CXL on Linux: Boot To Bash

This documentation is intended as a companion to systems engineers (Platform, BIOS, EFI, OS, etc) trying to enable CXL devices on Linux.

CXL device configuration requires platform (Hardware, BIOS, EFI), OS (early boot, core kernel, driver), and user policy decisions that all impact each other. This doc breaks up these configurations into five main areas:

Devices and Protocols

The type of CXL device (Memory, Accelerator, etc) dictates many configuration steps. This section covers some basic background on device types and on-device resources used by the platform and OS which impact configuration.

Protocols

There are three core protocols to CXL. For the purpose of this documentation, we will only discuss very high level definitions as the specific hardware details are largely abstracted away from Linux. See the CXL specification for more details.

CXL.io

The basic interaction protocol, similar to PCIe configuration mechanisms. Typically used for initialization, configuration, and I/O access for anything other than memory (CXL.mem) or cache (CXL.cache) operations.

The Linux CXL driver exposes access to .io functionalty via the various sysfs interfaces and /dev/cxl/ devices (which exposes direct access to device mailboxes).

CXL.cache

The mechanism by which a device may coherently access and cache host memory.

Largely transparent to Linux once configured.

CXL.mem

The mechanism by which the CPU may coherently access and cache device memory.

Largely transparent to Linux once configured.

Device Types

Type-1

A Type-1 CXL device:

- Supports cxl.io and cxl.cache protocols
- Implements a fully coherent cache
- Allow Device-to-Host coherence and Host-to-Device snoops.
- Does NOT have host-managed device memory (HDM)

Typical examples of type-1 devices is a Smart NIC - which may want to directly operate on host-memory (DMA) to store incoming packets. These devices largely rely on CPU-attached memory.

Type-2

A Type-2 CXL Device:

- Supports cxl.io, cxl.cache, and cxl.mem protocols
- Optionally implements coherent cache and Host-Managed Device Memory
- Is typically an accelerator device w/ high bandwidth memory.

The primary difference between a type-1 and type-2 device is the presence of host-managed device memory, which allows the device to operate on a local memory bank - while the CPU sill has coherent DMA to the same memory.

The allows things like GPUs to expose their memory via DAX devices or file descriptors, allows drivers and programs direct access to device memory rather than use block-transfer semantics.

Type-3

A Type-3 CXL Device

- Supports cxl.io and cxl.mem
- Implements Host-Managed Device Memory
- May provide either Volatile or Persistent memory capacity (or both).

A basic example of a type-3 device is a simple memory expanded, whose local memory capacity is exposed to the CPU for access directly via basic coherent DMA.

Switch

A CXL switch is a device capacity of routing any CXL (and by extension, PCIe) protocol between an upstream, downstream, or peer devices. Many devices, such as Multi-Logical Devices, imply the presence of switching in some manner.

Logical Devices and Heads

A CXL device may present one or more “Logical Devices” to one or more hosts (via physical “Heads”).

A Single-Logical Device (SLD) is a device which presents a single device to one or more heads.

A Multi-Logical Device (MLD) is a device which may present multiple devices to one or more devices.

A Single-Headed Device exposes only a single physical connection.

A Multi-Headed Device exposes multiple physical connections.

MHSLD

A Multi-Headed Single-Logical Device (MHSLD) exposes a single logical device to multiple heads which may be connected to one or more discrete hosts. An example of this would be a simple memory-pool which may be statically configured (prior to boot) to expose portions of its memory to Linux via the CEDT ACPI table.

MHMLD

A Multi-Headed Multi-Logical Device (MHMLD) exposes multiple logical devices to multiple heads which may be connected to one or more discrete hosts. An example of this would be a Dynamic Capacity Device or which may be configured at runtime to expose portions of its memory to Linux.

Example Devices

Memory Expander

The simplest form of Type-3 device is a memory expander. A memory expander exposes Host-Managed Device Memory (HDM) to Linux. This memory may be Volatile or Non-Volatile (Persistent).

Memory Expanders will typically be considered a form of Single-Headed, Single-Logical Device - as its form factor will typically be an add-in-card (AIC) or some other similar form-factor.

The Linux CXL driver provides support for static or dynamic configuration of basic memory expanders. The platform may program decoders prior to OS init (e.g. auto-decoders), or the user may program the fabric if the platform defers these operations to the OS.

Multiple Memory Expanders may be added to an external chassis and exposed to a host via a head attached to a CXL switch. This is a “memory pool”, and would be considered an MHSLD or MHMLD depending on the management capabilities provided by the switch platform.

As of v6.14, Linux does not provide a formalized interface to manage non-DCD MHSLD or MHMLD devices.

Dynamic Capacity Device (DCD)

A Dynamic Capacity Device is a Type-3 device which provides dynamic management of memory capacity. The basic premise of a DCD to to provide an allocator-like interface for physical memory capacity to a “Fabric Manager” (an external, privileged host with privileges to change configurations for other hosts).

A DCD manages “Memory Extents”, which may be volatile or persistent. Extents may also be exclusive to a single host or shared across multiple.

As of v6.14, Linux does not provide a formalized interface to manage DCD devices, however there is active work on LKML targeting future release.

Example T2 Device

Todo

UEFI Data

Coherent Device Attribute Table (CDAT)

todo

BIOS/EFI Configuration

BIOS and EFI are largely responsible for configuring static information about devices (or potential future devices) such that Linux can build the appropriate logical representations of these devices.

At a high level, this is what occurs during this phase of configuration.

- The bootloader starts the BIOS/EFI.
- BIOS/EFI do early device probe to determine static configuration
- BIOS/EFI creates ACPI Tables that describe static config for the OS
- BIOS/EFI create the system memory map (EFI Memory Map, E820, etc)
- BIOS/EFI calls `start_kernel` and begins the Linux Early Boot process.

Much of what this section is concerned with is ACPI Table production and static memory map configuration. More detail on these tables can be found under Platform Configuration -> ACPI Table Reference.

Note

Platform Vendors should read carefully, as this sections has recommendations on physical memory region size and alignment, memory holes, HDM interleave, and what linux expects of HDM decoders trying to work with these features.

UEFI Settings

If your platform supports it, the `uefisettings` command can be used to read/write EFI settings. Changes will be reflected on the next reboot. Kexec is not a sufficient reboot.

One notable configuration here is the EFI_MEMORY_SP (Specific Purpose) bit. When this is enabled, this bit tells linux to defer management of a memory region to a driver (in this case, the CXL driver). Otherwise, the memory is treated as “normal memory”, and is exposed to the page allocator during `__init`.

uefisettings examples

`uefisettings identify`

```
uefisettings identify

bios_vendor: xxx
bios_version: xxx
bios_release: xxx
bios_date: xxx
product_name: xxx
product_family: xxx
product_version: xxx
```

On some AMD platforms, the `EFI_MEMORY_SP` bit is set via the `CXL Memory Attribute` field. This may be called something else on your platform.

`uefisettings get "CXL Memory Attribute"`

```
selector: xxx
...
question: Question {
    name: "CXL Memory Attribute",
    answer: "Enabled",
    ...
}
```

Physical Memory Map

Physical Address Region Alignment

As of Linux v6.14, the hotplug memory system requires memory regions to be uniform in size and alignment. While the CXL specification allows for memory regions as small as 256MB, the supported memory block size and alignment for hotplugged memory is architecture-defined.

A Linux memory blocks may be as small as 128MB and increase in powers of two.

- On ARM, the default block size and alignment is either 128MB or 256MB.
- On x86, the default block size is 256MB, and increases to 2GB as the capacity of the system increases up to 64GB.

For best support across versions, platform vendors should place CXL memory at a 2GB aligned base address, and regions should be 2GB aligned. This also helps prevent the creating thousands of memory devices (one per block).

Memory Holes

Holes in the memory map are tricky. Consider a 4GB device located at base address 0x100000000, but with the following following memory map

```
---------------------
|    0x100000000    |
|        CXL        |
|    0x1BFFFFFFF    |
---------------------
|    0x1C0000000    |
|    MEMORY HOLE    |
|    0x1FFFFFFFF    |
---------------------
|    0x200000000    |
|     CXL CONT.     |
|    0x23FFFFFFF    |
---------------------
```

There are two issues to consider:

- decoder programming, and
- memory block alignment.

If your architecture requires 2GB uniform size and aligned memory blocks, the only capacity Linux is capable of mapping (as of v6.14) would be the capacity from *0x100000000-0x180000000*. The remaining capacity will be stranded, as they are not of 2GB aligned length.

Assuming your architecture and memory configuration allows 1GB memory blocks, this memory map is supported and this should be presented as multiple CFMWS in the CEDT that describe each side of the memory hole separately - along with matching decoders.

Multiple decoders can (and should) be used to manage such a memory hole (see below), but each chunk of a memory hole should be aligned to a reasonable block size (larger alignment is always better). If you intend to have memory holes in the memory map, expect to use one decoder per contiguous chunk of host physical memory.

As of v6.14, Linux does provide support for memory hotplug of multiple physical memory regions separated by a memory hole described by a single HDM decoder.

Decoder Programming

If BIOS/EFI intends to program the decoders to be statically configured, there are a few things to consider to avoid major pitfalls that will prevent Linux compatibility. Some of these recommendations are not not required "per the specification", but Linux makes no guarantees of support otherwise.

Translation Point

Per the specification, the only decoders which **TRANSLATE** Host Physical Address (HPA) to Device Physical Address (DPA) are the **Endpoint Decoders**. All other decoders in the fabric are intended to route accesses without translating the addresses.

This is heavily implied by the specification, see:

```
CXL Specification 3.1
8.2.4.20: CXL HDM Decoder Capability Structure
- Implementation Note: CXL Host Bridge and Upstream Switch Port Decoder Flow
- Implementation Note: Device Decoder Logic
```

Given this, Linux makes a strong assumption that decoders between CPU and endpoint will all be programmed with addresses ranges that are subsets of their parent decoder.

