New Standards for Next Generation Computational Storage Devices

Brian Myungjune JUNG
Samsung Electronics
Oct. 23, 2020

Topics

New Standards for the **Next Generation Computational Storage Devices**

- SNIA Computational Storage (CS) Programming Model and API
- Extended Berkeley Packet Filter (eBPF)
- Compute Express Link (CXL)

What is Computational Storage (CS)?

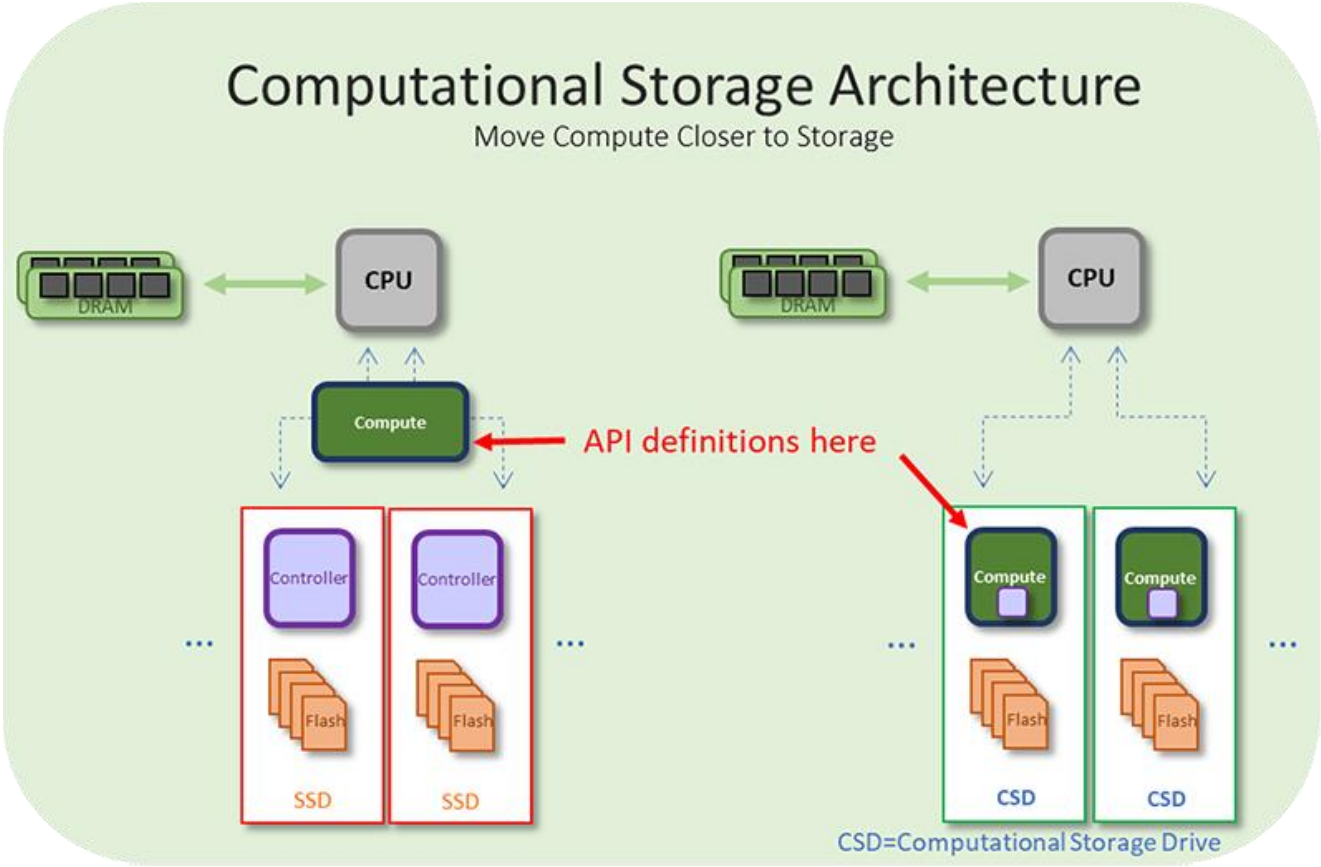
Definitions

- Computational Storage is defined as architectures that provide Computational Storage Services coupled to storage, offloading host processing, or reducing data movement

< "What is Computational Storage?", SNIA >

- Computational storage adds computing capabilities to traditional storage devices and systems, enabling processing to be carried out directly on the storage drive

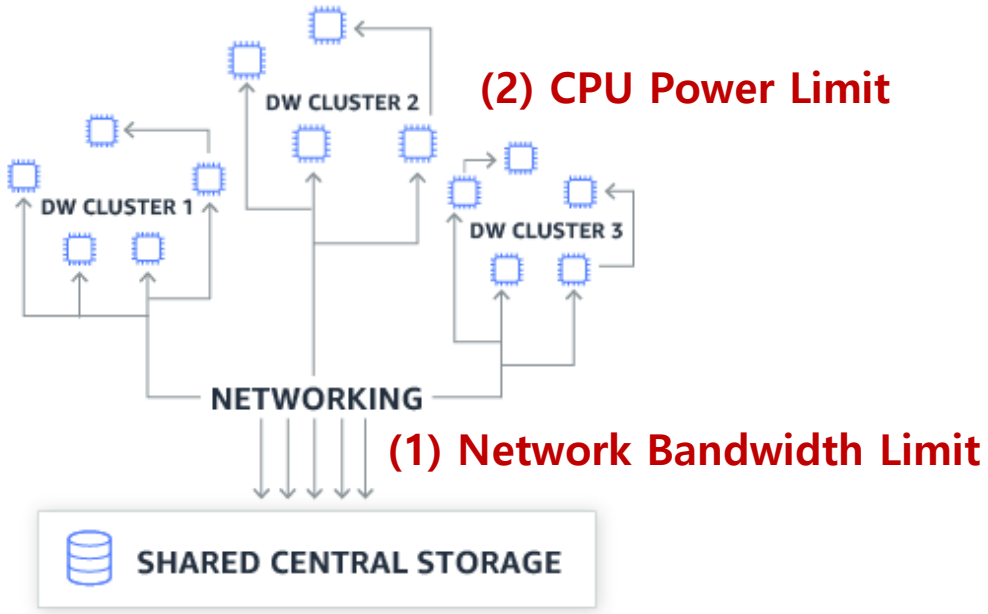
< "A Guide to Computational Storage on ARM", ARM >



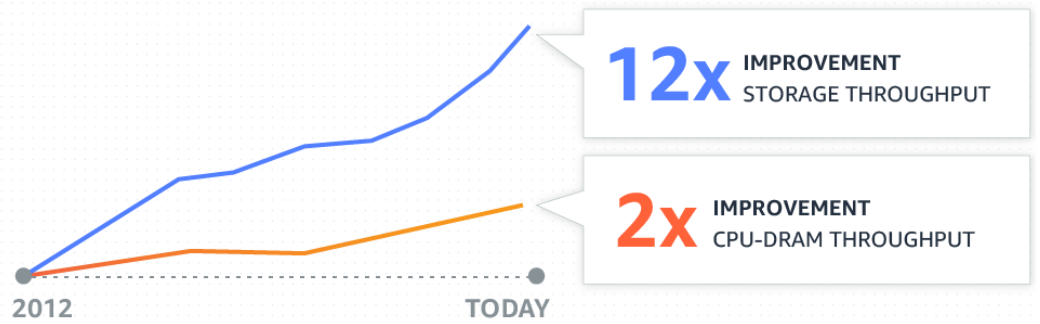
< Source: https://www.snia.org/education/what-is-computational-storage >

Datacenter Pain Points:

The first is network and the second is CPU



Existing Data Warehousing Architecture

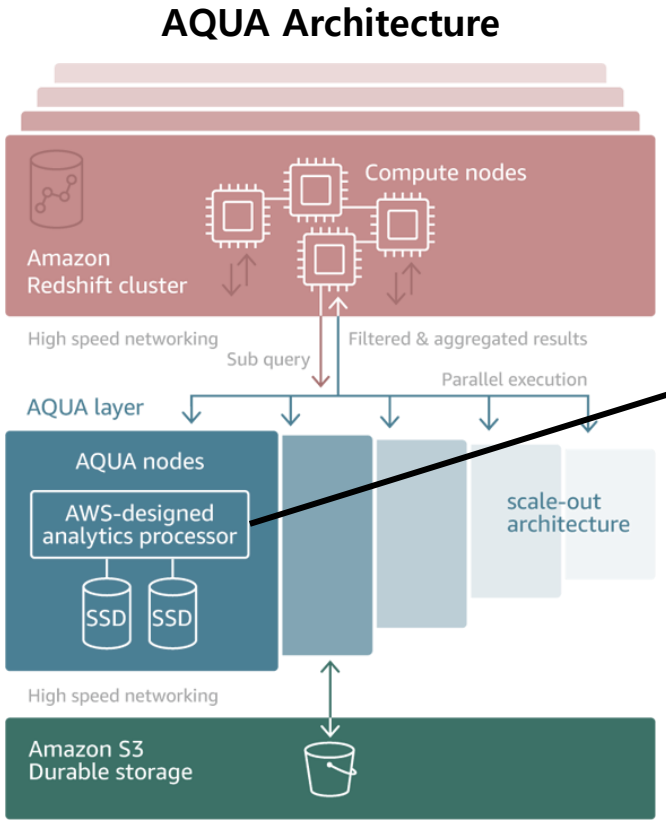


Hardware Trends Since 2012

< source: https://pages.awscloud.com/AQUA_Preview.html >

Strategy:

Bring compute closer to storage



(*) AQUA: Advanced Query Accelerator

AQUA* accelerates Redshift queries by running data intensive tasks such as filtering and aggregation closer to the storage layer.

※ AQUA uses AWS-designed processors to accelerate queries.
AWS Nitro chips adapted to speed up data encryption and compression,
and custom analytics processors, implemented in FPGAs, to accelerate operations such as filtering and aggregation.

-> Computational Storage, Finally?

