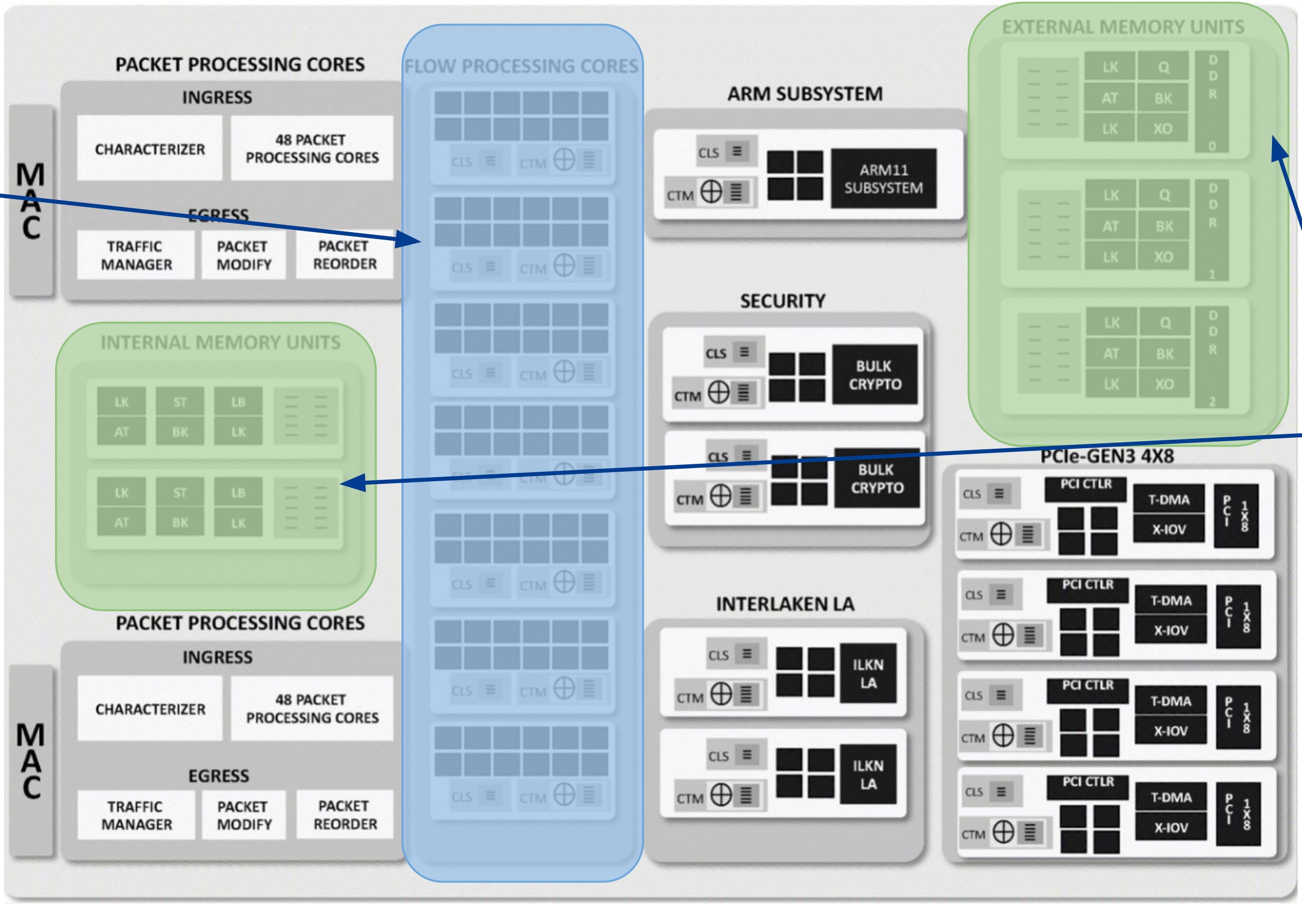
NETRONOME



NFP SoC Architecture

NETRONOME

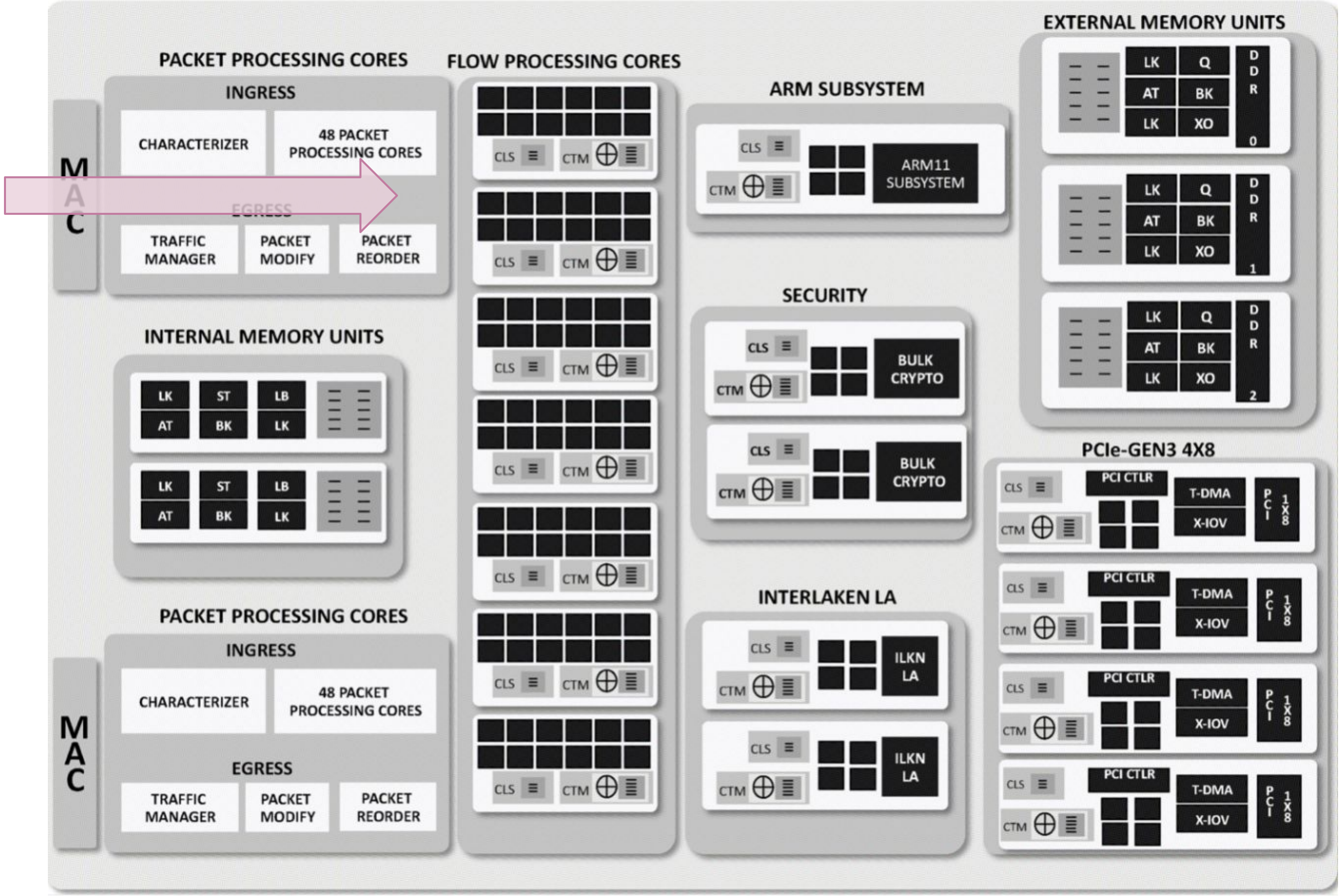
BPF programs



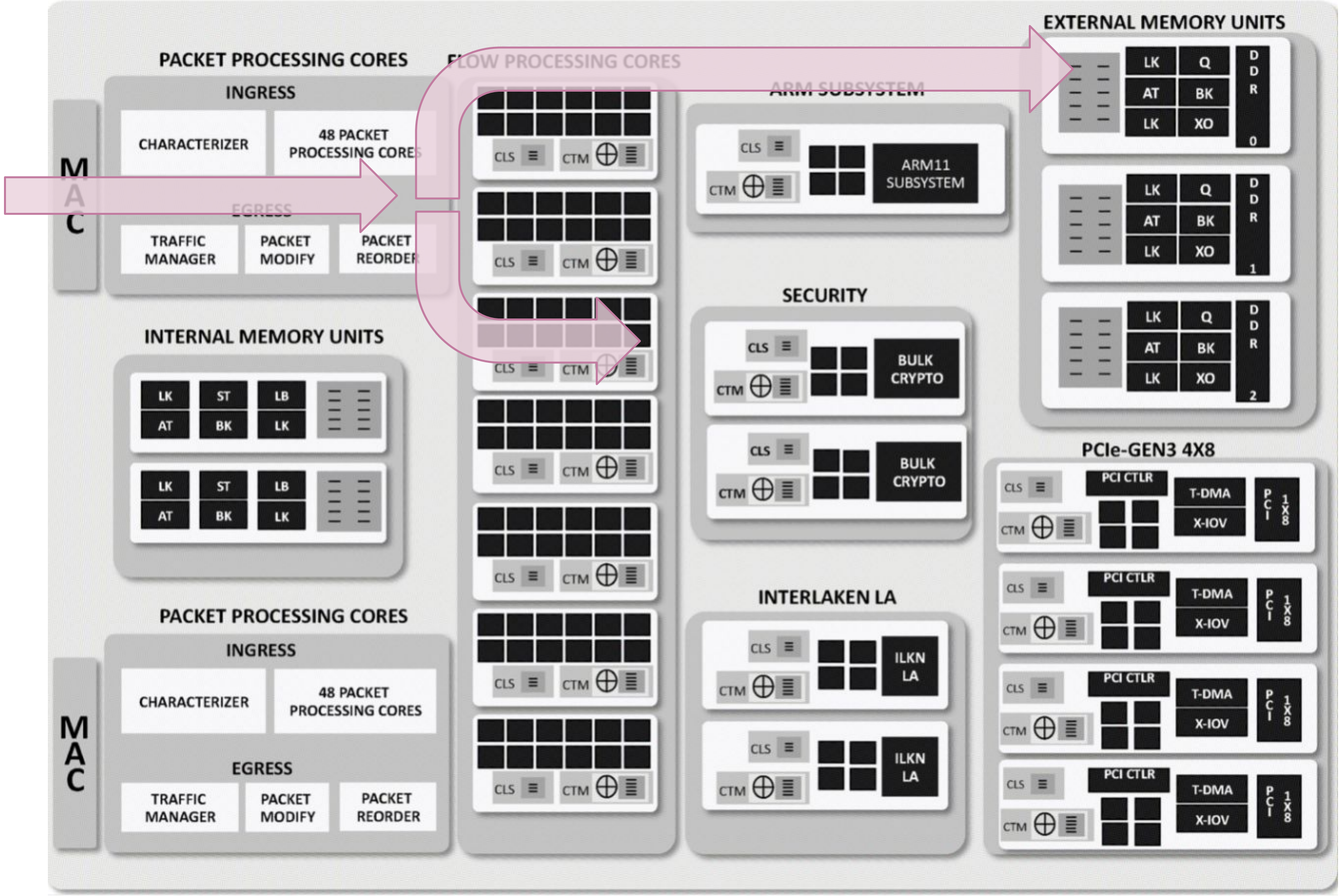
BPF maps

NFP SoC Packet Flow

NETRONOME

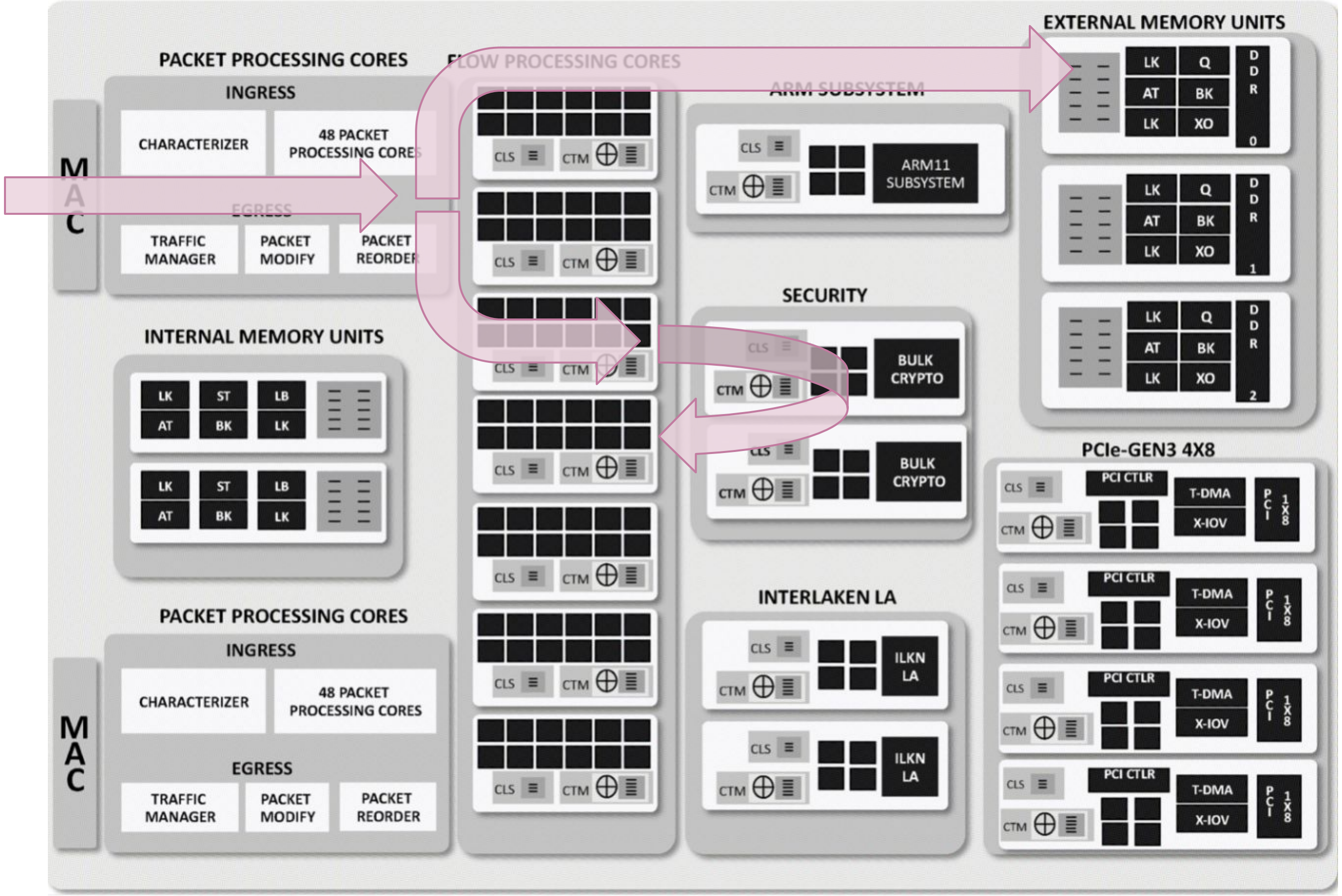


NFP SoC Packet Flow

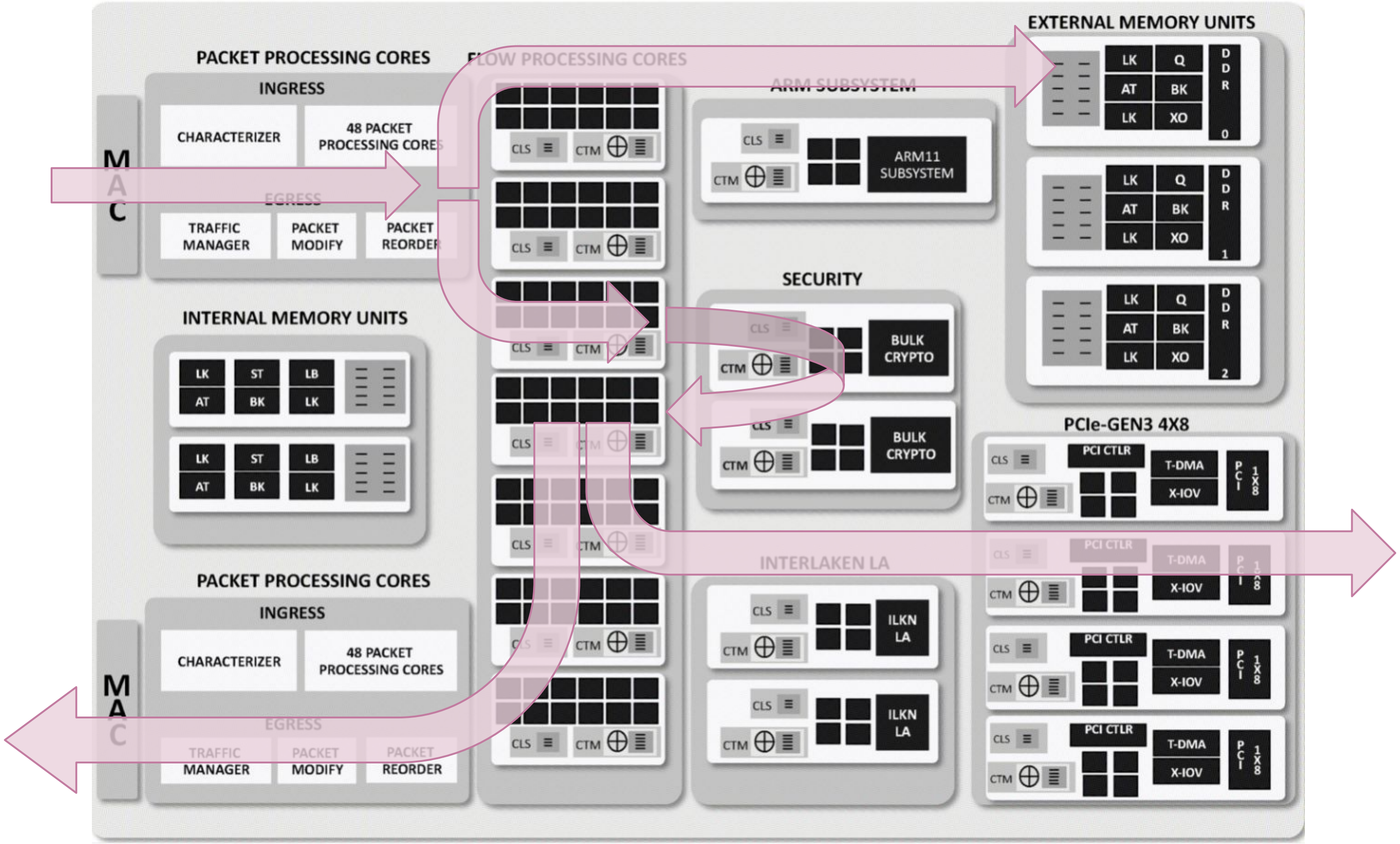


NFP SoC Packet Flow

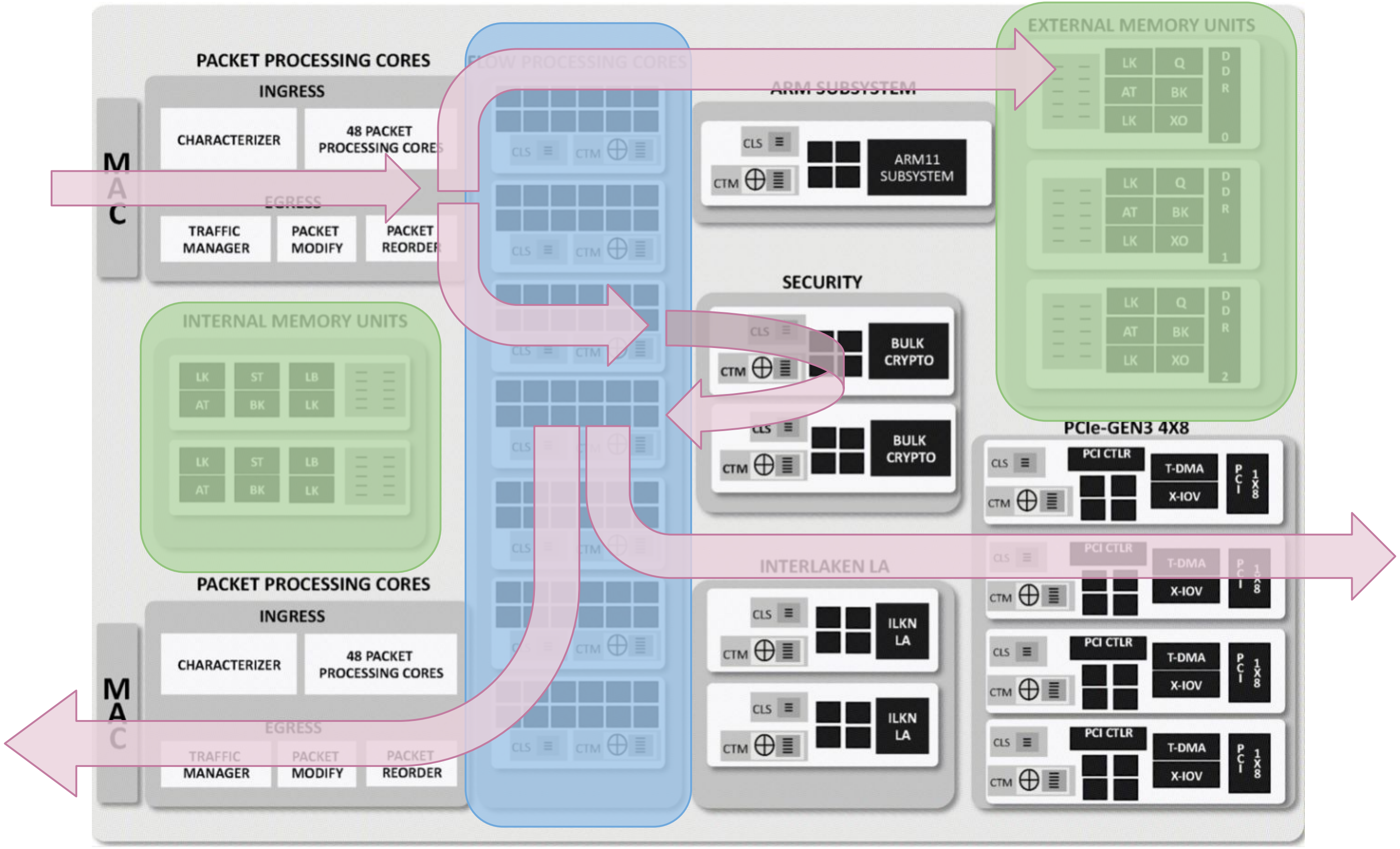
NETRONOME



NFP SoC Packet Flow



NFP SoC Packet Flow



Memory Architecture - Latencies

NETRONOME

GPRS/xfer regs - 1 cycle

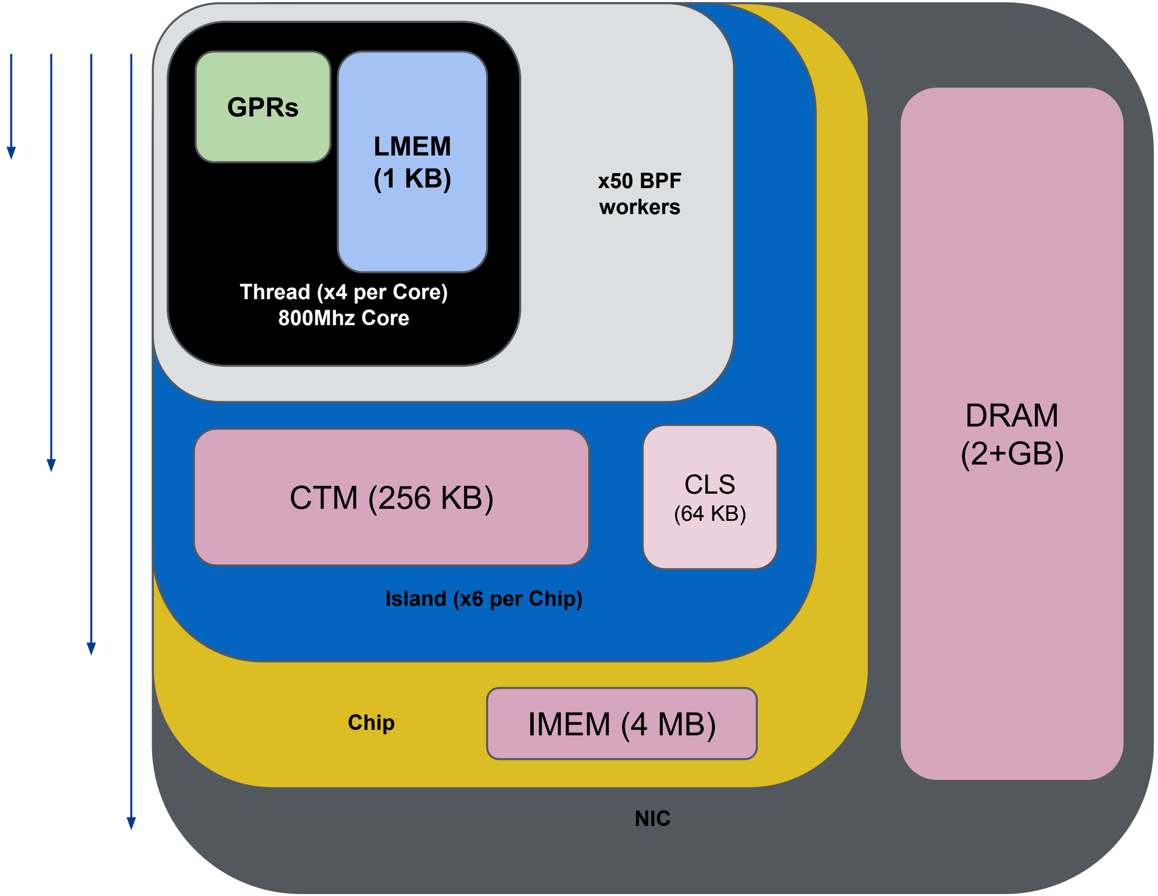
LMEM - 1-3 cycles

CLS - 20-50 cycles

CTM - 50-100 cycles

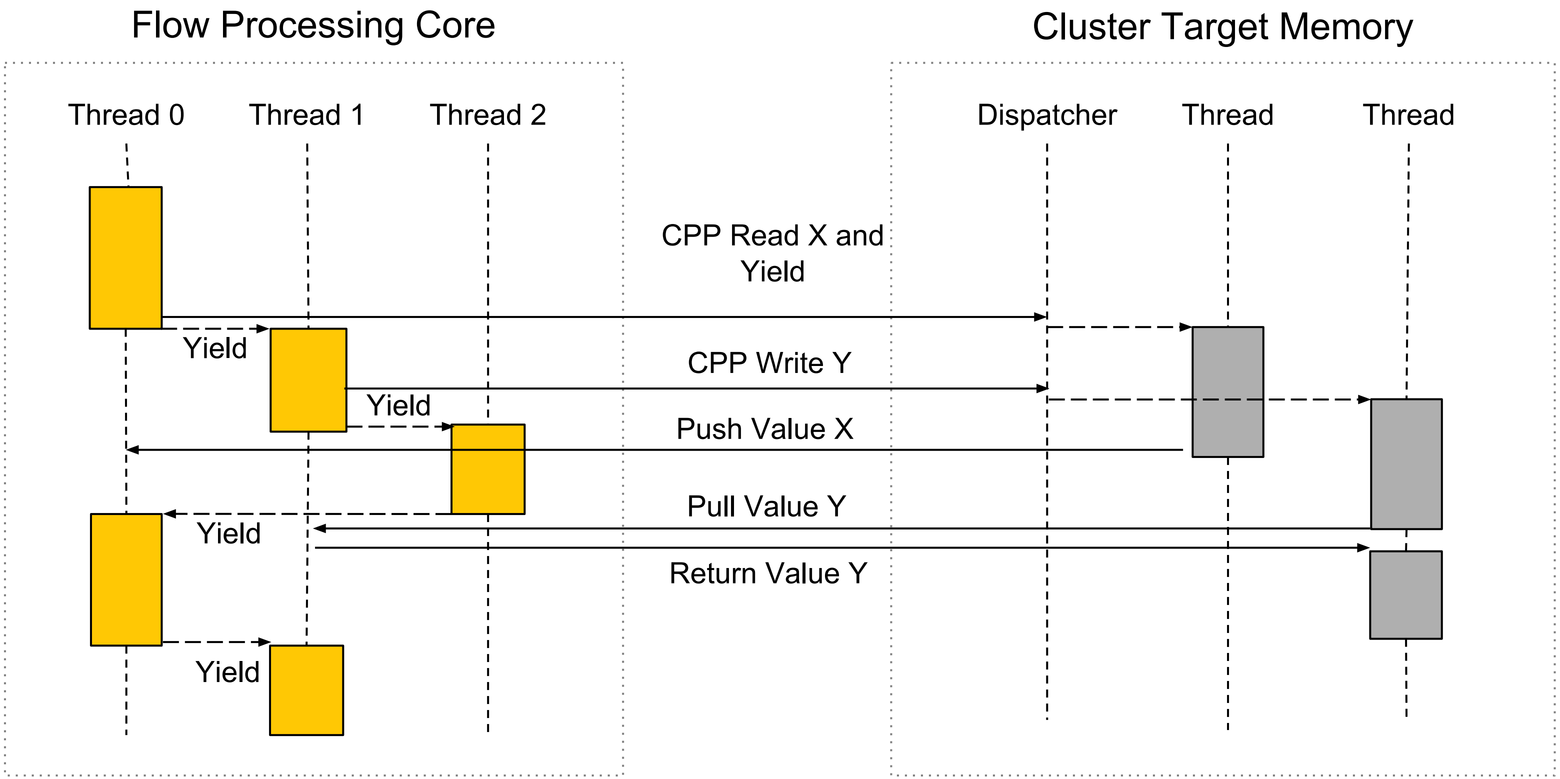