Due to some ambiguity in how Architecture, ACPI, PCI, and CXL specifications "hand off" responsibility between domains, some early adopting platforms attempted to do translation at the originating memory controller or host bridge. This configuration requires a platform specific extension to the driver and is not officially endorsed - despite being supported.

It is *highly recommended* **NOT** to do this; otherwise, you are on your own to implement driver support for your platform.

Interleave and Configuration Flexibility

If providing cross-host-bridge interleave, a CFMWS entry in the CEDT must be presented with target host-bridges for the interleaved device sets (there may be multiple behind each host bridge).

If providing intra-host-bridge interleaving, only 1 CFMWS entry in the CEDT is required for that host bridge - if it covers the entire capacity of the devices behind the host bridge.

If intending to provide users flexibility in programming decoders beyond the root, you may want to provide multiple CFMWS entries in the CEDT intended for different purposes. For example, you may want to consider adding:

1. A CFMWS entry to cover all interleavable host bridges.
2. A CFMWS entry to cover all devices on a single host bridge.
3. A CFMWS entry to cover each device.

A platform may choose to add all of these, or change the mode based on a BIOS setting. For each CFMWS entry, Linux expects descriptions of the described memory regions in the SRAT to determine the number of NUMA nodes it should reserve during early boot / init.

As of v6.14, Linux will create a NUMA node for each CEDT CFMWS entry, even if a matching SRAT entry does not exist; however, this is not guaranteed in the future and such a configuration should be avoided.

Memory Holes

If your platform includes memory holes intersparsed between your CXL memory, it is recommended to utilize multiple decoders to cover these regions of memory, rather than try to program the decoders to accept the entire range and expect Linux to manage the overlap.

For example, consider the Memory Hole described above

```
---------------------
|    0x100000000    |
|        CXL        |
|    0x1BFFFFFFF    |
---------------------
|    0x1C0000000    |
|    MEMORY HOLE    |
|    0x1FFFFFFFF    |
---------------------
|    0x200000000    |
|     CXL CONT.     |
|    0x23FFFFFFF    |
---------------------
```

Assuming this is provided by a single device attached directly to a host bridge, Linux would expect the following decoder programming

```
-----------------------   -----------------------
| root-decoder-0      |   | root-decoder-1      |
|   base: 0x100000000 |   |   base: 0x200000000 |
|   size:  0xC0000000 |   |   size:  0x40000000 |
-----------------------   -----------------------
           |                          |
-----------------------   -----------------------
| HB-decoder-0        |   | HB-decoder-1        |
|   base: 0x100000000 |   |   base: 0x200000000 |
|   size:  0xC0000000 |   |   size:  0x40000000 |
-----------------------   -----------------------
           |                          |
-----------------------   -----------------------
| ep-decoder-0        |   | ep-decoder-1        |
|   base: 0x100000000 |   |   base: 0x200000000 |
|   size:  0xC0000000 |   |   size:  0x40000000 |
-----------------------   -----------------------
```

With a CEDT configuration with two CFMWS describing the above root decoders.

Linux makes no guarantee of support for strange memory hole situations.

Multi-Media Devices

The CFMWS field of the CEDT has special restriction bits which describe whether the described memory region allows volatile or persistent memory (or both). If the platform intends to support either:

1. A device with multiple medias, or
2. Using a persistent memory device as normal memory

A platform may wish to create multiple CEDT CFMWS entries to describe the same memory, with the intent of allowing the end user flexibility in how that memory is configured. Linux does not presently have strong requirements in this area.

ACPI Tables

ACPI is the “Advanced Configuration and Power Interface”, which is a standard that defines how platforms and OS manage pwoer and configure computer hardware. For the purpose of this theory of operation, when referring to “ACPI” we will usually refer to “ACPI Tables” - which are the way a platform (BIOS/EFI) communicates static configuration information to the operation system.

The Following ACPI tables contain *static* configuration and performance data about CXL devices.

CEDT - CXL Early Discovery Table

The CXL Early Discovery Table is generated by BIOS to describe the CXL memory regions configured at boot by the BIOS.

CHBS

The CXL Host Bridge Structure describes CXL host bridges. Other than describing device register information, it reports the specific host bridge UID for this host bridge. These host bridge ID's will be referenced in other tables.

Example

```
          Subtable Type : 00 [CXL Host Bridge Structure]
               Reserved : 00
                 Length : 0020
Associated host bridge : 00000007    <- Host bridge _UID
 Specification version : 00000001
               Reserved : 00000000
          Register base : 0000010370400000
        Register length : 0000000000010000
```

CFMWS

The CXL Fixed Memory Window structure describes a memory region associated with one or more CXL host bridges (as described by the CHBS). It additionally describes any inter-host-bridge interleave configuration that may have been programmed by BIOS.

Example

```
            Subtable Type : 01 [CXL Fixed Memory Window Structure]
                 Reserved : 00
                   Length : 002C
                 Reserved : 00000000
      Window base address : 000000C050000000   <- Memory Region
              Window size : 0000003CA0000000
Interleave Members (2^n) : 01                 <- Interleave configuration
   Interleave Arithmetic : 00
                 Reserved : 0000
              Granularity : 00000000
             Restrictions : 0006
                    QtgId : 0001
             First Target : 00000007           <- Host Bridge _UID
              Next Target : 00000006           <- Host Bridge _UID
```

The restriction field dictates what this SPA range may be used for (memory type, voltile vs persistent, etc). One or more bits may be set.

```
Bit[0]: CXL Type 2 Memory
Bit[1]: CXL Type 3 Memory
Bit[2]: Volatile Memory
Bit[3]: Persistent Memory
Bit[4]: Fixed Config (HPA cannot be re-used)
```

INTRA-host-bridge interleave (multiple devices on one host bridge) is NOT reported in this structure, and is solely defined via CXL device decoder programming (host bridge and endpoint decoders).

SRAT - Static Resource Affinity Table

The System/Static Resource Affinity Table describes resource (CPU, Memory) affinity to "Proximity Domains". This table is technically optional, but for performance information (see "HMAT") to be enumerated by linux it must be present.

There is a careful dance between the CEDT and SRAT tables and how NUMA nodes are created. If things don't look quite the way you expect - check the SRAT Memory Affinity entries and CEDT CFMWS to determine what your platform actually supports in terms of flexible topologies.

The SRAT may statically assign portions of a CFMWS SPA range to a specific proximity domains. See linux numa creation for more information about how this presents in the NUMA topology.

Proximity Domain

A proximity domain is ROUGHLY equivalent to "NUMA Node" - though a 1-to-1 mapping is not guaranteed. There are scenarios where "Proximity Domain 4" may map to "NUMA Node 3", for example. (See "NUMA Node Creation")

Memory Affinity

Generally speaking, if a host does any amount of CXL fabric (decoder) programming in BIOS - an SRAT entry for that memory needs to be present.

Example

```
        Subtable Type : 01 [Memory Affinity]
               Length : 28
     Proximity Domain : 00000001          <- NUMA Node 1
            Reserved1 : 0000
         Base Address : 000000C050000000  <- Physical Memory Region
       Address Length : 0000003CA0000000
            Reserved2 : 00000000
Flags (decoded below) : 0000000B
             Enabled : 1
       Hot Pluggable : 1
        Non-Volatile : 0
```

Generic Initiator / Port

todo

HMAT - Heterogeneous Memory Attribute Table

The Heterogeneous Memory Attributes Table contains information such as cache attributes and bandwidth and latency details for memory proximity domains. For the purpose of this document, we will only discuss the SSLIB entry.

SLLBI

The System Locality Latency and Bandwidth Information records latency and bandwidth information for proximity domains.

This table is used by Linux to configure interleave weights and memory tiers.

Example (Heavily truncated for brevity)

```
              Structure Type : 0001 [SLLBI]
                   Data Type : 00         <- Latency
Target Proximity Domain List : 00000000
Target Proximity Domain List : 00000001
                       Entry : 0080       <- DRAM LTC
                       Entry : 0100       <- CXL LTC

              Structure Type : 0001 [SLLBI]
                   Data Type : 03         <- Bandwidth
Target Proximity Domain List : 00000000
Target Proximity Domain List : 00000001
                       Entry : 1200       <- DRAM BW
                       Entry : 0200       <- CXL BW
```

SLIT - System Locality Information Table

The system locality information table provides “abstract distances” between accessor and memory nodes. Node without initiators (cpus) are infinitely (FF) distance away from all other nodes.

The abstract distance described in this table does not describe any real latency of bandwidth information.

Example

```
   Signature : "SLIT"    [System Locality Information Table]
  Localities : 0000000000000004
Locality   0 : 10 20 20 30
Locality   1 : 20 10 30 20
Locality   2 : FF FF 0A FF
Locality   3 : FF FF FF 0A
```

DSDT - Differentiated system Description Table

This table describes what peripherals a machine has.

This table's UIDs for CXL devices - specifically host bridges, must be consistent with the contents of the CEDT, otherwise the CXL driver will fail to probe correctly.

Example Compute Express Link Host Bridge

```
Scope (_SB)
{
    Device (S0D0)
    {
        Name (_HID, "ACPI0016" /* Compute Express Link Host Bridge */)  // _HID:
Hardware ID
        Name (_CID, Package (0x02)  // _CID: Compatible ID
        {
            EisaId ("PNP0A08") /* PCI Express Bus */,
            EisaId ("PNP0A03") /* PCI Bus */
        })
        ...
        Name (_UID, 0x05)  // _UID: Unique ID
        ...
  }
```

The SRAT table may also contain generic port/initiator content that is intended to describe the generic port, but not information about the rest of the path to the endpoint.

Linux uses these tables to configure kernel resources for statically configured (by BIOS/EFI) CXL devices, such as:

- NUMA nodes
- Memory Tiers
- NUMA Abstract Distances
- SystemRAM Memory Regions
- Weighted Interleave Node Weights

ACPI Debugging

The `acpidump -b` command dumps the ACPI tables into binary format.

The `iasl -d` command disassembles the files into human readable format.