The AQUA hardware sits inline in the network between the S3 storage and Redshift cluster, acting as a caching bump in the wire and also as a sub-query offload processing system

< source: https://pages.awscloud.com/AQUA_Preview.html >

News from ARM:

ARM's New CPU for Computational Storage - Cortex R82

Highest performance Arm Cortex-R processor to power the future of computational storage

September 03, 2020

By Neil Werdmuller, director of storage solutions at Arm

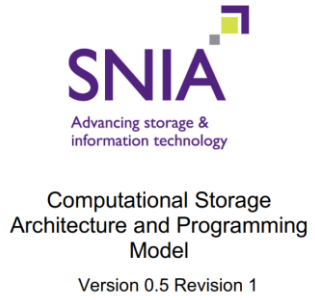
News highlights:

- Arm Cortex-R82 is the highest performance Cortex-R processor, with 64-bit support and Linux-capability
- Real-time processor enables data processing where it is stored for next-generation enterprise and computational storage solutions
- Combines higher performance and access to greater memory with extensive Arm Linux and server ecosystems

< source: https://www.arm.com/company/news/2020/09/highest-performance-arm-cortex-r-processor >

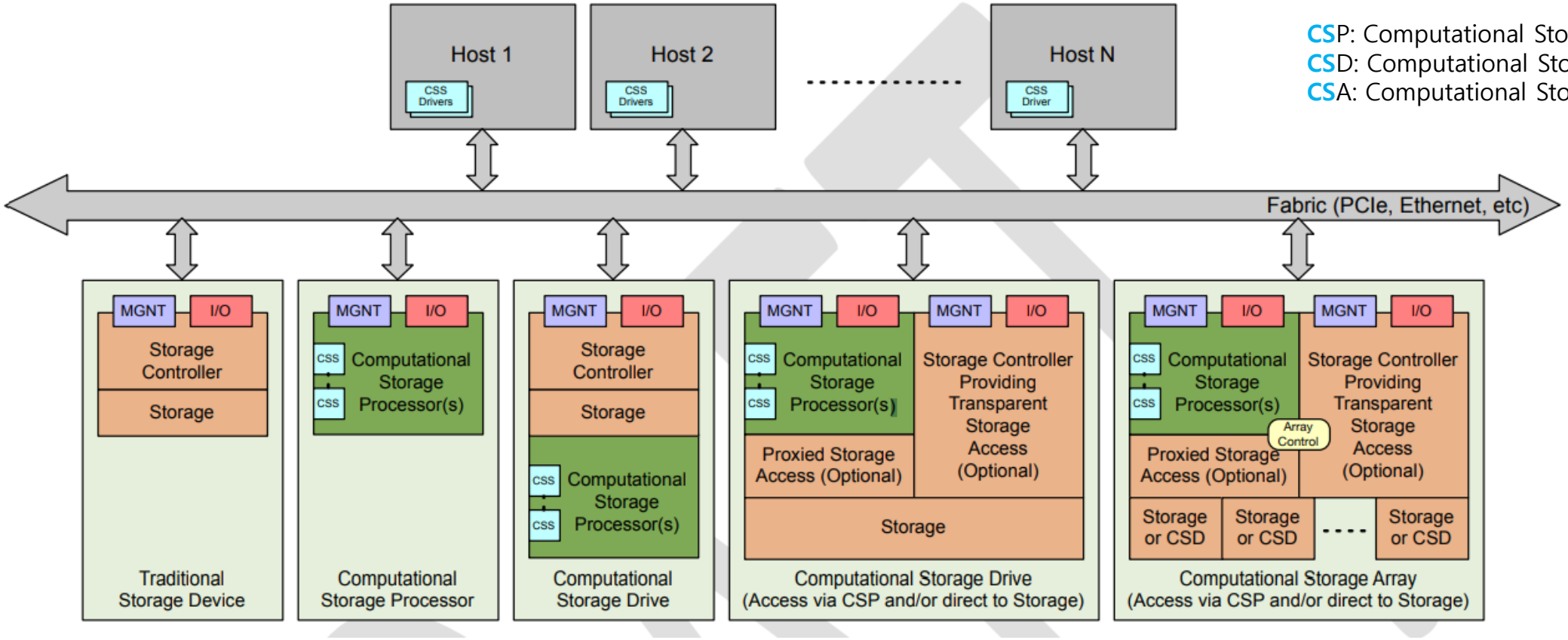
SNIA | COMPUTATIONAL STORAGE

Computational Storage (CS) Architecture / Programming Model (/ APIs)



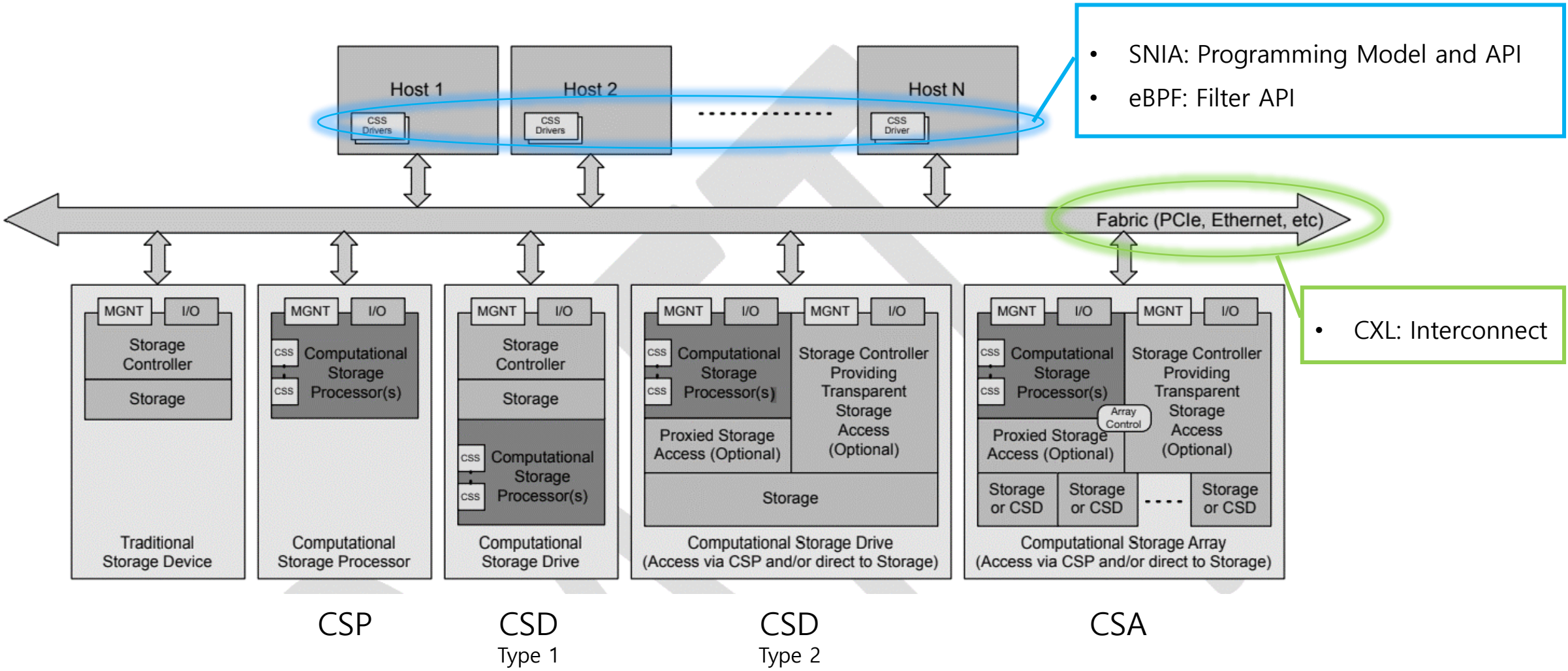
< source: SNIA Computational Storage Architecture and Programming Model Version 0.5 Revision 1, Working Draft, Aug. 6th, 2020 >

Computational Storage (CS) Architecture



CSP: Computational Storage Processor
CSD: Computational Storage Drive
CSA: Computational Storage Array

CS Related Standards to focus on

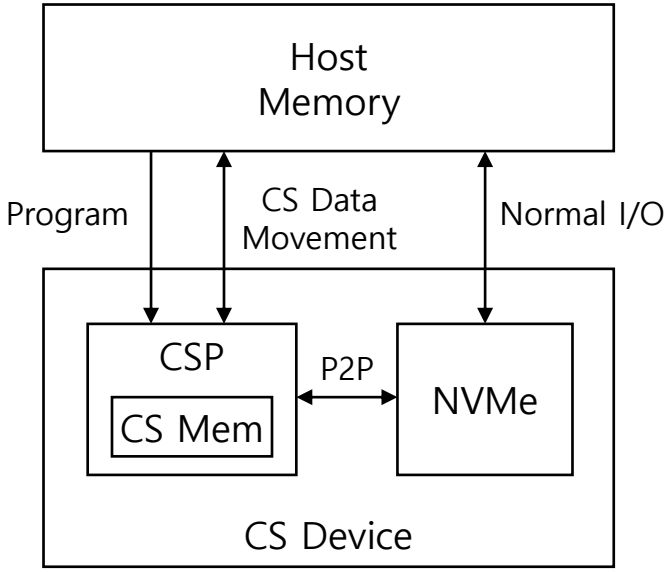


- SNIA: Programming Model and API
- eBPF: Filter API

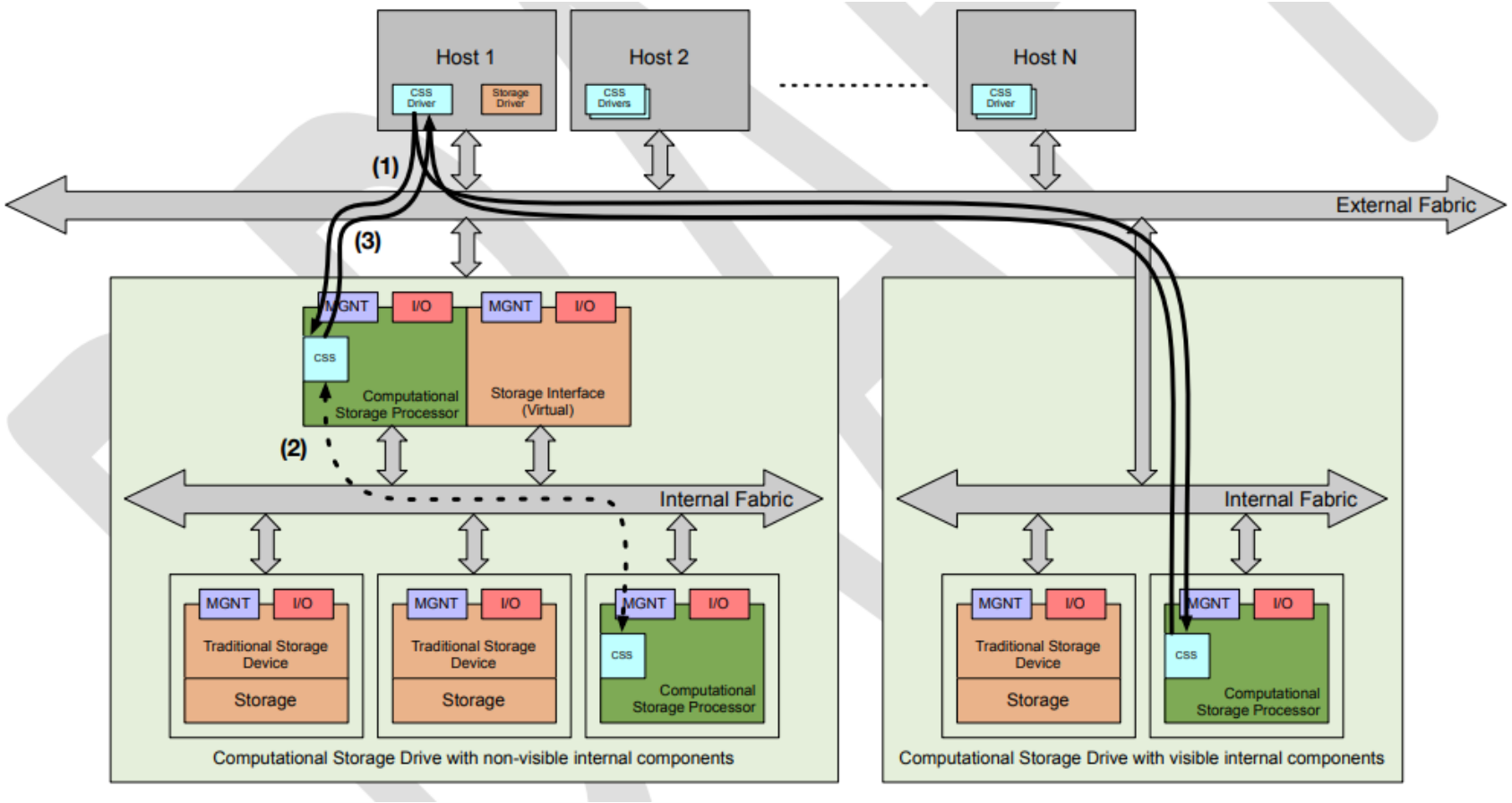
- CXL: Interconnect

CS Operations (APIs)