IMEM - 150-250 cycles

DRAM - 150-500 cycles



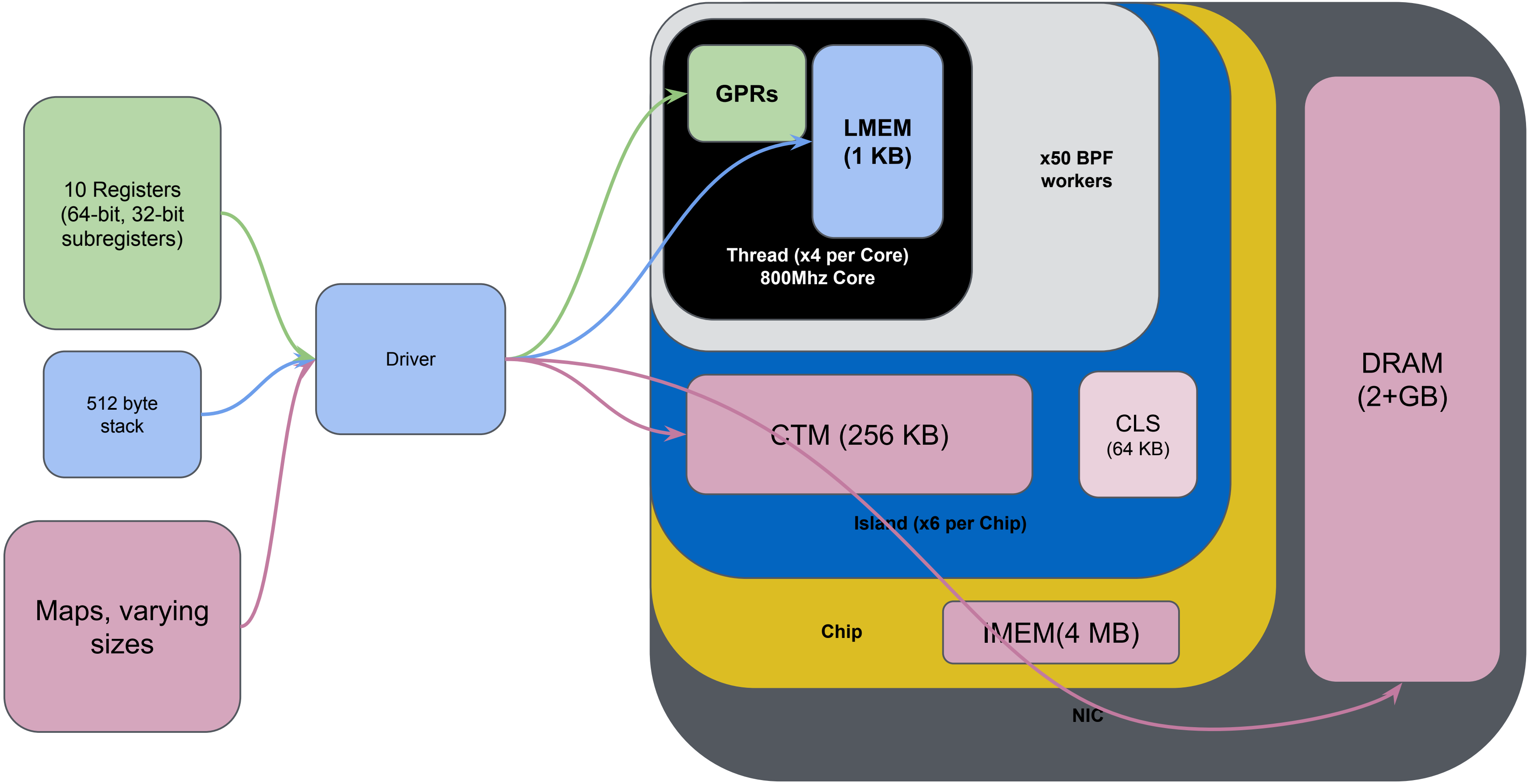
Memory Architecture

NETRONOME



Multithreaded Transactional Memory Architecture Hides Latency

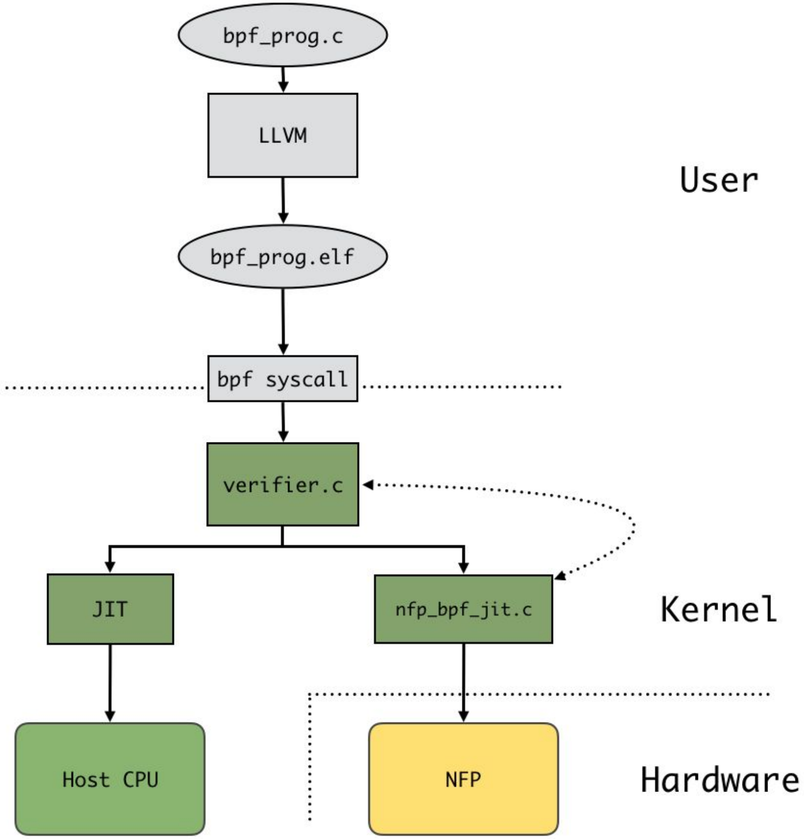
Kernel Offload - BPF Offload Memory Mapping



Programming Model

NETRONOME

- Program is written in standard manner
- LLVM compiled as normal
- iproute/tc/libbpf loads the program requesting offload
- The nfp_bpf_jit.c converts the BPF bytecode to NFP machine code (and we mean the actual machine code :))
- Translation reuses a significant amount of verifier infrastructure



BPF Object Creation (maps)

NETRONOME