Example `acpidump -b && iasl -d cedt.dat`

```
/*
 * Intel ACPI Component Architecture
 * AML/ASL+ Disassembler version 20210604 (64-bit version)
 * Copyright (c) 2000 - 2021 Intel Corporation
 *
 * Disassembly of cedt.dat, Fri Apr 11 07:47:31 2025
 *
 * ACPI Data Table [CEDT]
 *
 * Format: [HexOffset DecimalOffset ByteLength]  FieldName : FieldValue
 */
 [000h 0000   4]   Signature : "CEDT"    [CXL Early Discovery Table]
 ...
```

Common Issues

Most failures described here result in a failure of the driver to surface memory as a DAX device and/or kmem.

- CEDT CFMWS targets list UIDs do not match CEDT CHBS UIDs.
- CEDT CFMWS targets list UIDs do not match DSDT CXL Host Bridge UIDs.
- CEDT CFMWS Restriction Bits are not correct.
- CEDT CFMWS Memory regions are poorly aligned.
- CEDT CFMWS Memory regions spans a platform memory hole.
- CEDT CHBS UIDs do not match DSDT CXL Host Bridge UIDs.
- CEDT CHBS Specification version is incorrect.
- SRAT is missing regions described in CEDT CFMWS.
  - Result: failure to create a NUMA node for the region, or region is placed in wrong node.
- HMAT is missing data for regions described in CEDT CFMWS.
  - Result: NUMA node being placed in the wrong memory tier.
- SLIT has bad data.
  - Result: Lots of performance mechanisms in the kernel will be very unhappy.

All of these issues will appear to users as if the driver is failing to support CXL - when in reality they are all the failure of a platform to configure the ACPI tables correctly.

Example Platform Configurations

One Device per Host Bridge

This system has a single socket with two CXL host bridges. Each host bridge has a single CXL memory expander with a 4GB of memory.

Things to note:

- Cross-Bridge interleave is not being used.

- The expanders are in two separate but adjascent memory regions.
- This CEDT/SRAT describes one-node per device
- The expanders have the same performance and will be in the same memory tier.

CEDT

```
           Subtable Type : 00 [CXL Host Bridge Structure]
                Reserved : 00
                  Length : 0020
   Associated host bridge : 00000007
   Specification version : 00000001
                Reserved : 00000000
           Register base : 0000010370400000
         Register length : 0000000000010000

           Subtable Type : 00 [CXL Host Bridge Structure]
                Reserved : 00
                  Length : 0020
   Associated host bridge : 00000006
   Specification version : 00000001
                Reserved : 00000000
           Register base : 0000010380800000
         Register length : 0000000000010000

           Subtable Type : 01 [CXL Fixed Memory Window Structure]
                Reserved : 00
                  Length : 002C
                Reserved : 00000000
     Window base address : 0000001000000000
             Window size : 0000000100000000
Interleave Members (2^n) : 00
   Interleave Arithmetic : 00
                Reserved : 0000
             Granularity : 00000000
            Restrictions : 0006
                   QtgId : 0001
            First Target : 00000007

           Subtable Type : 01 [CXL Fixed Memory Window Structure]
                Reserved : 00
                  Length : 002C
                Reserved : 00000000
     Window base address : 0000001100000000
             Window size : 0000000100000000
Interleave Members (2^n) : 00
   Interleave Arithmetic : 00
                Reserved : 0000
             Granularity : 00000000
            Restrictions : 0006
                   QtgId : 0001
            First Target : 00000006
```

SRAT

```
        Subtable Type : 01 [Memory Affinity]
               Length : 28
     Proximity Domain : 00000001
            Reserved1 : 0000
         Base Address : 0000001000000000
       Address Length : 0000000100000000
            Reserved2 : 00000000
Flags (decoded below) : 0000000B
            Enabled : 1
      Hot Pluggable : 1
       Non-Volatile : 0

        Subtable Type : 01 [Memory Affinity]
               Length : 28
     Proximity Domain : 00000002
            Reserved1 : 0000
         Base Address : 0000001100000000
       Address Length : 0000000100000000
            Reserved2 : 00000000
Flags (decoded below) : 0000000B
            Enabled : 1
      Hot Pluggable : 1
       Non-Volatile : 0
```

HMAT

```
              Structure Type : 0001 [SLLBI]
                   Data Type : 00   [Latency]
Target Proximity Domain List : 00000000
Target Proximity Domain List : 00000001
Target Proximity Domain List : 00000002
                       Entry : 0080
                       Entry : 0100
                       Entry : 0100

              Structure Type : 0001 [SLLBI]
                   Data Type : 03   [Bandwidth]
Target Proximity Domain List : 00000000
Target Proximity Domain List : 00000001
Target Proximity Domain List : 00000002
                       Entry : 1200
                       Entry : 0200
                       Entry : 0200
```

SLIT

```
   Signature : "SLIT"    [System Locality Information Table]
  Localities : 0000000000000003
Locality   0 : 10 20 20
Locality   1 : FF 0A FF
Locality   2 : FF FF 0A
```

DSDT

```
Scope (_SB)
{
  Device (S0D0)
  {
      Name (_HID, "ACPI0016" /* Compute Express Link Host Bridge */)  // _HID: Hardware
ID
      ...
      Name (_UID, 0x07)  // _UID: Unique ID
  }
  ...
  Device (S0D5)
  {
      Name (_HID, "ACPI0016" /* Compute Express Link Host Bridge */)  // _HID: Hardware
ID
      ...
      Name (_UID, 0x06)  // _UID: Unique ID
  }
}
```

Multiple Devices per Host Bridge

In this example system we will have a single socket and one CXL host bridge. There are two CXL memory expanders with 4GB attached to the host bridge.

Things to note:

- Intra-Bridge interleave is not described here.
- The expanders are described by a single CEDT/CFMWS.
- This CEDT/SRAT describes one node for both devices.
- There is only one proximity domain the HMAT for both devices.

CEDT

```
           Subtable Type : 00 [CXL Host Bridge Structure]
                Reserved : 00
                  Length : 0020
  Associated host bridge : 00000007
   Specification version : 00000001
                Reserved : 00000000
           Register base : 0000010370400000
         Register length : 0000000000010000

           Subtable Type : 01 [CXL Fixed Memory Window Structure]
                Reserved : 00
                  Length : 002C
                Reserved : 00000000
     Window base address : 0000001000000000
             Window size : 0000000200000000
Interleave Members (2^n) : 00
   Interleave Arithmetic : 00
                Reserved : 0000
             Granularity : 00000000
            Restrictions : 0006
                   QtgId : 0001
            First Target : 00000007
```

SRAT

```
          Subtable Type : 01 [Memory Affinity]
                 Length : 28
       Proximity Domain : 00000001
              Reserved1 : 0000
           Base Address : 0000001000000000
         Address Length : 0000000200000000
              Reserved2 : 00000000
Flags (decoded below) : 0000000B
              Enabled : 1
        Hot Pluggable : 1
         Non-Volatile : 0
```

HMAT

```
              Structure Type : 0001 [SLLBI]
                   Data Type : 00   [Latency]
Target Proximity Domain List : 00000000
Target Proximity Domain List : 00000001
                       Entry : 0080
                       Entry : 0100

              Structure Type : 0001 [SLLBI]
                   Data Type : 03   [Bandwidth]
Target Proximity Domain List : 00000000
Target Proximity Domain List : 00000001
                       Entry : 1200
                       Entry : 0200
```

SLIT

```
   Signature : "SLIT"    [System Locality Information Table]
  Localities : 0000000000000003
Locality    0 : 10 20
Locality    1 : FF 0A
```

DSDT

```
Scope (_SB)
{
  Device (S0D0)
  {
      Name (_HID, "ACPI0016" /* Compute Express Link Host Bridge */)  // _HID: Hardware
ID
      ...
      Name (_UID, 0x07)  // _UID: Unique ID
  }
  ...
}
```

Cross-Host-Bridge Interleave

This system has a single socket with two CXL host bridges. Each host bridge has a single CXL memory expander with a 4GB of memory.

Things to note:

- Cross-Bridge interleave is described.
- The expanders are described by a single CFMWS.
- This SRAT describes one-node for both host bridges.
- The HMAT describes a single node's performance.

CEDT

```
           Subtable Type : 00 [CXL Host Bridge Structure]
                Reserved : 00
                  Length : 0020
   Associated host bridge : 00000007
    Specification version : 00000001
                Reserved : 00000000
           Register base : 0000010370400000
         Register length : 0000000000010000

           Subtable Type : 00 [CXL Host Bridge Structure]
                Reserved : 00
                  Length : 0020
   Associated host bridge : 00000006
    Specification version : 00000001
                Reserved : 00000000
           Register base : 0000010380800000
         Register length : 0000000000010000

           Subtable Type : 01 [CXL Fixed Memory Window Structure]
                Reserved : 00
                  Length : 002C
                Reserved : 00000000
      Window base address : 0000001000000000
             Window size : 0000000200000000
 Interleave Members (2^n) : 01
    Interleave Arithmetic : 00
                Reserved : 0000
             Granularity : 00000000
            Restrictions : 0006
                   QtgId : 0001
            First Target : 00000007
           Second Target : 00000006
```

SRAT

```
        Subtable Type : 01 [Memory Affinity]
               Length : 28
     Proximity Domain : 00000001
            Reserved1 : 0000
         Base Address : 0000001000000000
       Address Length : 0000000200000000
            Reserved2 : 00000000
Flags (decoded below) : 0000000B
            Enabled : 1
      Hot Pluggable : 1
       Non-Volatile : 0
```

HMAT

```
              Structure Type : 0001 [SLLBI]
                   Data Type : 00   [Latency]
Target Proximity Domain List : 00000000
Target Proximity Domain List : 00000001
Target Proximity Domain List : 00000002
                       Entry : 0080
                       Entry : 0100

              Structure Type : 0001 [SLLBI]
                   Data Type : 03   [Bandwidth]
Target Proximity Domain List : 00000000
Target Proximity Domain List : 00000001
Target Proximity Domain List : 00000002
                       Entry : 1200
                       Entry : 0400
```

SLIT

```
    Signature : "SLIT"    [System Locality Information Table]
   Localities : 0000000000000003
Locality   0 : 10 20
Locality   1 : FF 0A
```

DSDT

```
Scope (_SB)
{
  Device (S0D0)
  {
      Name (_HID, "ACPI0016" /* Compute Express Link Host Bridge */)  // _HID: Hardware
ID
      ...
      Name (_UID, 0x07)  // _UID: Unique ID
  }
  ...
  Device (S0D5)
  {
      Name (_HID, "ACPI0016" /* Compute Express Link Host Bridge */)  // _HID: Hardware
ID
      ...
      Name (_UID, 0x06)  // _UID: Unique ID
  }
}
```

Flexible Presentation

This system has a single socket with two CXL host bridges. Each host bridge has two CXL memory expanders with a 4GB of memory (32GB total).