- **Device Discovery**
  - Identify CSx Devices
- **Device Access**
  - Open, Close
- **CS Memory Management**
  - Allocate/Deallocate CS Memory
- **Data Movement**
  - Transfer data between host memory and CS memory area
  - Transfer (P2P) data between CS memory area and SSD
- **CSx Scheduling**
  - Schedule compute offload to device
- **General Device Management**
  - Download programs
  - Query device properties & capabilities
  - Configure device functionalities

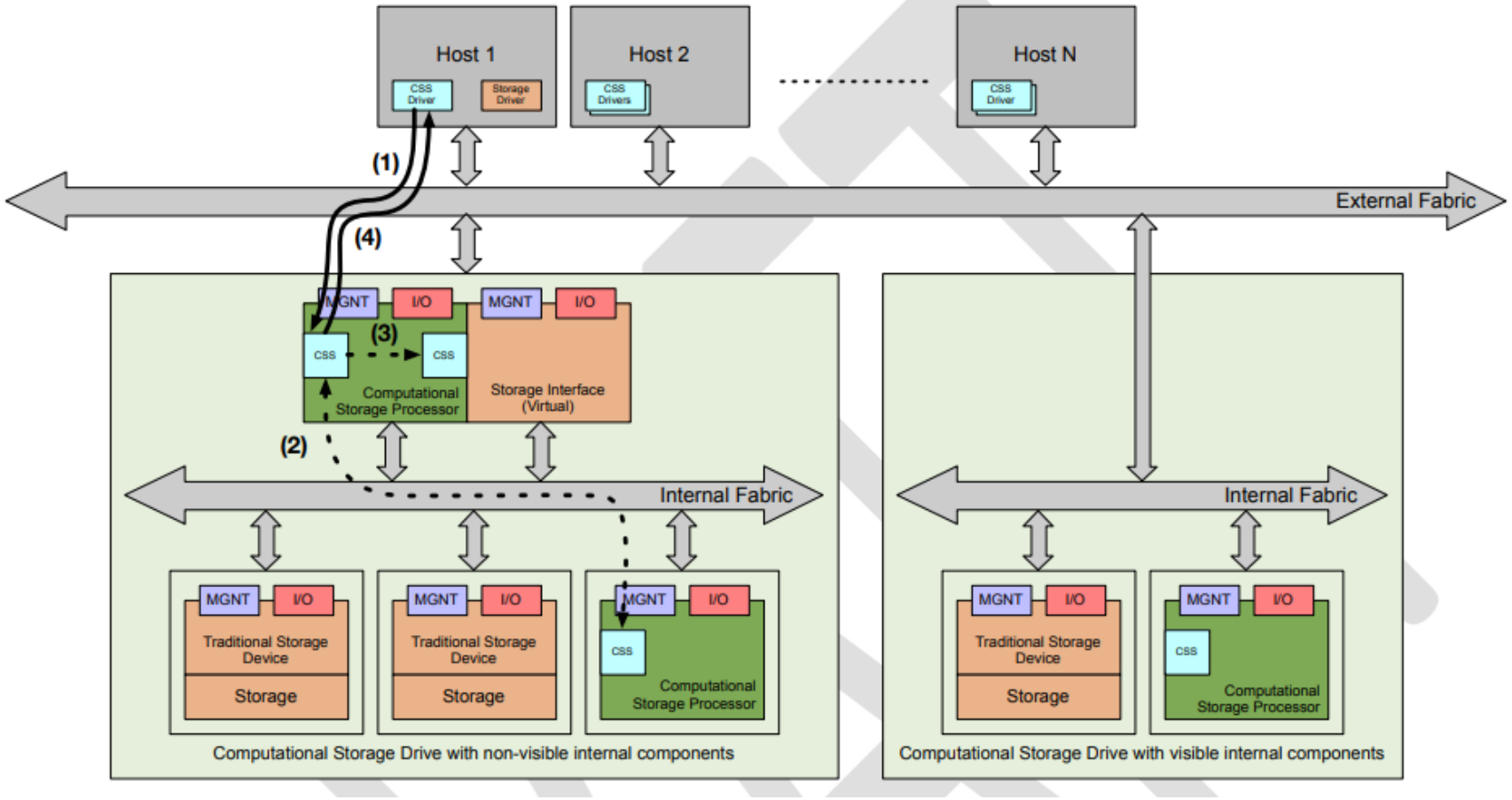


CS Operation Example: Discovery



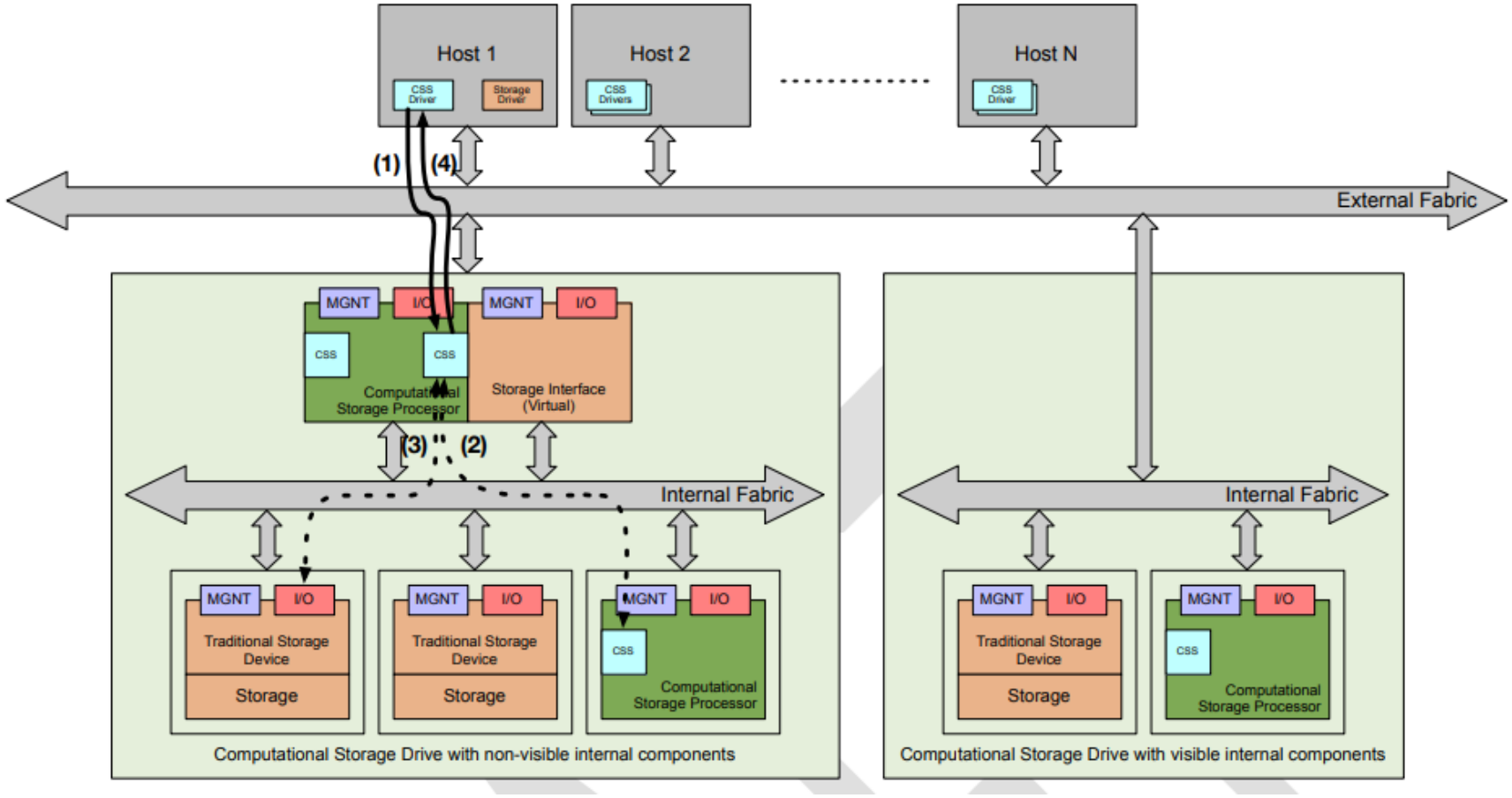
(1) The host sends a CSS discovery request over the fabric to one or more CSxes.
(2) CSxes may repeat this discovery process for internally accessible CSxes, which may use the standardized CSx/CSS discovery process.
(3) Each CSx that accepts the discovery request returns a CSS discovery response to the requesting host.