1. Get map file descriptors:
   a. For existing maps - get access to a file descriptor:
      i. from bpffs (pinned map) - open a pseudo file
      ii. by ID - use BPF_MAP_GET_FD_BY_ID bpf syscall command
   b. Create new maps - BPF_MAP_CREATE bpf syscall command:

```
union bpf_attr {
        struct { /* anonymous struct used by BPF_MAP_CREATE command */
                __u32  map_type;    /* one of enum bpf_map_type */
                __u32  key_size;    /* size of key in bytes */
                __u32  value_size;  /* size of value in bytes */
                __u32  max_entries; /* max number of entries in a map */
                __u32  map_flags;   /* BPF_MAP_CREATE related
                                     * flags defined above.
                                     */
                __u32  inner_map_fd;/* fd pointing to the inner map */
                __u32  numa_node;   /* numa node (effective only if
                                     * BPF_F_NUMA_NODE is set).
                                     */
                char   map_name[BPF_OBJ_NAME_LEN];
                __u32  map_ifindex; /* ifindex of netdev to create on */
                __u32  btf_fd;             /* fd pointing to a BTF type data */
                __u32  btf_key_type_id;    /* BTF type_id of the key */
                __u32  btf_value_type_id;  /* BTF type_id of the value */
        };
```

BPF object creation (programs)

NETRONOME

1. Get program instructions;
2. Perform relocations (replace map references with file descriptors IDs);
3. Use BPF_PROG_LOAD to load the program;

```
union bpf_attr {
      struct { /* anonymous struct used by BPF_PROG_LOAD command */
            __u32          prog_type;    /* one of enum bpf_prog_type */
            __u32          insn_cnt;
            __aligned_u64 insns;
            __aligned_u64 license;
            __u32          log_level;    /* verbosity level of verifier */
            __u32          log_size;     /* size of user buffer */
            __aligned_u64 log_buf;      /* user supplied buffer */
            __u32          kern_version; /* checked when prog_type=kprobe */
            __u32          prog_flags;
            char           prog_name[BPF_OBJ_NAME_LEN];
            __u32          prog_ifindex; /* ifindex of netdev to prep for */
            /* For some prog types expected attach type must be known at
             * load time to verify attach type specific parts of prog
             * (context accesses, allowed helpers, etc).
             */
            __u32          expected_attach_type;
      };
```

BPF Object Creation (`libbpf`)