On this system, the platform designer wanted to provide the user flexibility to configure the memory devices in various interleave or NUMA node configurations. So they provided every combination.

Things to note:

- Cross-Bridge interleave is described in one CFMWS that covers all capacity.
- One CFMWS is also described per-host bridge.
- One CFMWS is also described per-device.
- This SRAT describes one-node for each of the above CFMWS.
- The HMAT describes performance for each node in the SRAT.

CEDT

```
          Subtable Type : 00 [CXL Host Bridge Structure]
               Reserved : 00
                 Length : 0020
  Associated host bridge : 00000007
   Specification version : 00000001
               Reserved : 00000000
          Register base : 0000010370400000
        Register length : 0000000000010000

          Subtable Type : 00 [CXL Host Bridge Structure]
               Reserved : 00
                 Length : 0020
  Associated host bridge : 00000006
   Specification version : 00000001
               Reserved : 00000000
          Register base : 0000010380800000
        Register length : 0000000000010000

          Subtable Type : 01 [CXL Fixed Memory Window Structure]
               Reserved : 00
                 Length : 002C
               Reserved : 00000000
     Window base address : 0000001000000000
            Window size : 0000000400000000
Interleave Members (2^n) : 01
   Interleave Arithmetic : 00
               Reserved : 0000
            Granularity : 00000000
           Restrictions : 0006
                  QtgId : 0001
           First Target : 00000007
          Second Target : 00000006

          Subtable Type : 01 [CXL Fixed Memory Window Structure]
               Reserved : 00
                 Length : 002C
               Reserved : 00000000
     Window base address : 0000002000000000
            Window size : 0000000200000000
Interleave Members (2^n) : 00
   Interleave Arithmetic : 00
               Reserved : 0000
            Granularity : 00000000
           Restrictions : 0006
                  QtgId : 0001
           First Target : 00000007

          Subtable Type : 01 [CXL Fixed Memory Window Structure]
               Reserved : 00
                 Length : 002C
               Reserved : 00000000
     Window base address : 0000002200000000
            Window size : 0000000200000000
Interleave Members (2^n) : 00
   Interleave Arithmetic : 00
               Reserved : 0000
            Granularity : 00000000
           Restrictions : 0006
                  QtgId : 0001
           First Target : 00000006

          Subtable Type : 01 [CXL Fixed Memory Window Structure]
               Reserved : 00
                 Length : 002C
```

```
                Reserved : 00000000
     Window base address : 0000003000000000
             Window size : 0000000100000000
Interleave Members (2^n) : 00
   Interleave Arithmetic : 00
                Reserved : 0000
             Granularity : 00000000
            Restrictions : 0006
                   QtgId : 0001
            First Target : 00000007

           Subtable Type : 01 [CXL Fixed Memory Window Structure]
                Reserved : 00
                  Length : 002C
                Reserved : 00000000
     Window base address : 0000003100000000
             Window size : 0000000100000000
Interleave Members (2^n) : 00
   Interleave Arithmetic : 00
                Reserved : 0000
             Granularity : 00000000
            Restrictions : 0006
                   QtgId : 0001
            First Target : 00000007

           Subtable Type : 01 [CXL Fixed Memory Window Structure]
                Reserved : 00
                  Length : 002C
                Reserved : 00000000
     Window base address : 0000003200000000
             Window size : 0000000100000000
Interleave Members (2^n) : 00
   Interleave Arithmetic : 00
                Reserved : 0000
             Granularity : 00000000
            Restrictions : 0006
                   QtgId : 0001
            First Target : 00000006

           Subtable Type : 01 [CXL Fixed Memory Window Structure]
                Reserved : 00
                  Length : 002C
                Reserved : 00000000
     Window base address : 0000003300000000
             Window size : 0000000100000000
Interleave Members (2^n) : 00
   Interleave Arithmetic : 00
                Reserved : 0000
             Granularity : 00000000
            Restrictions : 0006
                   QtgId : 0001
            First Target : 00000006
```

SRAT

```
        Subtable Type : 01 [Memory Affinity]
               Length : 28
     Proximity Domain : 00000001
            Reserved1 : 0000
         Base Address : 0000001000000000
       Address Length : 0000000400000000
            Reserved2 : 00000000
Flags (decoded below) : 0000000B
            Enabled : 1
      Hot Pluggable : 1
       Non-Volatile : 0

        Subtable Type : 01 [Memory Affinity]
               Length : 28
     Proximity Domain : 00000002
            Reserved1 : 0000
         Base Address : 0000002000000000
       Address Length : 0000000200000000
            Reserved2 : 00000000
Flags (decoded below) : 0000000B
            Enabled : 1
      Hot Pluggable : 1
       Non-Volatile : 0

        Subtable Type : 01 [Memory Affinity]
               Length : 28
     Proximity Domain : 00000003
            Reserved1 : 0000
         Base Address : 0000002200000000
       Address Length : 0000000200000000
            Reserved2 : 00000000
Flags (decoded below) : 0000000B
            Enabled : 1
      Hot Pluggable : 1
       Non-Volatile : 0

        Subtable Type : 01 [Memory Affinity]
               Length : 28
     Proximity Domain : 00000004
            Reserved1 : 0000
         Base Address : 0000003000000000
       Address Length : 0000000100000000
            Reserved2 : 00000000
Flags (decoded below) : 0000000B
            Enabled : 1
      Hot Pluggable : 1
       Non-Volatile : 0

        Subtable Type : 01 [Memory Affinity]
               Length : 28
     Proximity Domain : 00000005
            Reserved1 : 0000
         Base Address : 0000003100000000
       Address Length : 0000000100000000
            Reserved2 : 00000000
Flags (decoded below) : 0000000B
            Enabled : 1
      Hot Pluggable : 1
       Non-Volatile : 0

        Subtable Type : 01 [Memory Affinity]
               Length : 28
     Proximity Domain : 00000006
            Reserved1 : 0000
```

```
          Base Address : 0000003200000000
        Address Length : 0000000100000000
             Reserved2 : 00000000
 Flags (decoded below) : 0000000B
             Enabled : 1
       Hot Pluggable : 1
        Non-Volatile : 0

         Subtable Type : 01 [Memory Affinity]
                Length : 28
      Proximity Domain : 00000007
             Reserved1 : 0000
          Base Address : 0000003300000000
        Address Length : 0000000100000000
             Reserved2 : 00000000
 Flags (decoded below) : 0000000B
             Enabled : 1
       Hot Pluggable : 1
        Non-Volatile : 0
```

HMAT

```
              Structure Type : 0001 [SLLBI]
                   Data Type : 00   [Latency]
Target Proximity Domain List : 00000000
Target Proximity Domain List : 00000001
Target Proximity Domain List : 00000002
Target Proximity Domain List : 00000003
Target Proximity Domain List : 00000004
Target Proximity Domain List : 00000005
Target Proximity Domain List : 00000006
Target Proximity Domain List : 00000007
                       Entry : 0080
                       Entry : 0100
                       Entry : 0100
                       Entry : 0100
                       Entry : 0100
                       Entry : 0100
                       Entry : 0100
                       Entry : 0100

              Structure Type : 0001 [SLLBI]
                   Data Type : 03   [Bandwidth]
Target Proximity Domain List : 00000000
Target Proximity Domain List : 00000001
Target Proximity Domain List : 00000002
Target Proximity Domain List : 00000003
Target Proximity Domain List : 00000004
Target Proximity Domain List : 00000005
Target Proximity Domain List : 00000006
Target Proximity Domain List : 00000007
                       Entry : 1200
                       Entry : 0400
                       Entry : 0200
                       Entry : 0200
                       Entry : 0100
                       Entry : 0100
                       Entry : 0100
                       Entry : 0100
```

SLIT

```
   Signature : "SLIT"    [System Locality Information Table]
  Localities : 0000000000000003
Locality   0 : 10 20 20 20 20 20 20 20
Locality   1 : FF 0A FF FF FF FF FF FF
Locality   2 : FF FF 0A FF FF FF FF FF
Locality   3 : FF FF FF 0A FF FF FF FF
Locality   4 : FF FF FF FF 0A FF FF FF
Locality   5 : FF FF FF FF FF 0A FF FF
Locality   6 : FF FF FF FF FF FF 0A FF
Locality   7 : FF FF FF FF FF FF FF 0A
```

DSDT

```
Scope (_SB)
{
  Device (S0D0)
  {
      Name (_HID, "ACPI0016" /* Compute Express Link Host Bridge */)  // _HID: Hardware
ID
      ...
      Name (_UID, 0x07)  // _UID: Unique ID
  }
  ...
  Device (S0D5)
  {
      Name (_HID, "ACPI0016" /* Compute Express Link Host Bridge */)  // _HID: Hardware
ID
      ...
      Name (_UID, 0x06)  // _UID: Unique ID
  }
}
```

Overview

This section presents the configuration process of a CXL Type-3 memory device, and how it is ultimately exposed to users as either a `DAX` device or normal memory pages via the kernel’s page allocator.

Portions marked with a bullet are points at which certain kernel objects are generated.