CS Operation Example: Configuration



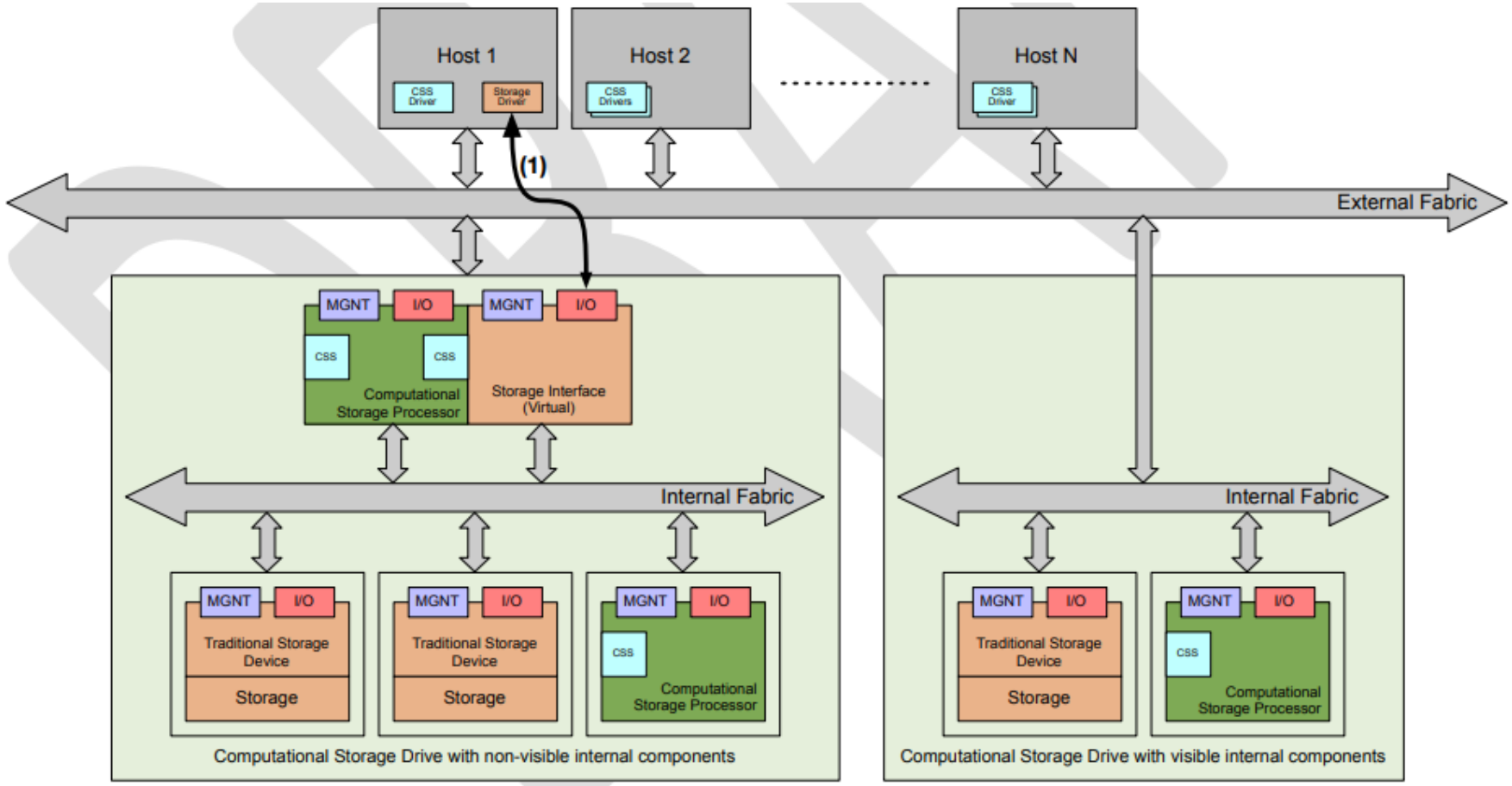
(1) The host sends a CSS configuration request over the fabric to a target CSx
(2) The target CSx may repeat the configuration process with internally accessible CSxes.
(3) For Programmable CSSes, the configuration process may result in the creation of one or more new Programmable and/or Fixed CSSes.
(4) The target CSx returns a CSS configuration response to the requesting host.

CS Operation Example: Direct Usage



(1) The host sends a CSS command to a target CSS.
(2) The target CSS may send one or more commands to other internally accessible CSSes.
(3) The target CSS may send one or more commands to other storage or memory devices to retrieve and/or store data.
(4) The target CSS returns a CSS command response to the requesting host.

CS Operation Example: Transparent Usage



(1) The host sends unmodified storage interface operations to a target Storage Interfaced that is associated with the target CSS.

CS Services: Fixed and Programmable

Fixed CS Services

- **Compression**
- **Data Filter**
- **Encryption**
- **Erasure Coding**
- **RegEx**
- **Scatter-Gather**
- **Pipeline**
- **Video Compression**
- **Hash/CRC**
- **Data Deduplication**

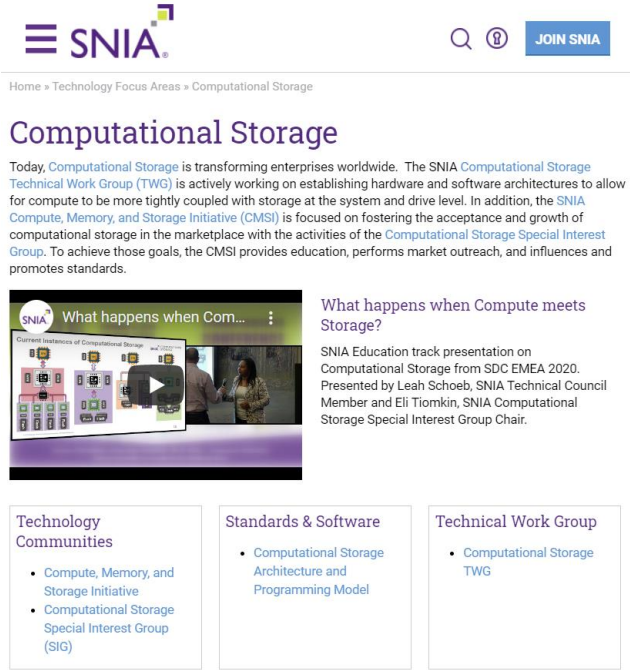
Programmable CS Services

- **Operating System Image Loader**
- **Container Image Loader**
- **eBPF Loader**
- **FPGA Bitstream Loader**
- **Large Data Set**

SNIA CS Technical Work Group

Introduction

- 45 participating companies / 202 member representatives
- Focus on definition list to ensure the TWG covers question on what computational storage is and what its products can be
- Drive to a scope and path to a universal usage model
- SNIA's Computational Storage Technical Work Group is developing a **Computational Storage Architecture and Programming Model** – defining recommended behavior for hardware and software that support computational storage



SNIA
Advancing storage & information technology

Computational Storage Architecture and Programming Model
Version 0.5 Revision 1

SNIA CS Technical Work Group

45 participating companies / 202 member representatives

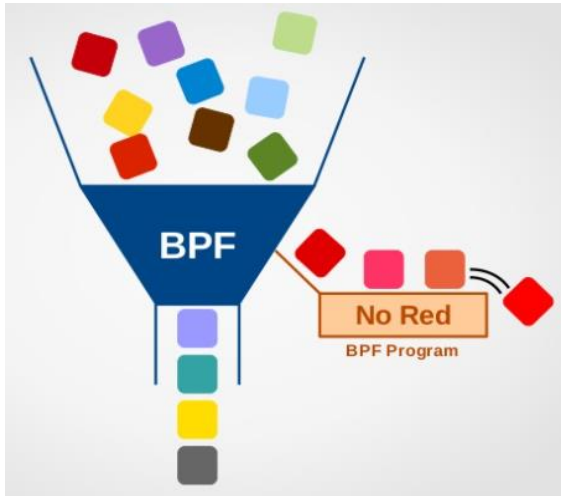
eBPF (extended Berkeley Packet Filter)

Source: https://ebpf.io/

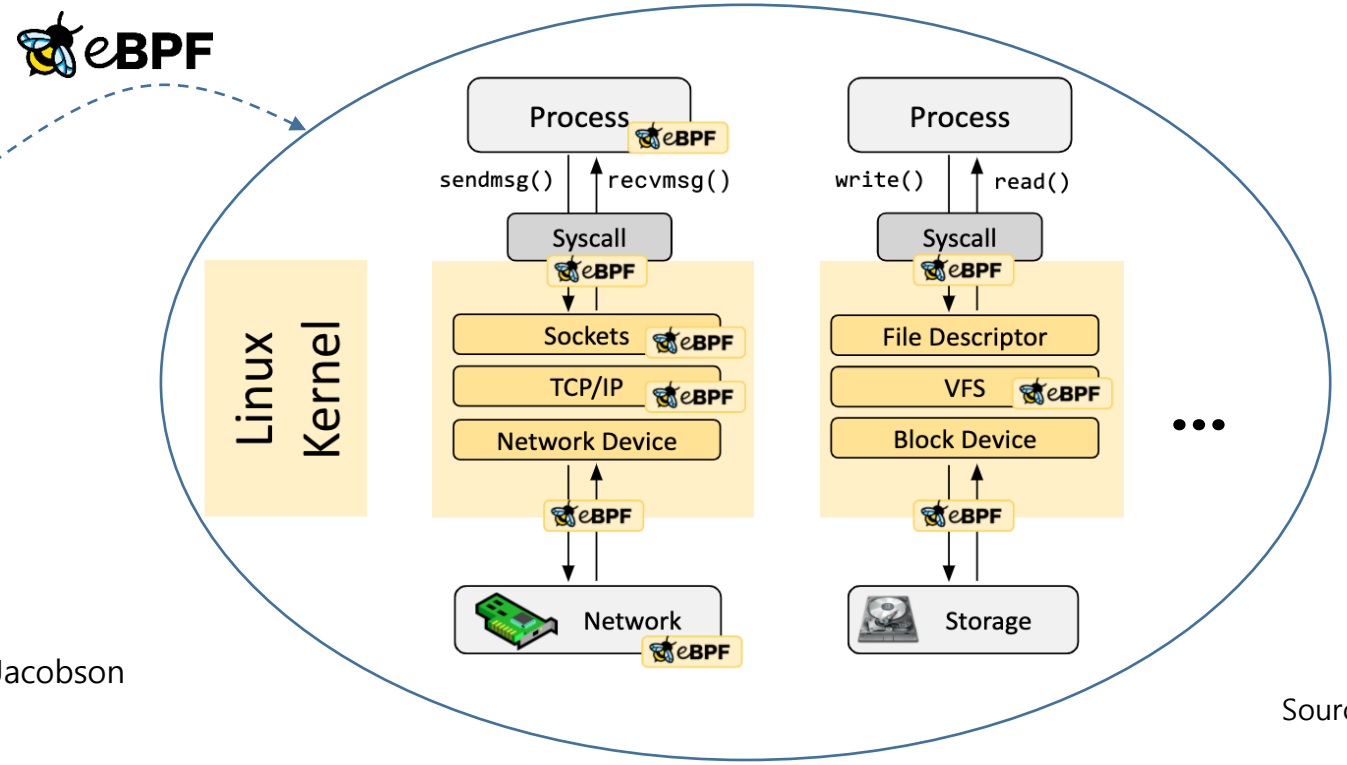
What is eBPF?

: extended Berkeley Packet Filter

- eBPF is a small virtual machine which runs programs injected from user space and attached to specific hooks in the kernel



Network Packet Filtering
- Classic BPF, introduced in 1992 by Van Jacobson



Source: https://ebpf.io/

Why so special?

: New generation of tooling

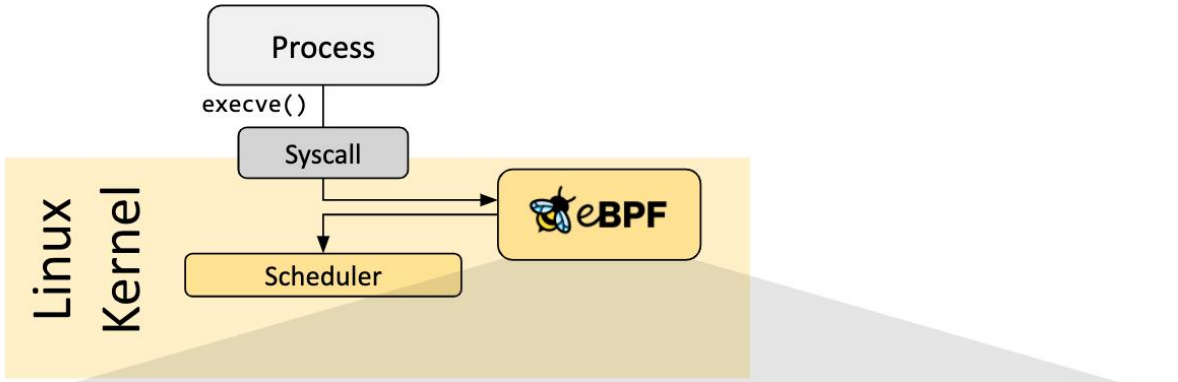
- By making the Linux kernel programmable, infrastructure software can leverage existing layers, making them more intelligent and feature-rich without continuing to add additional layers of complexity to the system.