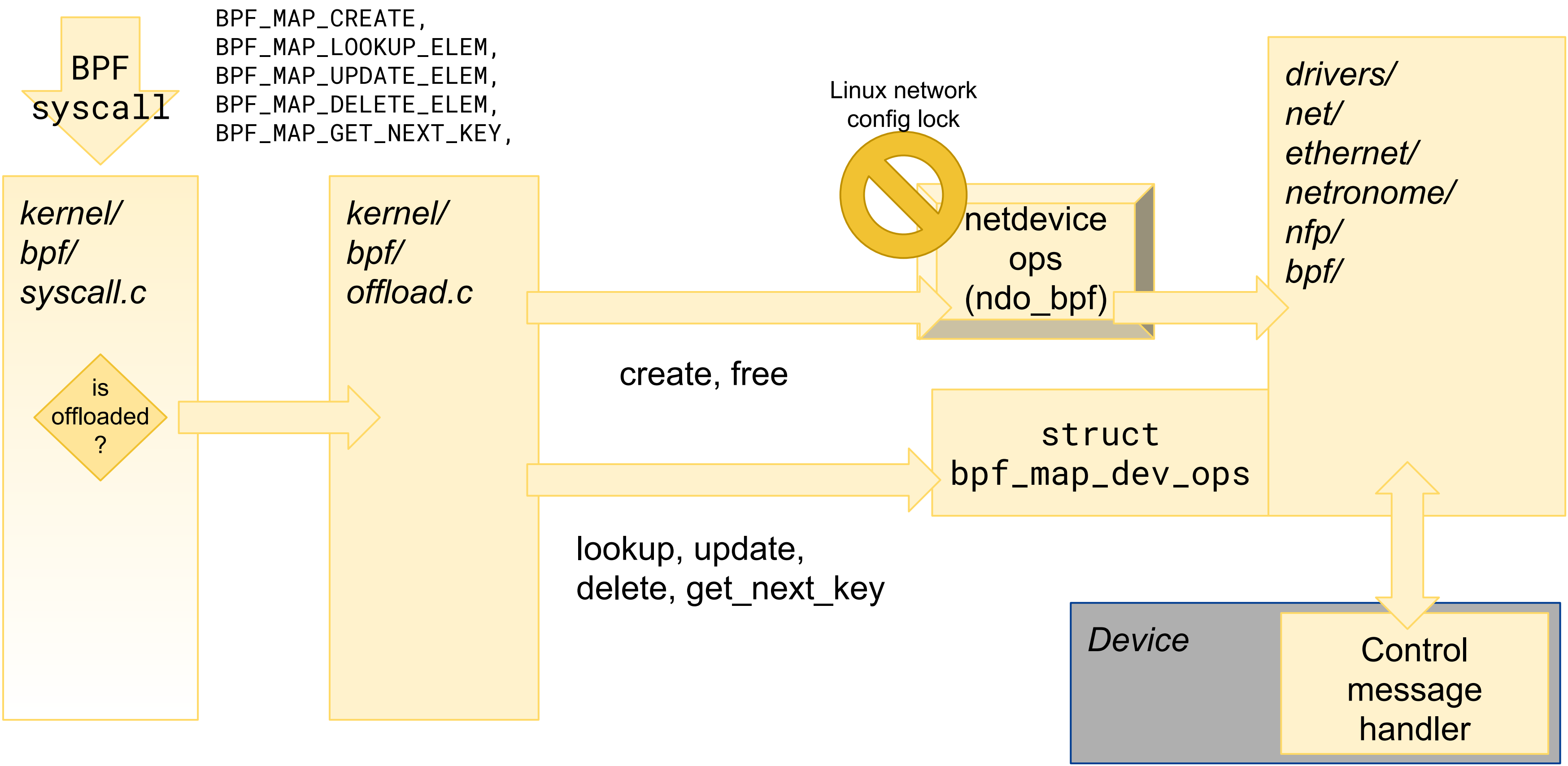
- With `libbpf` use the extended attributes to set the ifindex:

```
struct bpf_prog_load_attr {
        const char *file;
        enum bpf_prog_type prog_type;
        enum bpf_attach_type expected_attach_type;
        int ifindex;
};

int bpf_prog_load_xattr(const struct bpf_prog_load_attr *attr,
                        struct bpf_object **pobj, int *prog_fd);
```

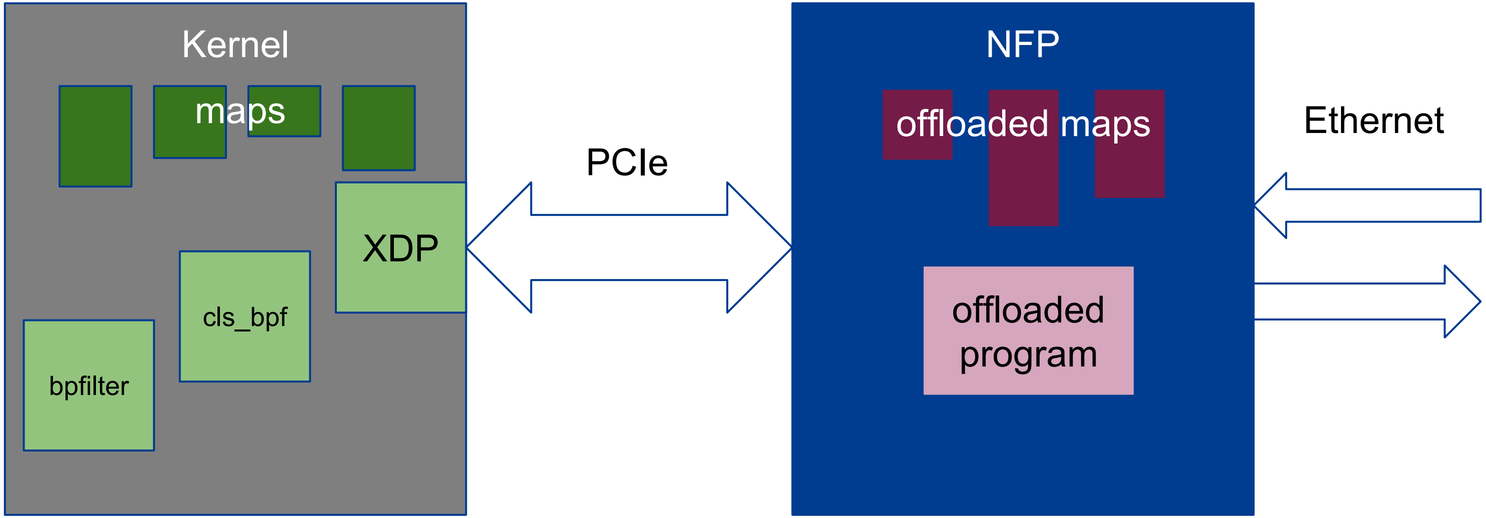
Normal kernel BPF ABIs are used, opt-in for offload by setting ifindex.