1. Early Boot

a. BIOS, Build, and Boot Parameters

1. EFI_MEMORY_SP
2. CONFIG_EFI_SOFT_RESERVE
3. CONFIG_MHP_DEFAULT_ONLINE_TYPE
4. nosoftreserve

b. Memory Map Creation

1. EFI Memory Map / E820 Consulted for Soft-Reserved

- CXL Memory is set aside to be handled by the CXL driver
- IO Resources are created for CFMWS entry

c. NUMA Node Creation

- ACPI CEDT and SRAT table are used to create Nodes from Proximity domains (PXM)

d. Memory Tier Creation

- A default memory_tier is created with all nodes.

e. Contiguous Memory Allocation

- Any requested CMA is allocated from Online nodes

f. Init Finishes, Drivers start probing

2. ACPI and PCI Drivers

a. Detect CXL device, marking it for probe by CXL driver
b. This portion will not be covered specifically.

3. CXL Driver Operation

a. Base device creation

- root, port, and memdev devices created
- CEDT CFMWS IO Resource creation

b. Decoder creation

- root, switch, and endpoint decoders created

c. Logical device creation

- memory_region and endpoint devices created

d. Devices are associated with each other

- If auto-decoder (BIOS-programmed decoders), driver validates configurations, builds associations, and locks configs at probe time.
- If user-configured, validation and associations are built at decoder-commit time.

e. Regions surfaced as DAX region

- dax_region created
- DAX device created via DAX driver

4. DAX Driver Operation

a. DAX driver surfaces DAX region as one of two dax device modes

- kmem - dax device is converted to hotplug memory blocks
    - DAX kmem IO resource creation
- hmem - dax device is left as daxdev to be accessed as a file.
    - If hmem, journey ends here.

b. DAX kmem surfaces memory region to Memory Hotplug to add to page allocator as “driver managed memory”

5. Memory Hotplug

a. mhp component surfaces a dax device memory region as multiple memory blocks to the page allocator

   - blocks appear in `/sys/bus/memory/devices` and linked to a NUMA node

b. blocks are onlined into the requested zone (NORMAL or MOVABLE)

   - Memory is marked “Driver Managed” to avoid kexec from using it as region for kernel updates

Linux Init (Early Boot)

Linux configuration is split into two major steps: Early-Boot and everything else.

During early boot, Linux sets up immutible resources (such as numa nodes), while later operations include things like driver probe and memory hotplug. Linux may read EFI and ACPI information throughout this process to configure logical representations of the devices.

During Linux Early Boot stage (functions in the kernel that have the __init decorator), the system takes the resources created by EFI/BIOS (ACPI tables) and turns them into resources that the kernel can consume.

BIOS, Build and Boot Options

There are 4 pre-boot options that need to be considered during kernel build which dictate how memory will be managed by Linux during early boot.

- EFI_MEMORY_SP
  - BIOS/EFI Option that dictates whether memory is SystemRAM or Specific Purpose. Specific Purpose memory will be deferred to drivers to manage - and not immediately exposed as system RAM.
- CONFIG_EFI_SOFT_RESERVE
  - Linux Build config option that dictates whether the kernel supports Specific Purpose memory.
- CONFIG_MHP_DEFAULT_ONLINE_TYPE
  - Linux Build config that dictates whether and how Specific Purpose memory converted to a dax device should be managed (left as DAX or onlined as SystemRAM in ZONE_NORMAL or ZONE_MOVABLE).
- nosoftreserve
  - Linux kernel boot option that dictates whether Soft Reserve should be supported. Similar to CONFIG_EFI_SOFT_RESERVE.

Memory Map Creation

While the kernel parses the EFI memory map, if `Specific Purpose` memory is supported and detect, it will set this region aside as `SOFT_RESERVED`.

If `EFI_MEMORY_SP=0`, `CONFIG_EFI_SOFT_RESERVE=n`, or `nosoftreserve=y` - Linux will default a CXL device memory region to SystemRAM. This will expose the memory to the kernel page allocator in `ZONE_NORMAL`, making it available for use for most allocations (including `struct page` and page tables).

If *Specific Purpose* is set and supported, `CONFIG_MHP_DEFAULT_ONLINE_TYPE_*` dictates whether the memory is onlined by default (`_OFFLINE` or `_ONLINE_*`), and if online which zone to online this memory to by default (`_NORMAL` or `_MOVABLE`).

If placed in `ZONE_MOVABLE`, the memory will not be available for most kernel allocations (such as `struct page` or page tables). This may significant impact performance depending on the memory capacity of the system.

NUMA Node Reservation

Linux refers to the proximity domains (`PXM`) defined in the SRAT to create NUMA nodes in `acpi_numa_init`. Typically, there is a 1:1 relation between `PXM` and NUMA node IDs.

SRAT is the only ACPI defined way of defining Proximity Domains. Linux chooses to, at most, map those 1:1 with NUMA nodes. CEDT adds a description of SPA ranges which Linux may wish to map to one or more NUMA nodes

If there are CXL ranges in the CFMWS but not in SRAT, then a fake `PXM` is created (as of v6.15). In the future, Linux may reject CFMWS not described by SRAT due to the ambiguity of proximity domain association.

It is important to note that NUMA node creation cannot be done at runtime. All possible NUMA nodes are identified at `__init` time, more specifically during `mm_init`. The CEDT and SRAT must contain sufficient `PXM` data for Linux to identify NUMA nodes their associated memory regions.

The relevant code exists in: `linux/drivers/acpi/numa/srat.c`.

See the Example Platform Configurations section for more information.

Memory Tiers Creation

Memory tiers are a collection of NUMA nodes grouped by performance characteristics. During `__init`, Linux initializes the system with a default memory tier that contains all nodes marked `N_MEMORY`.

`memory_tier_init` is called at boot for all nodes with memory online by default. `memory_tier_late_init` is called during late-init for nodes setup during driver configuration.

Nodes are only marked `N_MEMORY` if they have *online* memory.

Tier membership can be inspected in

```
/sys/devices/virtual/memory_tiering/memory_tierN/nodelist
0-1
```

If nodes are grouped which have clear difference in performance, check the HMAT and CDAT information for the CXL nodes. All nodes default to the DRAM tier, unless HMAT/CDAT information is reported to the memory_tier component via *access_coordinates*.

Contiguous Memory Allocation

The contiguous memory allocator (CMA) enables reservation of contiguous memory regions on NUMA nodes during early boot. However, CMA cannot reserve memory on NUMA nodes that are not online during early boot.

```
void __init hugetlb_cma_reserve(int order) {
  if (!node_online(nid))
    /* do not allow reservations */
}
```

This means if users intend to defer management of CXL memory to the driver, CMA cannot be used to guarantee huge page allocations. If enabling CXL memory as SystemRAM in *ZONE_NORMAL* during early boot, CMA reservations per-node can be made with the `cma_pernuma` or `numa_cma` kernel command line parameters.

CXL Driver Operation

The devices described in this section are present in

```
/sys/bus/cxl/devices/
/dev/cxl/
```

The `cxl-cli` library, maintained as part of the NDTCL project, may be used to script interactions with these devices.

CXL-CLI Reference

todo

Drivers

The CXL driver is split into a number of drivers.

- cxl_core - fundamental init interface and core object creation
- cxl_port - initializes root and provides port enumeration interface.
- cxl_acpi - initializes root decoders and interacts with ACPI data.
- cxl_p/mem - initializes memory devices
- cxl_pci - uses cxl_port to enumates the actual fabric hierarchy.

Driver Devices

Here is an example from a single-socket system with 4 host bridges. Two host bridges have a single memory device attached, and the devices are interleaved into a single memory region. The memory region has been converted to dax.

```
# ls /sys/bus/cxl/devices/
  dax_region0  decoder3.0  decoder6.0  mem0   port3
  decoder0.0   decoder4.0  decoder6.1  mem1   port4
  decoder1.0   decoder5.0  endpoint5   port1  region0
  decoder2.0   decoder5.1  endpoint6   port2  root0
```

For this section we'll explore the devices present in this configuration, but we'll explore more configurations in-depth in example configurations below.

Base Devices

Most devices in a CXL fabric are a *port* of some kind (because each device mostly routes request from one device to the next, rather than provide a manageable service).

Root

The *CXL Root* is logical object created by the *cxl_acpi* driver during `cxl_acpi_probe` - if the `ACPI0017` *Compute Express Link Root Object* Device Class is found.

The Root contains links to:

- *Host Bridge Ports* defined by ACPI CEDT CHBS.
- *Root Decoders* defined by ACPI CEDT CFMWS.

```
# ls /sys/bus/cxl/devices/root0
  decoder0.0          dport0  dport5    port2  subsystem
  decoders_committed  dport1  modalias  port3  uevent
  devtype             dport4  port1     port4  uport

# cat /sys/bus/cxl/devices/root0/devtype
  cxl_port

# cat port1/devtype
  cxl_port

# cat decoder0.0/devtype
  cxl_decoder_root
```

The root is first *logical port* in the CXL fabric, as presented by the Linux CXL driver. The *CXL root* is a special type of *switch port*, in that it only has downstream port connections.

Port

A *port* object is better described as a *switch port*. It may represent a host bridge to the root or an actual switch port on a switch. A *switch port* contains one or more decoders used to route memory requests downstream ports, which may be connected to another *switch port* or an *endpoint port*.

```
# ls /sys/bus/cxl/devices/port1
  decoder1.0          dport0    driver     parent_dport  uport
  decoders_committed  dport113  endpoint5  subsystem
  devtype             dport2    modalias   uevent

# cat devtype
  cxl_port

# cat decoder1.0/devtype
  cxl_decoder_switch

# cat endpoint5/devtype
  cxl_port
```

CXL *Host Bridges* in the fabric are probed during `cxl_acpi_probe` at the time the *CXL Root* is probed. The allows for the immediate logical connection to between between the root and host bridge.

- The root has a downstream port connection to a host bridge
- The host bridge has an upstream port connection to the root.
- The host bridge has one or more downstream port connections to switch or endpoint ports.