- eBPF has resulted in the development of a completely new generation of tooling in areas such as **networking, security, application profiling/tracing and performance troubleshooting** that no longer rely on existing kernel functionality but instead actively reprogram runtime behavior without compromising execution efficiency or safety.

Source: https://ebpf.io/

eBPF Hook Overview

: eBPF programs are event-driven and are run when reached to certain hook point




```c
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```

Pre-defined hooks include

- system calls
- function entry/exit
- kernel tracepoints
- network events
- …

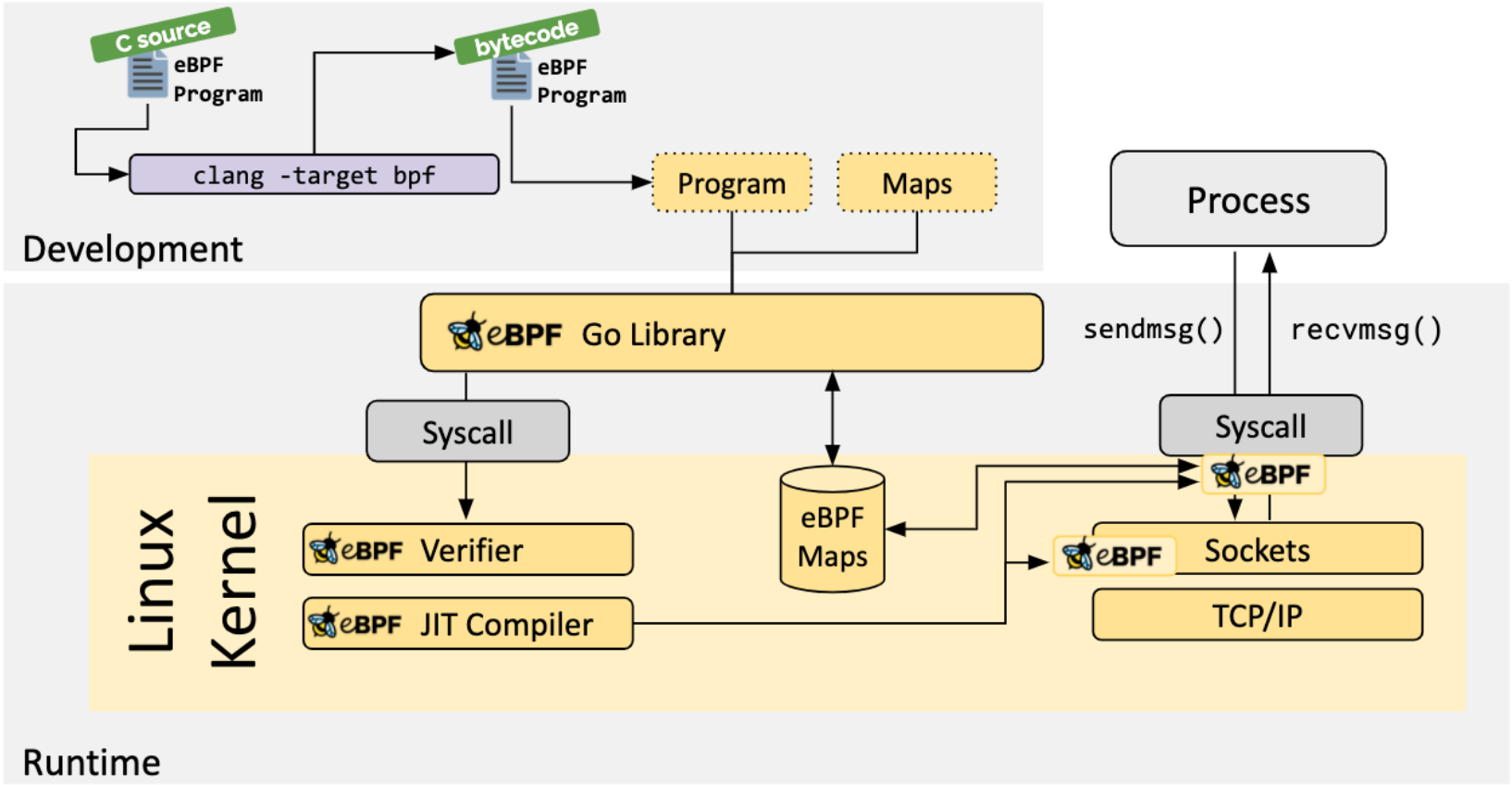
Customizability

- If a pre-defined hook does not exist for a particular need, it is possible to create a kernel probe (kprobe) or user probe (uprobe) to attach eBPF programs almost anywhere in kernel or user applications

Source: https://ebpf.io/

How are eBPF programs written?

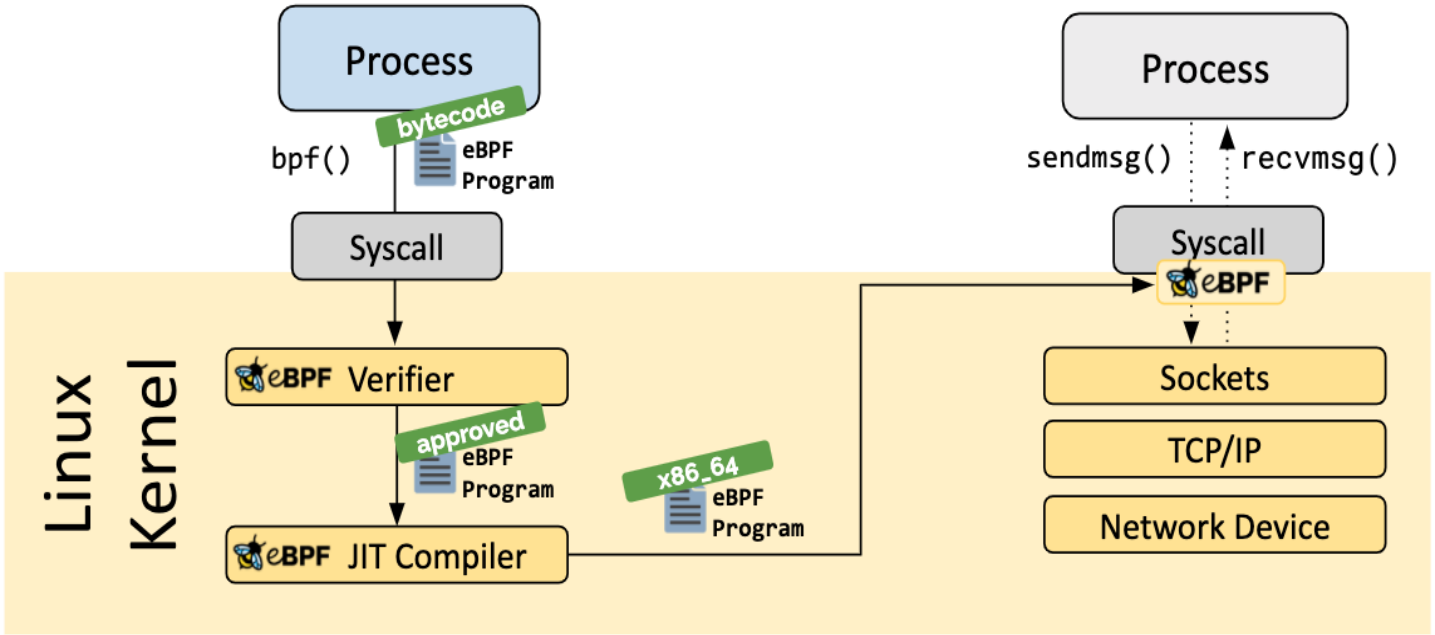
: High level user codes -> eBPF Bytecodes -> Machine specific instruction sets



Source: https://ebpf.io/

Loader & Verification Architecture

: eBPF bytecode loading -> verification -> JIT compilation



Loading

- eBPF program can be loaded into the Linux kernel using the bpf() system call
- As the program is loaded into the Linux kernel, it passes through two steps before being attached to the requested hook

Verification

- The verification step ensures that the eBPF program is safe to run

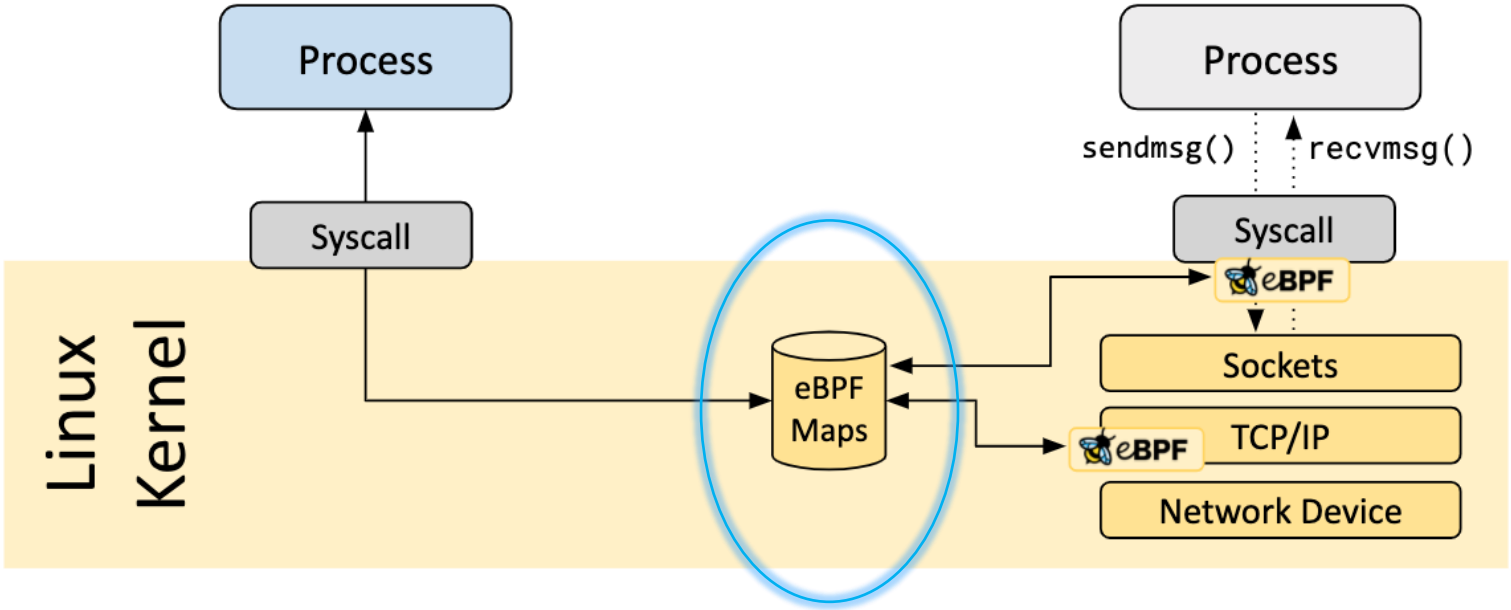
JIT Compilation

- This step translates the generic bytecode of the program into the machine specific instruction set
- This makes eBPF programs run as efficiently as natively compiled kernel code or as code loaded as a kernel module.

Source: https://ebpf.io/

eBPF Maps

: to store and retrieve data in a wide set of data structures



eBPF maps

- can be accessed from eBPF programs as well as from applications in user space via a system call

Supported map types

- Hash tables, Arrays
- LRU (Least Recently Used)
- Ring Buffer
- Stack Trace
- LPM (Longest Prefix match)
- ...

Source: https://ebpf.io/

Helper Calls

: to help eBPF programs which cannot call into kernel functions directly

Why eBPF helper calls?

- eBPF programs cannot call into arbitrary kernel functions
- Instead, eBPF programs can make function calls into helper functions, a well-known and stable API offered by the kernel

Helper call examples:

- Generate random numbers
- Get current time & date
- eBPF map access
- Get process/cgroup context
- Manipulate network packets and forwarding logic

Source: https://ebpf.io/

eBPF Example Codes

: bpf+sockets example

1. Create array map of 256 elements
2. Load program that counts number of packets received
   r0 = skb->data[ETH_HLEN + offsetof(struct iphdr, protocol)]
   map[r0]++
3. Attach prog_fd to raw socket via setsockopt()
4. Print number of received TCP/UDP packets every second