Map Offload


BPF syscall
BPF_MAP_CREATE,
BPF_MAP_LOOKUP_ELEM,
BPF_MAP_UPDATE_ELEM,
BPF_MAP_DELETE_ELEM,
BPF_MAP_GET_NEXT_KEY,
kernel/ bpf/ syscall.c
is offloaded ?
kernel/ bpf/ offload.c
Linux network config lock
netdevice ops (ndo_bpf)
create, free
struct bpf_map_dev_ops
lookup, update, delete, get_next_key
drivers/ net/ ethernet/ netronome/ nfp/ bpf/
Device
Control message handler

Map Offload

- Maps reside entirely in device memory
- Programs running on the host do not have access to offloaded maps and vice versa (because host cannot efficiently access device memory)
- User space API remains unchanged



Map Offload

NETRONOME

- Each map in the kernel has set of ops associated:

```
/* map is generic key/value storage optionally accessible by eBPF programs */
struct bpf_map_ops {
	/* funcs callable from userspace (via syscall) */
	int (*map_alloc_check)(union bpf_attr *attr);
	struct bpf_map *(*map_alloc)(union bpf_attr *attr);
	void (*map_release)(struct bpf_map *map, struct file *map_file);
	void (*map_free)(struct bpf_map *map);
	int (*map_get_next_key)(struct bpf_map *map, void *key, void *next_key);

	/* funcs callable from userspace and from eBPF programs */
	void *(*map_lookup_elem)(struct bpf_map *map, void *key);
	int (*map_update_elem)(struct bpf_map *map, void *key, void *value, u64 flags);
	int (*map_delete_elem)(struct bpf_map *map, void *key);
};
```

- Each map type (array, hash, LRU, LPM, etc.) has its own set of ops which implement the map specific logic
- If `map_ifindex` is set the ops are pointed to an empty set of “offload ops” regardless of the type (`bpf_offload_prog_ops`)
- Only calls from user space will now be allowed

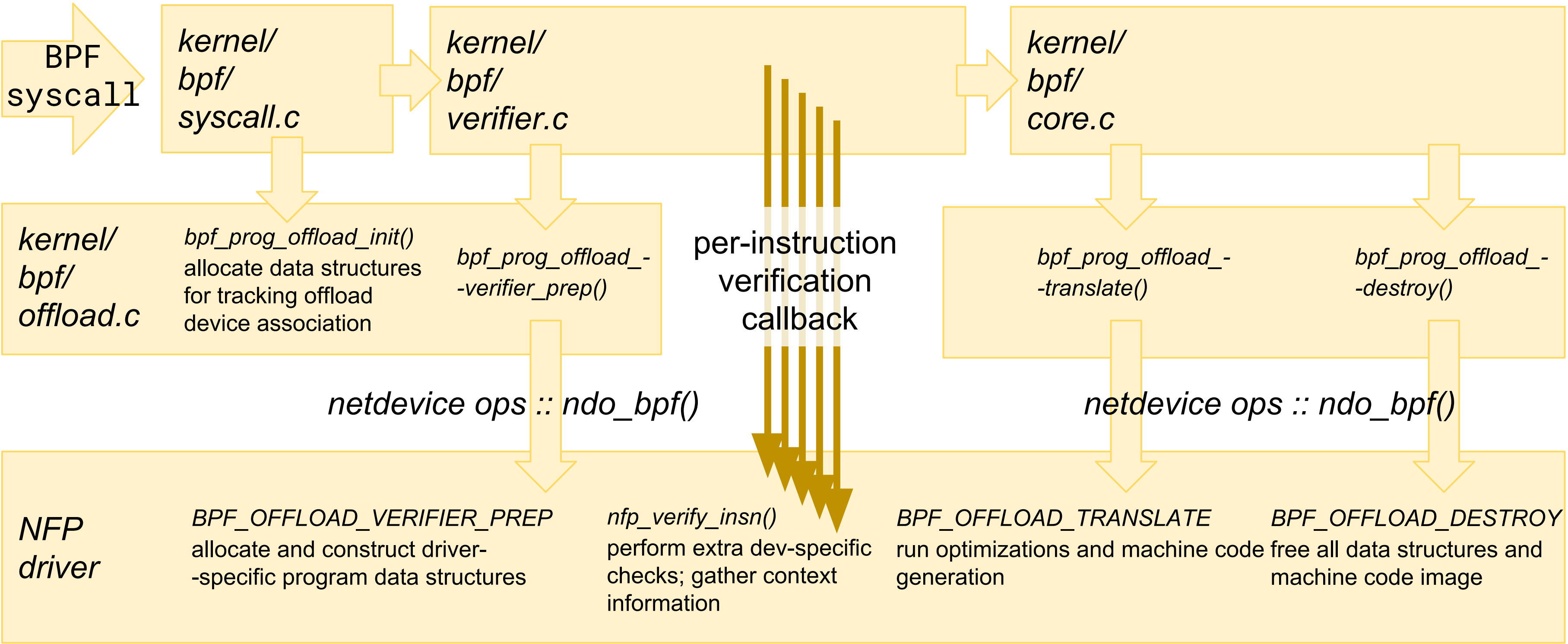
Program Offload

NETRONOME

- Kernel verifier performs verification and some of common JIT steps for the host architectures
- For offload these steps cause loss of context information and are incompatible with the target
- Allow device translator to access the loaded program as-is:
  - IDs/offsets not translated:
    - structure field offsets
    - functions
    - map IDs
  - No prolog/epilogue injected
  - No optimizations made

For offloaded devices the verifier skips the extra host-centric rewrites.

Program Offload Lifecycle



Program Offload

- After program has been loaded into the kernel the subsystem specific handling remains unchanged
- For network programs offloaded program can be attached to device ingress to XDP (`BPF_PROG_TYPE_XDP`) or `cls_bpf` (`BPF_PROG_TYPE_SCHED_CLS`)
- Program can be attached to any of the ports of device for which it was loaded
- Actually loading program to device memory only happens when it's being attached

BPF Offload - Summary

NETRONOME

- BPF VM/sandbox is well suited for a heterogeneous processing engine
- BPF offload allows loading a BPF program onto a device instead of host CPU
- All user space tooling and ABIs remain unchanged
- No vendor-specific APIs or SDKs
- BPF offload is part of the upstream Linux kernel (recent kernel required)
- BPF programs loaded onto device can take advantage of HW accelerators such as HW memory lookup engines
- Try it out today on standard NFP server cards! (academic pricing available on open-nfp.org 😊)
- Reach out with BPF-related questions:
  - https://help.netronome.com/a/forums/
  - https://groups.google.com/forum/#!forum/open-nfp
  - xdp-newbies@vger.kernel.org
- Register for the next webinar in this series!