A *Host Bridge* is a special type of CXL *switch port*. It is explicitly defined in the ACPI specification via *ACPI0016* ID. *Host Bridge* ports will be probed at *acpi_probe* time, while similar ports on an actual switch will be probed later. Otherwise, switch and host bridge ports look very similar - the both contain switch decoders which route accesses between upstream and downstream ports.

Endpoint

An *endpoint* is a terminal port in the fabric. This is a *logical device*, and may be one of many *logical devices* presented by a memory device. It is still considered a type of *port* in the fabric.

An *endpoint* contains *endpoint decoders* available for use and the *Coherent Device Attribute Table* (CDAT) used to describe the capabilities of the device.

```
# ls /sys/bus/cxl/devices/endpoint5
  CDAT         decoders_committed  modalias      uevent
  decoder5.0   devtype             parent_dport  uport
  decoder5.1   driver              subsystem

# cat /sys/bus/cxl/devices/endpoint5/devtype
  cxl_port

# cat /sys/bus/cxl/devices/endpoint5/decoder5.0/devtype
  cxl_decoder_endpoint
```

Memory Device (memdev)

A *memdev* is probed and added by the *cxl_pci* driver in `cxl_pci_probe` and is managed by the *cxl_mem* driver. It primarily provides the *IOCTL* interface to a memory device, via `/dev/cxl/memN`, and exposes various device configuration data.

```
# ls /sys/bus/cxl/devices/mem0
  dev       firmware_version    payload_max  security   uevent
  driver    label_storage_size  pmem         serial
  firmware  numa_node           ram          subsystem
```

Decoders

A *Decoder* is short for a CXL Host-Managed Device Memory (HDM) Decoder. It is a device that routes accesses through the CXL fabric to an endpoint, and at the endpoint translates a *Host Physical* to *Device Physical* Addressing.

The CXL 3.1 specification heavily implies that only endpoint decoders should engage in translation of *Host Physical Address* to *Device Physical Address*.

```
8.2.4.20 CXL HDM Decoder Capability Structure

IMPLEMENTATION NOTE
CXL Host Bridge and Upstream Switch Port Decode Flow

IMPLEMENTATION NOTE
Device Decode Logic
```

These notes imply that there are two logical groups of decoders.

- Routing Decoder - a decoder which routes accesses but does not translate addresses from HPA to DPA.
- Translating Decoder - a decoder which translates accesses from HPA to DPA for an endpoint to service.

The CXL drivers distinguish 3 decoder types: root, switch, and endpoint. Only endpoint decoders are Translating Decoders, all others are Routing Decoders.

Note

PLATFORM VENDORS BE AWARE

Linux makes a strong assumption that endpoint decoders are the only decoder in the fabric that actively translates HPA to DPA. Linux assumes routing decoders pass the HPA unchanged to the next decoder in the fabric.

It is therefore assumed that any given decoder in the fabric will have an address range that is a subset of its upstream port decoder. Any deviation from this scheme undefined per the specification. Linux prioritizes spec-defined / architectural behavior.

Decoders may have one or more *Downstream Targets* if configured to interleave memory accesses. This will be presented in sysfs via the `target_list` parameter.

Root Decoder

A *Root Decoder* is logical construct of the physical address and interleave configurations present in the ACPI CEDT CFMWS. Linux presents this information as a decoder present in the *CXL Root*. We consider this a *Root Decoder*, though technically it exists on the boundary of the CXL specification and platform-specific CXL root implementations.

Linux considers these logical decoders a type of *Routing Decoder*, and is the first decoder in the CXL fabric to recieve a memory access from the platform's memory controllers.

Root Decoders are created during `cxl_acpi_probe`. One root decoder is created per CFMWS entry in the ACPI CEDT.

The `target_list` parameter is filled by the CFMWS target fields. Targets of a root decoder are *Host Bridges*, which means interleave done at the root decoder level is an *Inter-Host-Bridge Interleave*.

Only root decoders are capable of *Inter-Host-Bridge Interleave*.

Such interleaves must be configured by the platform and described in the ACPI CEDT CFMWS, as the target CXL host bridge UIDs in the CFMWS must match the CXL host bridge UIDs in the ACPI CEDT CHBS and ACPI DSDT.

Interleave settings in a rootdecoder describe how to interleave accesses among the *immediate downstream targets*, not the entire interleave set.

The memory range described in the root decoder is used to

1. Create a memory region (`region0` in this example), and
2. Associate the region with an IO Memory Resource (`kernel/resource.c`)

```
# ls /sys/bus/cxl/devices/decoder0.0/
  cap_pmem           devtype                 region0
  cap_ram            interleave_granularity  size
  cap_type2          interleave_ways         start
  cap_type3          locked                  subsystem
  create_ram_region  modalias                target_list
  delete_region      qos_class               uevent

# cat /sys/bus/cxl/devices/decoder0.0/region0/resource
  0xc050000000
```

The IO Memory Resource is created during early boot when the CFMWS region is identified in the EFI Memory Map or E820 table (on x86).

Root decoders are defined as a separate devtype, but are also a type of *Switch Decoder* due to having downstream targets.

```
# cat /sys/bus/cxl/devices/decoder0.0/devtype
  cxl_decoder_root
```

Switch Decoder

Any non-root, translating decoder is considered a *Switch Decoder*, and will present with the type `cxl_decoder_switch` . Both *Host Bridge* and *CXL Switch* (device) decoders are of type `cxl_decoder_switch` .

```
# ls /sys/bus/cxl/devices/decoder1.0/
  devtype                 locked    size       target_list
  interleave_granularity  modalias  start      target_type
  interleave_ways         region    subsystem  uevent

# cat /sys/bus/cxl/devices/decoder1.0/devtype
  cxl_decoder_switch

# cat /sys/bus/cxl/devices/decoder1.0/region
  region0
```

A *Switch Decoder* has associations between a region defined by a root decoder and downstream target ports. Interleaving done within a switch decoder is a multi-downstream-port interleave (or *Intra-Host-Bridge Interleave* for host bridges).

Interleave settings in a switch decoder describe how to interleave accesses among the *immediate downstream targets*, not the entire interleave set.

Switch decoders are created during `cxl_switch_port_probe` in the `cxl_port` driver, and is created based on a PCI device's DVSEC registers.

Switch decoder programming is validated during probe if the platform programs them during boot (See *Auto Decoders* below), or on commit if programmed at runtime (See *Runtime Programming* below).

Endpoint Decoder

Any decoder attached to a *terminal* point in the CXL fabric (*An Endpoint*) is considered an *Endpoint Decoder*. Endpoint decoders are of type `cxl_decoder_endpoint`.

```
# ls /sys/bus/cxl/devices/decoder5.0
  devtype                 locked    start
  dpa_resource            modalias  subsystem
  dpa_size                mode      target_type
  interleave_granularity  region    uevent
  interleave_ways         size

# cat /sys/bus/cxl/devices/decoder5.0/devtype
  cxl_decoder_endpoint

# cat /sys/bus/cxl/devices/decoder5.0/region
  region0
```

An *Endpoint Decoder* has an association with a region defined by a root decoder and describes the device-local resource associated with this region.

Unlike root and switch decoders, endpoint decoders translate *Host Physical* to *Device Physical* address ranges. The interleave settings on an endpoint therefore describe the entire *interleave set*.

Device Physical Address regions must be committed in-order. For example, the DPA region starting at 0x80000000 cannot be committed before the DPA region starting at 0x0.

As of Linux v6.15, Linux does not support *imbalanced* interleave setups, all endpoints in an interleave set are are expected to have the same interleave settings (granularity and ways must be the same).

Endpoint decoders are created during `cxl_endpoint_port_probe` in the `cxl_port` driver, and is created based on a PCI device's DVSEC registers.

Regions

Memory Region

A *Memory Region* is a logical construct that connects a set of CXL ports in the fabric to an IO Memory Resource. It is ultimately used to expose the memory on these devices to the DAX subsystem via a *DAX Region*.

An example RAM region:

```
# ls /sys/bus/cxl/devices/region0/
  access0      devtype                 modalias  subsystem  uuid
  access1      driver                  mode      target0
  commit       interleave_granularity  resource  target1
  dax_region0  interleave_ways         size      uevent
```

A memory region can be constructed during endpoint probe, if decoders were programmed by BIOS/EFI (see *Auto Decoders*), or by creating a region manually via a *Root Decoder*'s `create_ram_region` or `create_pmem_region` interfaces.

The interleave settings in a *Memory Region* describe the configuration of the *Interleave Set* - and are what can be expected to be seen in the endpoint interleave settings.

DAX Region

A *DAX Region* is used to convert a CXL *Memory Region* to a DAX device. A DAX device may then be accessed directly via a file descriptor interface, or converted to System RAM via the DAX kmem driver. See the DAX driver section for more details.

```
# ls /sys/bus/cxl/devices/dax_region0/
  dax0.0      devtype  modalias   uevent
  dax_region  driver   subsystem
```

Mailbox Interfaces

A mailbox command interface for each device is exposed in

```
/dev/cxl/mem0
/dev/cxl/mem1
```

These mailboxes may receive any specification-defined command. Raw commands (custom commands) can only be sent to these interfaces if the build config `CXL_MEM_RAW_COMMANDS` is set. This is considered a debug and/or development interface, not an officially supported mechanism for creation of vendor-specific commands (see the *fwctl* subsystem for that).

Decoder Programming

Runtime Programming

During probe, the only decoders *required* to be programmed are *Root Decoders*. In reality, *Root Decoders* are a logical construct to describe the memory region and interleave configuration at the host bridge level - as described in the ACPI CEDT CFMWS.

All other *Switch* and *Endpoint* decoders may be programmed by the user at runtime - if the platform supports such configurations.

This interaction is what creates a *Software Defined Memory* environment.

See the `cxl-cli` documentation for more information about how to configure CXL decoders at runtime.

Auto Decoders

Auto Decoders are decoders programmed by BIOS/EFI at boot time, and are almost always locked (cannot be changed). This is done by a platform which may have a static configuration - or certain quirks which may prevent dynamic runtime changes to the decoders (such as requiring additional controller programming within the CPU complex outside the scope of CXL).