```
int main(int argc, char **argv)
{
    int sock, map_fd, prog_fd, key;
    long long value = 0, tcp_cnt, udp_cnt;
    map_fd = bpf_create_map(BPF_MAP_TYPE_ARRAY, sizeof(key), sizeof(value), 256);
    if (map_fd < 0) { /* likely not run as root */
        printf("failed to create map '%s'\n", strerror(errno)); return 1;
    }
    struct bpf_insn prog[] = {
        BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),        /* r6 = r1 */
        BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol)), /* r0 = ip->proto */
        BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4), /* *(u32 *)(fp - 4) = r0 */
        BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),       /* r2 = fp */
        BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),      /* r2 = r2 - 4 */
        BPF_LD_MAP_FD(BPF_REG_1, map_fd),           /* r1 = map_fd */
        BPF_CALL_FUNC(BPF_FUNC_map_lookup_elem),    /* r0 = map_lookup(r1, r2) */
        BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),      /* if (r0 == 0) goto pc+2 */
        BPF_MOV64_IMM(BPF_REG_1, 1),                /* r1 = 1 */
        BPF_XADD(BPF_DW, BPF_REG_0, BPF_REG_1, 0, 0), /* lock *(u64 *) r0 += r1 */
        BPF_MOV64_IMM(BPF_REG_0, 0),                /* r0 = 0 */
        BPF_EXIT_INSN(),                            /* return r0 */
    };
    prog_fd = bpf_prog_load(BPF_PROG_TYPE_SOCKET_FILTER, prog, sizeof(prog), "GPL");
    sock = open_raw_sock("lo");
    assert(setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, &prog_fd, sizeof(prog_fd)) == 0);
    for (;;) {
        key = IPPROTO_TCP; assert(bpf_lookup_elem(map_fd, &key, &tcp_cnt) == 0);
        key = IPPROTO_UDP; assert(bpf_lookup_elem(map_fd, &key, &udp_cnt) == 0);
        printf("TCP %lld UDP %lld packets\n", tcp_cnt, udp_cnt);
        sleep(1);
    }
    return 0;
}
```

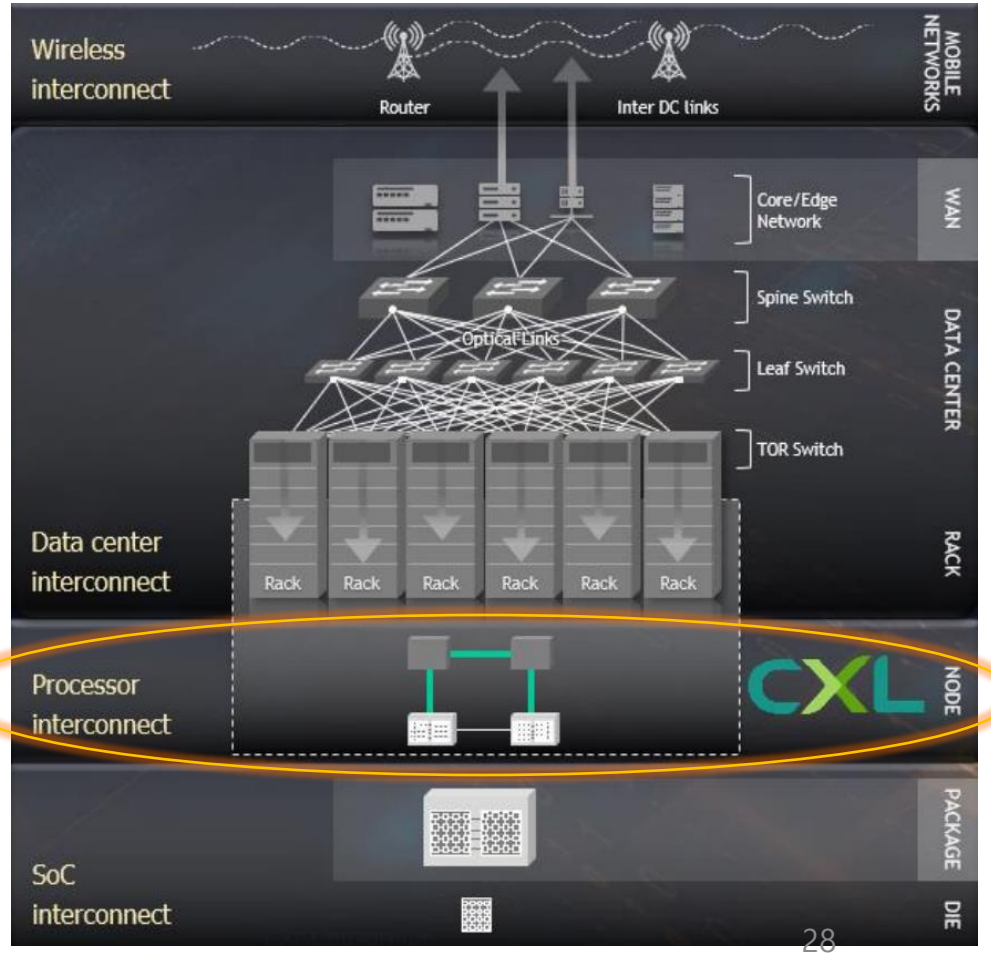
```
char bpf_log_buf[LOG_BUF_SIZE];

int
bpf_prog_load(enum bpf_prog_type type,
              const struct bpf_insn *insns, int insn_cnt,
              const char *license)
{
    union bpf_attr attr = {
        .prog_type = type,
        .insns     = ptr_to_u64(insns),
        .insn_cnt  = insn_cnt,
        .license   = ptr_to_u64(license),
        .log_buf   = ptr_to_u64(bpf_log_buf),
        .log_size  = LOG_BUF_SIZE,
        .log_level = 1,
    };

    return bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
}

prog_type is one of the available program types:

            enum bpf_prog_type {
                BPF_PROG_TYPE_UNSPEC,        /* Reserve 0 as invalid
                                                program type */
                BPF_PROG_TYPE_SOCKET_FILTER,
                BPF_PROG_TYPE_KPROBE,
                BPF_PROG_TYPE_SCHED_CLS,
                BPF_PROG_TYPE_SCHED_ACT,
            };
```

CXL (Compute Express Link)

What is CXL?

: Interconnect standard between host processors and devices

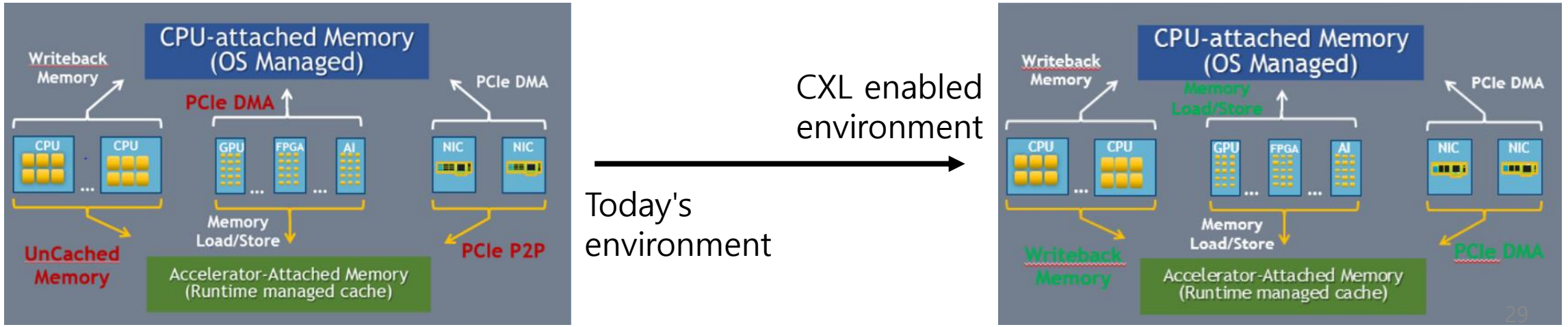
- Open industry standard for high bandwidth, low-latency interconnect between host processor and accelerators/ memory devices/ smart NICs/ and others
- Maintains memory coherency between CPU memory space and memory on attached devices
  - Allows resource sharing for higher performance
  - Reduced complexity and lower overall system cost
  - Permits users to focus on target workloads as opposed to redundant memory management
- Addresses high-performance computational workloads across AI, ML, HPC, and so on
- Based on PCIe 5.0 PHY infrastructure



Why CXL?

: For heterogeneous computing and disaggregation

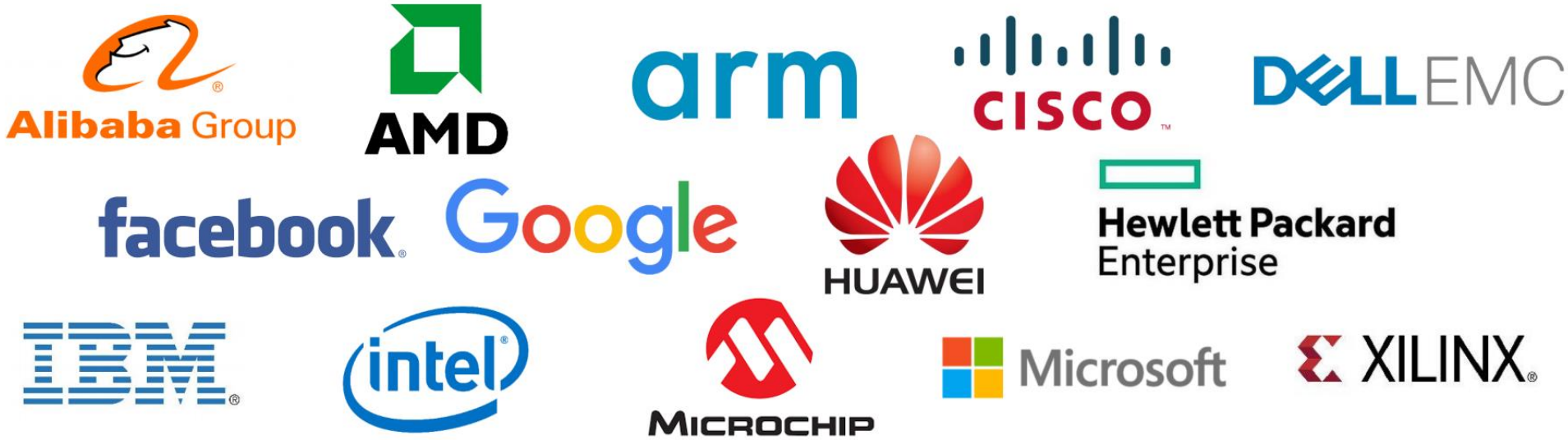
- Need a new class of interconnect for heterogeneous computing and disaggregation usages:
  - Efficient resource sharing
  - Shared memory pools with efficient access mechanisms
  - Enhanced movement of operands and results between accelerators and target devices
  - Significant latency reduction to enable disaggregated memory
- The industry needs open standards that can comprehensively address next-gen interconnect challenges



CXL Consortium Board of Directors

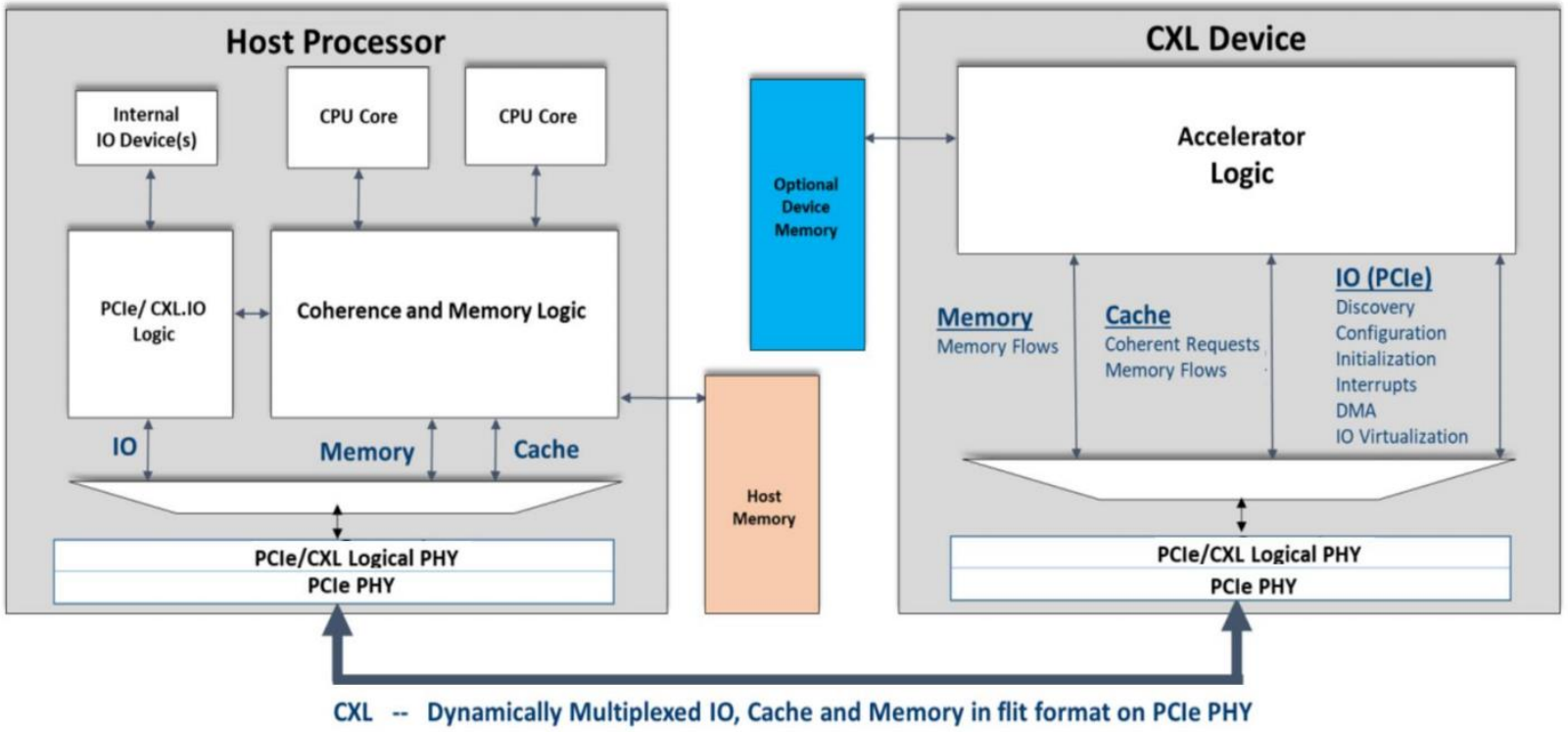
: 'Core group' announced incorporation of the CXL consortium on Sep. 17, 2019

- Alibaba, Cisco, Dell EMC, Facebook, Google, Hewlett Packard Enterprise, Huawei, Intel Corporation and Microsoft announced their intent to incorporate in March 2019
- This core group announced incorporation of the Compute Express Link (CXL) Consortium on Sep. 17, 2019 and unveiled the names of its Board of Directors



CXL Architecture

: Three multiplexed subprotocols (io, cache, mem) on a link, sharing PCIe PHY



CXL.io

- PCIe based - discovery, register access, interrupts, initialization, I/O virtualization, DMA, etc.

CXL.cache

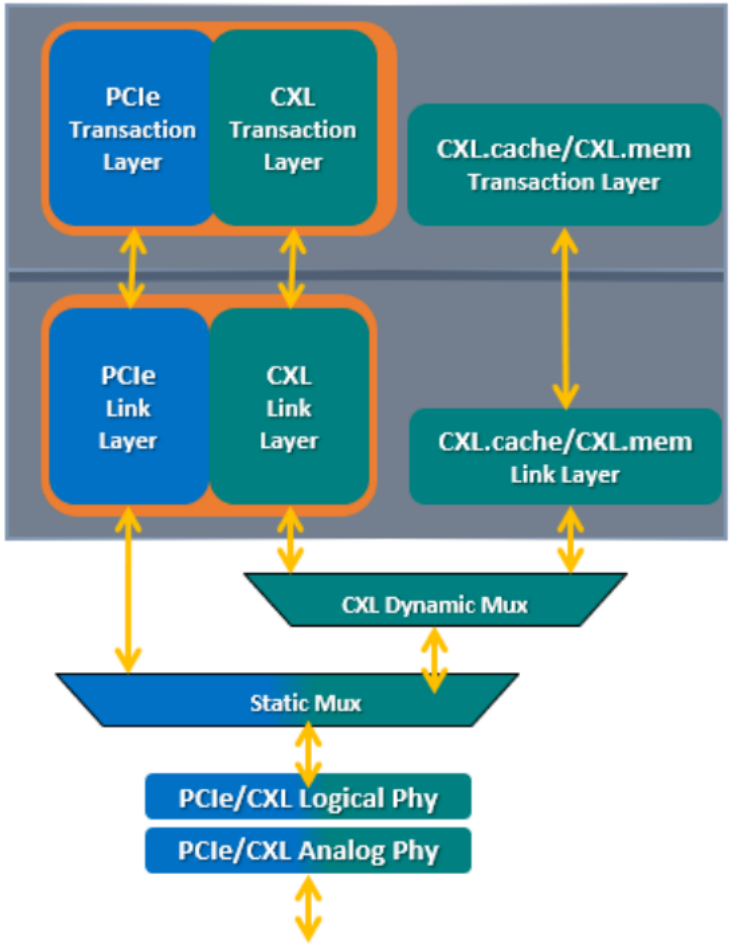
- Device access to processor memory
- Supports dievice caching of host memory with host processor orchestrating the coherency management

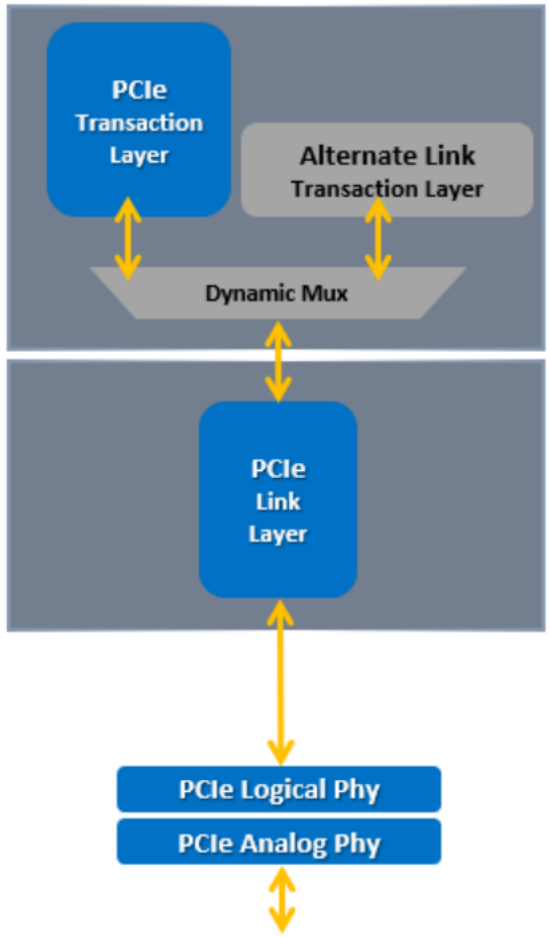
CXL.mem

- Processor access to device attached memory