Auto Decoders are probed automatically as long as the devices and memory regions they are associated with probe without issue. When probing Auto Decoders, the driver's primary responsibility is to ensure the fabric is sane - as-if validating runtime programmed regions and decoders.

If Linux cannot validate auto-decoder configuration, the memory will not be surfaced as a DAX device - and therefore not be exposed to the page allocator - effectively stranding it.

Interleave

The Linux CXL driver supports *Cross-Link First* interleave. This dictates how interleave is programmed at each decoder step, as the driver validates the relationships between a decoder and it's parent.

For example, in a *Cross-Link First* interleave setup with 16 endpoints attached to 4 host bridges, linux expects the following ways/granularity across the root, host bridge, and endpoints respectively.

```
                  ways   granularity
root               4        256
host bridge        4       1024
endpoint          16        256
```

At the root, every a given access will be routed to the `((HPA / 256) % 4)th` target host bridge. Within a host bridge, every `((HPA / 1024) % 4)th` target endpoint. Each endpoint will translate the access based on the entire 16 device interleave set.

Unbalanced interleave sets are not supported - decoders at a similar point in the hierarchy (e.g. all host bridge decoders) must have the same ways and granularity configuration.

At Root

Root decoder interleave is defined by the ACPI CEDT CFMWS. The CEDT may actually define multiple CFMWS configurations to describe the same physical capacity - with the intent to allow users to decide at runtime whether to online memory as interleaved or non-interleaved.

```
           Subtable Type : 01 [CXL Fixed Memory Window Structure]
     Window base address : 0000000100000000
             Window size : 0000000100000000
Interleave Members (2^n) : 00
   Interleave Arithmetic : 00
            First Target : 00000007

           Subtable Type : 01 [CXL Fixed Memory Window Structure]
     Window base address : 0000000200000000
             Window size : 0000000100000000
Interleave Members (2^n) : 00
   Interleave Arithmetic : 00
            First Target : 00000006

           Subtable Type : 01 [CXL Fixed Memory Window Structure]
     Window base address : 0000000300000000
             Window size : 0000000200000000
Interleave Members (2^n) : 01
   Interleave Arithmetic : 00
            First Target : 00000007
             Next Target : 00000006
```

In this example, the CFMWS defines two discrete non-interleaved 4GB regions for each host bridge, and one interleaved 8GB region that targets both. This would result in 3 root decoders presenting in the root.

```
# ls /sys/bus/cxl/devices/root0
  decoder0.0  decoder0.1  decoder0.2

# cat /sys/bus/cxl/devices/decoder0.0/target_list start size
  7
  0x100000000
  0x100000000

# cat /sys/bus/cxl/devices/decoder0.1/target_list start size
  6
  0x200000000
  0x100000000

# cat /sys/bus/cxl/devices/decoder0.2/target_list start size
  7,6
  0x300000000
  0x200000000
```

These decoders are not runtime programmable. They are used to generate a *Memory Region* to bring this memory online with runtime programmed settings at the *Switch* and *Endpoint* decoders.

At Host Bridge or Switch

Host Bridge and *Switch* decoders are programmable via the following fields:

- `start` - the HPA region associated with the memory region
- `size` - the size of the region
- `target_list` - the list of downstream ports
- `interleave_ways` - the number downstream ports to interleave across
- `interleave_granularity` - the granularity to interleave at.

Linux expects the `interleave_granularity` of switch decoders to be derived from their upstream port connections. In *Cross-Link First* interleave configurations, the `interleave_granularity` of a decoder is equal to `parent_interleave_granularity * parent_interleave_ways`.

At Endpoint

Endpoint Decoders are programmed similar to Host Bridge and Switch decoders, with the exception that the ways and granularity are defined by the interleave set (e.g. the interleave settings defined by the associated *Memory Region*).

- `start` - the HPA region associated with the memory region

- `size` - the size of the region
- `interleave_ways` - the number endpoints in the interleave set
- `interleave_granularity` - the granularity to interleave at.

These settings are used by endpoint decoders to *Translate* memory requests from HPA to DPA. This is why they must be aware of the entire interleave set.

Linux does not support unbalanced interleave configurations. As a result, all endpoints in an interleave set must have the same ways and granularity.

Example Configurations

Single Device

This cxl-cli configuration dump shows the following host configuration:

- A single socket system with one CXL root
- CXL Root has Four (4) CXL Host Bridges
- One CXL Host Bridges has a single CXL Memory Expander Attached
- No interleave is present.

This output is generated by `cxl list -v` and describes the relationships between objects exposed in `/sys/bus/cxl/devices/`.

```
[
  {
      "bus":"root0",
      "provider":"ACPI.CXL",
      "nr_dports":4,
      "dports":[
          {
              "dport":"pci0000:00",
              "alias":"ACPI0016:01",
              "id":0
          },
          {
              "dport":"pci0000:a8",
              "alias":"ACPI0016:02",
              "id":4
          },
          {
              "dport":"pci0000:2a",
              "alias":"ACPI0016:03",
              "id":1
          },
          {
              "dport":"pci0000:d2",
              "alias":"ACPI0016:00",
              "id":5
          }
      ],
```

This chunk shows the CXL “bus” (root0) has 4 downstream ports attached to CXL Host Bridges. The *Root* can be considered the singular upstream port attached to the platform’s memory controller - which routes memory requests to it.

The *ports:root0* section lays out how each of these downstream ports are configured. If a port is not configured (id’s 0, 1, and 4), they are omitted.

```
"ports:root0":[
    {
        "port":"port1",
        "host":"pci0000:d2",
        "depth":1,
        "nr_dports":3,
        "dports":[
            {
                "dport":"0000:d2:01.1",
                "alias":"device:02",
                "id":0
            },
            {
                "dport":"0000:d2:01.3",
                "alias":"device:05",
                "id":2
            },
            {
                "dport":"0000:d2:07.1",
                "alias":"device:0d",
                "id":113
            }
        ],
```

This chunk shows the available downstream ports associated with the CXL Host Bridge `port1`. In this case, `port1` has 3 avaiable downstream ports: `dport1`, `dport2`, and `dport113`..

```
"endpoints:port1":[
    {
        "endpoint":"endpoint5",
        "host":"mem0",
        "parent_dport":"0000:d2:01.1",
        "depth":2,
        "memdev":{
            "memdev":"mem0",
            "ram_size":137438953472,
            "serial":0,
            "numa_node":0,
            "host":"0000:d3:00.0"
        },
        "decoders:endpoint5":[
            {
                "decoder":"decoder5.0",
                "resource":825975898112,
                "size":137438953472,
                "interleave_ways":1,
                "region":"region0",
                "dpa_resource":0,
                "dpa_size":137438953472,
                "mode":"ram"
            }
        ]
    }
],
```

This chunk shows the endpoints attached to the host bridge `port1`.

`endpoint5` contains a single configured decoder `decoder5.0` which has the same interleave configuration as `region0` (shown later).

Next we have the decoders belonging to the host bridge:

```
    "decoders:port1":[
        {
            "decoder":"decoder1.0",
            "resource":825975898112,
            "size":137438953472,
            "interleave_ways":1,
            "region":"region0",
            "nr_targets":1,
            "targets":[
                {
                    "target":"0000:d2:01.1",
                    "alias":"device:02",
                    "position":0,
                    "id":0
                }
            ]
        }
    ]
},
```

Host Bridge `port1` has a single decoder (`decoder1.0`), whose only target is `dport1` - which is attached to `endpoint5`.

The next chunk shows the three CXL host bridges without attached endpoints.

```
    {
        "port":"port2",
        "host":"pci0000:00",
        "depth":1,
        "nr_dports":2,
        "dports":[
            {
                "dport":"0000:00:01.3",
                "alias":"device:55",
                "id":2
            },
            {
                "dport":"0000:00:07.1",
                "alias":"device:5d",
                "id":113
            }
        ]
    },
    {
        "port":"port3",
        "host":"pci0000:a8",
        "depth":1,
        "nr_dports":1,
        "dports":[
            {
                "dport":"0000:a8:01.1",
                "alias":"device:c3",
                "id":0
            }
        ]
    },
    {
        "port":"port4",
        "host":"pci0000:2a",
        "depth":1,
        "nr_dports":1,
        "dports":[
            {
                "dport":"0000:2a:01.1",
                "alias":"device:d0",
                "id":0
            }
        ]
    }
  ],
```

Next we have the *Root Decoders* belonging to `root0`. This root decoder is a pass-through decoder because `interleave_ways` is set to `1`.

This information is generated by the CXL driver reading the ACPI CEDT CMFWS.

```
"decoders:root0":[
    {
        "decoder":"decoder0.0",
        "resource":825975898112,
        "size":137438953472,
        "interleave_ways":1,
        "max_available_extent":0,
        "volatile_capable":true,
        "nr_targets":1,
        "targets":[
            {
                "target":"pci0000:d2",
                "alias":"ACPI0016:00",
                "position":0,
                "id":5
            }
        ],
```

Finally we have the *Memory Region* associated with the *Root Decoder* `decoder0.0`. This region describes the discrete region associated with the lone device.

```
                "regions:decoder0.0":[
                    {
                        "region":"region0",
                        "resource":825975898112,
                        "size":137438953472,
                        "type":"ram",
                        "interleave_ways":1,
                        "decode_state":"commit",
                        "mappings":[
                            {
                                "position":0,
                                "memdev":"mem0",
                                "decoder":"decoder5.0"
                            }
                        ]
                    }
                ]
            }
        ]
    }
]
```

Inter-Host-Bridge Interleave

This cxl-cli configuration dump shows the following host configuration:

- A single socket system with one CXL root
- CXL Root has Four (4) CXL Host Bridges
- Two CXL Host Bridges have a single CXL Memory Expander Attached
- The CXL root is configured to interleave across the two host bridges.