- Host manages device attached memory similar to host memory

CXL Protocol Stack

: Designed for low latency





CXL.cache and CXL.mem optimized for latency

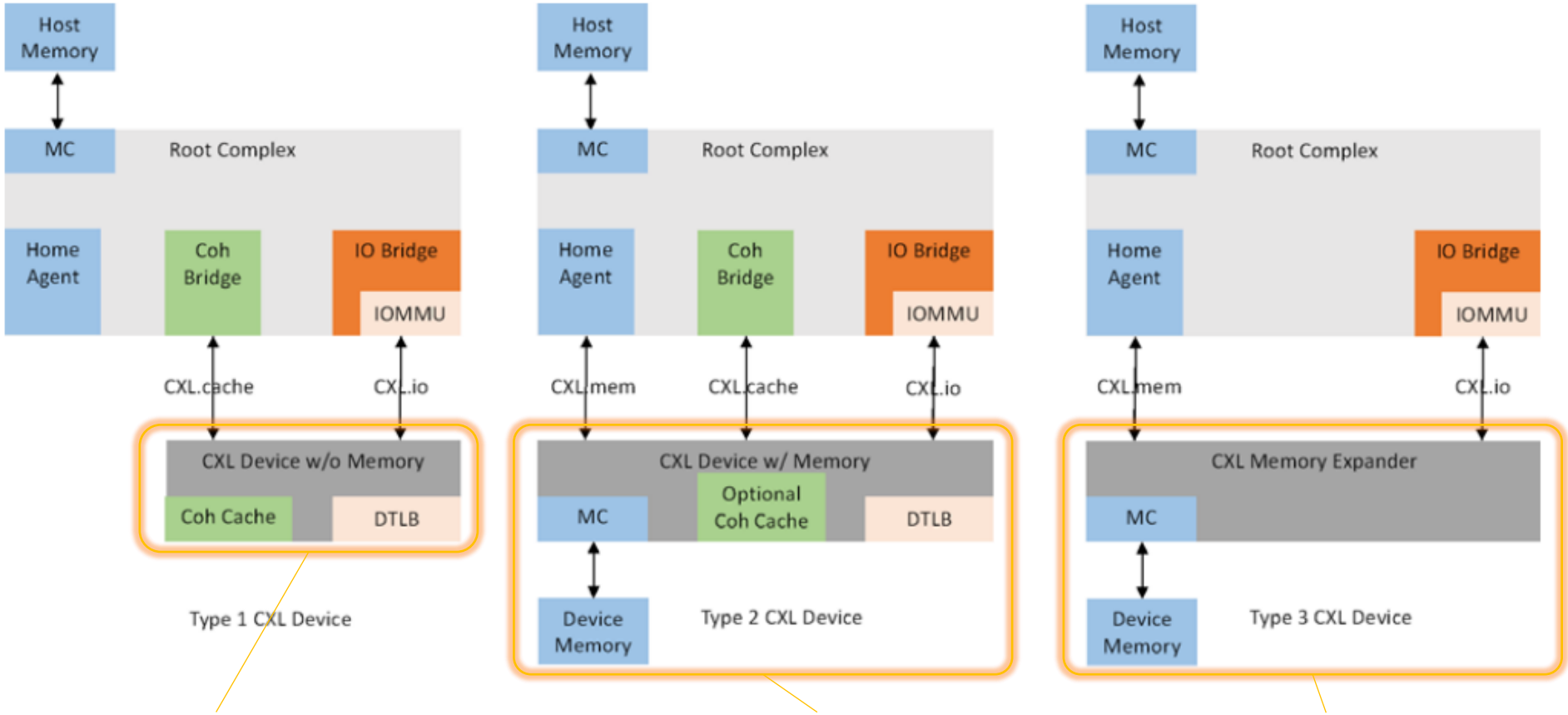
- Fixed message framing
- Separate transaction and link layer from IO

CXL.io flows pass through a stack that is largely identical a standard PCIe stack

- Dynamic framing
- Transaction Layer Packet (TLP) / Data Link Layer Packet (DLLP) encapsulated in CXL flits

CXL Use Cases

: There are three types of CXL device: Type 1, 2, 3



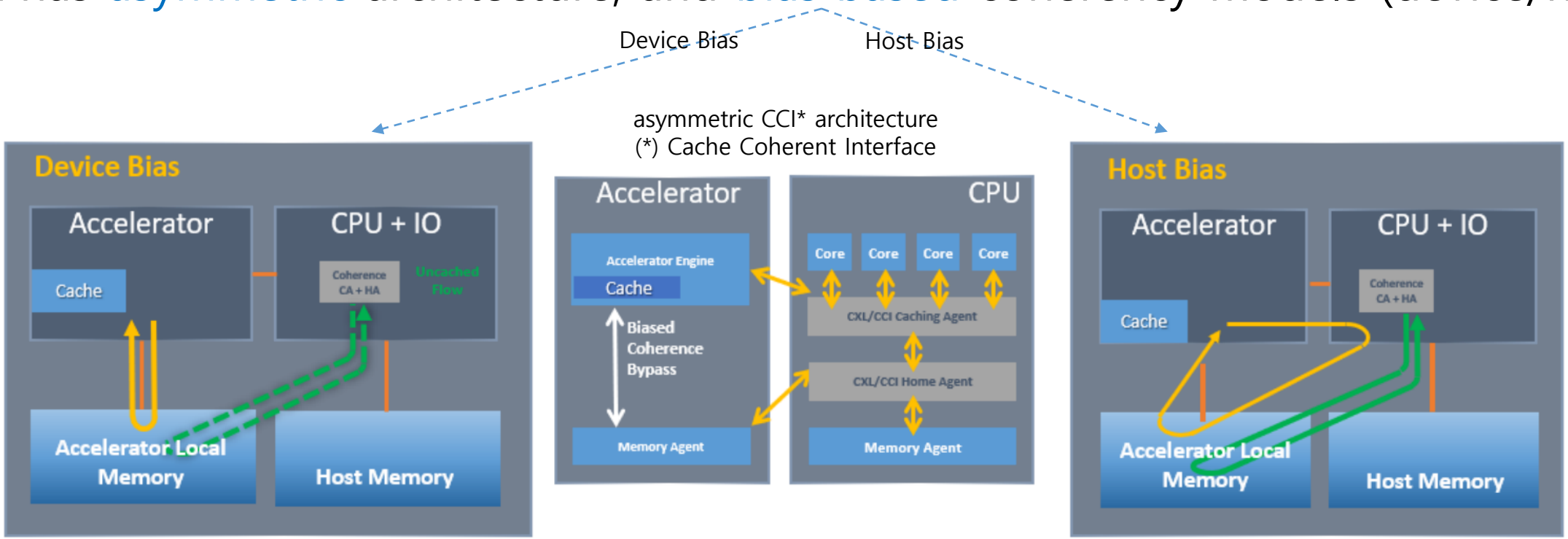
Type 1 Device:
Caching Devices / Accelerators
CXL.{io, cache}

Type 2 Device:
Accelerators with Memory
CXL.{io, cache, mem}

Type 3 CXL Device:
Memory Expanders
CXL.{io, mem}

CXL's Cache Coherence Architecture

: CXL has asymmetric architecture, and bias based coherency models (device/host)



Critical access class for accelerators is "device engine to device memory"

"Coherence Bias" allows a device engine to access its memory coherently without visiting the processor

Two driver managed modes or "Biases"

HOST BIAS: pages being used by the host or shared between host and device

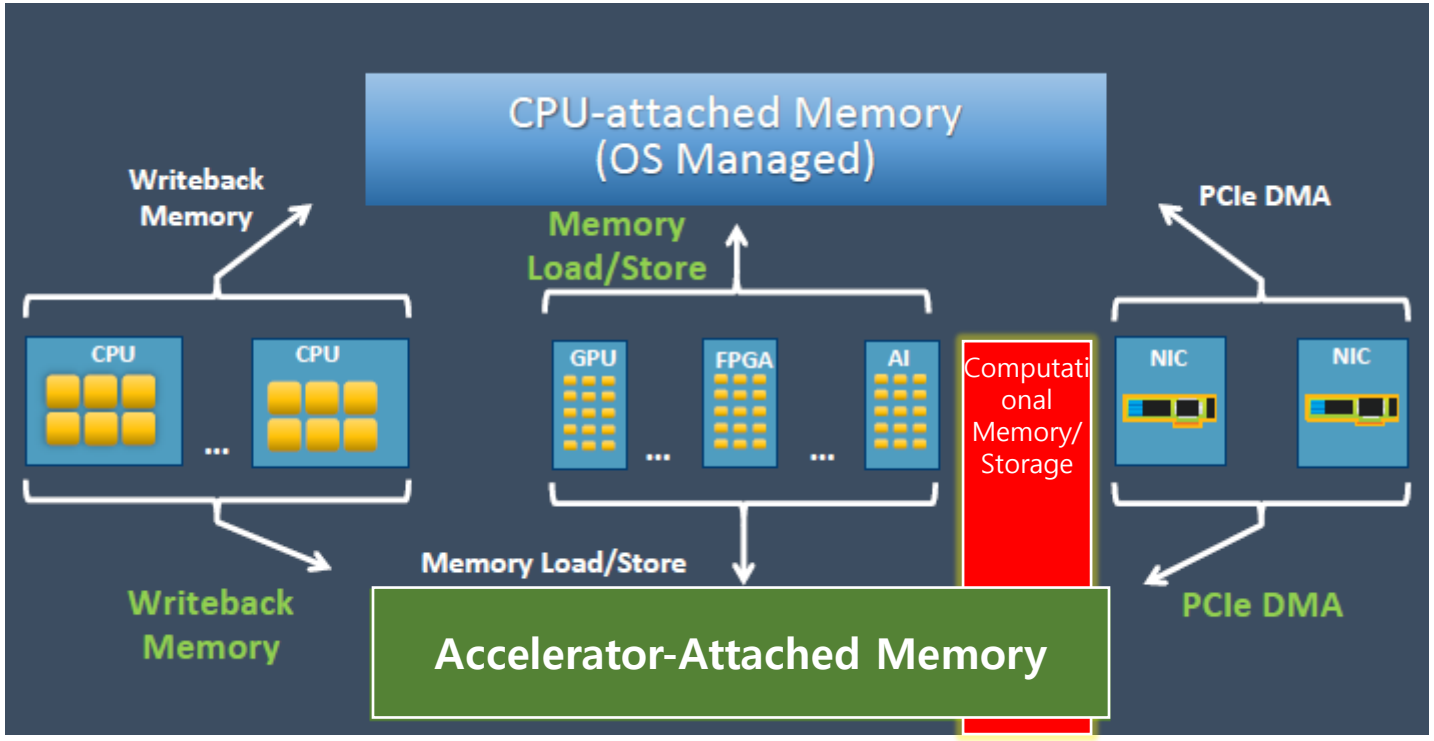
DEVICE BIAS: pages being used exclusively by the device

Both biases guaranteed correct/coherent

Guarantee applies even when software bugs or speculative accesses unexpectedly access device memory in the "Device Bias" state.

CXL Summary

: CXL will accelerate heterogeneous computing and disaggregation



- CXL has the right features and architecture to enable a broad, open ecosystem for heterogeneous computing and disaggregation
- Right abstractions with CXL.cache and CXL.mem to deliver better performance with emerging applications such as AI, HPC, DL and future compute/memory-intensive computing
- CXL 2.0 development in full progress for additional usage models
- What if we have CXL based computational memory/storage?

Wrap-up

New Standards for the **Next Generation Computational Storage Devices**

- SNIA Computational Storage (CS) Programming Model and API
  - New standard for computational storage, essential to build computational storage ecosystem
- Extended Berkeley Packet Filter (eBPF)
  - New usage of eBPF - to monitor/handle/filter device events, to execute user codes inside the device
- Compute Express Link (CXL)
  - New CPU-device interconnect standard for heterogeneous computing and disaggregation
  - Essential for computational SCM or memory pool for memory-intensive computing systems

Thank You