This output is generated by `cxl list -v` and describes the relationships between objects exposed in `/sys/bus/cxl/devices/`.

```
[
  {
      "bus":"root0",
      "provider":"ACPI.CXL",
      "nr_dports":4,
      "dports":[
          {
              "dport":"pci0000:00",
              "alias":"ACPI0016:01",
              "id":0
          },
          {
              "dport":"pci0000:a8",
              "alias":"ACPI0016:02",
              "id":4
          },
          {
              "dport":"pci0000:2a",
              "alias":"ACPI0016:03",
              "id":1
          },
          {
              "dport":"pci0000:d2",
              "alias":"ACPI0016:00",
              "id":5
          }
      ],
```

This chunk shows the CXL “bus” (root0) has 4 downstream ports attached to CXL Host Bridges. The *Root* can be considered the singular upstream port attached to the platform’s memory controller - which routes memory requests to it.

The *ports:root0* section lays out how each of these downstream ports are configured. If a port is not configured (id’s 0 and 1), they are omitted.

```
"ports:root0":[
    {
        "port":"port1",
        "host":"pci0000:d2",
        "depth":1,
        "nr_dports":3,
        "dports":[
            {
                "dport":"0000:d2:01.1",
                "alias":"device:02",
                "id":0
            },
            {
                "dport":"0000:d2:01.3",
                "alias":"device:05",
                "id":2
            },
            {
                "dport":"0000:d2:07.1",
                "alias":"device:0d",
                "id":113
            }
        ],
```

This chunk shows the available downstream ports associated with the CXL Host Bridge `port1`. In this case, `port1` has 3 avaiable downstream ports: `dport1`, `dport2`, and `dport113`..

```
"endpoints:port1":[
    {
        "endpoint":"endpoint5",
        "host":"mem0",
        "parent_dport":"0000:d2:01.1",
        "depth":2,
        "memdev":{
            "memdev":"mem0",
            "ram_size":137438953472,
            "serial":0,
            "numa_node":0,
            "host":"0000:d3:00.0"
        },
        "decoders:endpoint5":[
            {
                "decoder":"decoder5.0",
                "resource":825975898112,
                "size":274877906944,
                "interleave_ways":2,
                "interleave_granularity":256,
                "region":"region0",
                "dpa_resource":0,
                "dpa_size":137438953472,
                "mode":"ram"
            }
        ]
    }
],
```

This chunk shows the endpoints attached to the host bridge `port1`.

`endpoint5` contains a single configured decoder `decoder5.0` which has the same interleave configuration as `region0` (shown later).

Next we have the decodesr belonging to the host bridge:

```
    "decoders:port1":[
        {
            "decoder":"decoder1.0",
            "resource":825975898112,
            "size":274877906944,
            "interleave_ways":1,
            "region":"region0",
            "nr_targets":1,
            "targets":[
                {
                    "target":"0000:d2:01.1",
                    "alias":"device:02",
                    "position":0,
                    "id":0
                }
            ]
        }
    ]
},
```

Host Bridge `port1` has a single decoder (`decoder1.0`), whose only target is `dport1` - which is attached to `endpoint5`.

The following chunk shows a similar configuration for Host Bridge `port3`, the second host bridge with a memory device attached.

```
{
    "port":"port3",
    "host":"pci0000:a8",
    "depth":1,
    "nr_dports":1,
    "dports":[
        {
            "dport":"0000:a8:01.1",
            "alias":"device:c3",
            "id":0
        }
    ],
    "endpoints:port3":[
        {
            "endpoint":"endpoint6",
            "host":"mem1",
            "parent_dport":"0000:a8:01.1",
            "depth":2,
            "memdev":{
                "memdev":"mem1",
                "ram_size":137438953472,
                "serial":0,
                "numa_node":0,
                "host":"0000:a9:00.0"
            },
            "decoders:endpoint6":[
                {
                    "decoder":"decoder6.0",
                    "resource":825975898112,
                    "size":274877906944,
                    "interleave_ways":2,
                    "interleave_granularity":256,
                    "region":"region0",
                    "dpa_resource":0,
                    "dpa_size":137438953472,
                    "mode":"ram"
                }
            ]
        }
    ],
    "decoders:port3":[
        {
            "decoder":"decoder3.0",
            "resource":825975898112,
            "size":274877906944,
            "interleave_ways":1,
            "region":"region0",
            "nr_targets":1,
            "targets":[
                {
                    "target":"0000:a8:01.1",
                    "alias":"device:c3",
                    "position":0,
                    "id":0
                }
            ]
        }
    ]
},
```

The next chunk shows the two CXL host bridges without attached endpoints.

```
    {
        "port":"port2",
        "host":"pci0000:00",
        "depth":1,
        "nr_dports":2,
        "dports":[
            {
                "dport":"0000:00:01.3",
                "alias":"device:55",
                "id":2
            },
            {
                "dport":"0000:00:07.1",
                "alias":"device:5d",
                "id":113
            }
        ]
    },
    {
        "port":"port4",
        "host":"pci0000:2a",
        "depth":1,
        "nr_dports":1,
        "dports":[
            {
                "dport":"0000:2a:01.1",
                "alias":"device:d0",
                "id":0
            }
        ]
    }
],
```

Next we have the *Root Decoders* belonging to `root0`. This root decoder applies the interleave across the downstream ports `port1` and `port3` - with a granularity of 256 bytes.

This information is generated by the CXL driver reading the ACPI CEDT CMFWS.

```
"decoders:root0":[
    {
        "decoder":"decoder0.0",
        "resource":825975898112,
        "size":274877906944,
        "interleave_ways":2,
        "interleave_granularity":256,
        "max_available_extent":0,
        "volatile_capable":true,
        "nr_targets":2,
        "targets":[
            {
                "target":"pci0000:a8",
                "alias":"ACPI0016:02",
                "position":1,
                "id":4
            },
            {
                "target":"pci0000:d2",
                "alias":"ACPI0016:00",
                "position":0,
                "id":5
            }
        ],
```

Finally we have the *Memory Region* associated with the *Root Decoder* `decoder0.0`. This region describes the overall interleave configuration of the interleave set.

```
              "regions:decoder0.0":[
                  {
                      "region":"region0",
                      "resource":825975898112,
                      "size":274877906944,
                      "type":"ram",
                      "interleave_ways":2,
                      "interleave_granularity":256,
                      "decode_state":"commit",
                      "mappings":[
                          {
                              "position":1,
                              "memdev":"mem1",
                              "decoder":"decoder6.0"
                          },
                          {
                              "position":0,
                              "memdev":"mem0",
                              "decoder":"decoder5.0"
                          }
                      ]
                  }
              ]
          }
      ]
  }
]
```

Intra-Host-Bridge Interleave

This cxl-cli configuration dump shows the following host configuration:

- A single socket system with one CXL root
- CXL Root has Four (4) CXL Host Bridges
- One (1) CXL Host Bridges has two CXL Memory Expanders Attached
- The Host bridge decoder is programmed to interleave across the expanders.

This output is generated by `cxl list -v` and describes the relationships between objects exposed in `/sys/bus/cxl/devices/`.

```
[
  {
      "bus":"root0",
      "provider":"ACPI.CXL",
      "nr_dports":4,
      "dports":[
          {
              "dport":"pci0000:00",
              "alias":"ACPI0016:01",
              "id":0
          },
          {
              "dport":"pci0000:a8",
              "alias":"ACPI0016:02",
              "id":4
          },
          {
              "dport":"pci0000:2a",
              "alias":"ACPI0016:03",
              "id":1
          },
          {
              "dport":"pci0000:d2",
              "alias":"ACPI0016:00",
              "id":5
          }
      ],
```

This chunk shows the CXL "bus" (root0) has 4 downstream ports attached to CXL Host Bridges. The *Root* can be considered the singular upstream port attached to the platform's memory controller - which routes memory requests to it.

The *ports:root0* section lays out how each of these downstream ports are configured. If a port is not configured (id's 0 and 1), they are omitted.

```
"ports:root0":[
    {
        "port":"port1",
        "host":"pci0000:d2",
        "depth":1,
        "nr_dports":3,
        "dports":[
            {
                "dport":"0000:d2:01.1",
                "alias":"device:02",
                "id":0
            },
            {
                "dport":"0000:d2:01.3",
                "alias":"device:05",
                "id":2
            },
            {
                "dport":"0000:d2:07.1",
                "alias":"device:0d",
                "id":113
            }
        ],
```

This chunk shows the available downstream ports associated with the CXL Host Bridge `port1`. In this case, `port1` has 3 avaiable downstream ports: `dport1`, `dport2`, and `dport113`..

```
"endpoints:port1":[
    {
        "endpoint":"endpoint5",
        "host":"mem0",
        "parent_dport":"0000:d2:01.1",
        "depth":2,
        "memdev":{
            "memdev":"mem0",
            "ram_size":137438953472,
            "serial":0,
            "numa_node":0,
            "host":"0000:d3:00.0"
        },
        "decoders:endpoint5":[
            {
                "decoder":"decoder5.0",
                "resource":825975898112,
                "size":274877906944,
                "interleave_ways":2,
                "interleave_granularity":256,
                "region":"region0",
                "dpa_resource":0,
                "dpa_size":137438953472,
                "mode":"ram"
            }
        ]
    },
    {
        "endpoint":"endpoint6",
        "host":"mem1",
        "parent_dport":"0000:d2:01.3,
        "depth":2,
        "memdev":{
            "memdev":"mem1",
            "ram_size":137438953472,
            "serial":0,
            "numa_node":0,
            "host":"0000:a9:00.0"
        },
        "decoders:endpoint6":[
            {
                "decoder":"decoder6.0",
                "resource":825975898112,
                "size":274877906944,
                "interleave_ways":2,
                "interleave_granularity":256,
                "region":"region0",
                "dpa_resource":0,
                "dpa_size":137438953472,
                "mode":"ram"
            }
        ]
    }
],
```

This chunk shows the endpoints attached to the host bridge `port1`.

`endpoint5` contains a single configured decoder `decoder5.0` which has the same interleave configuration memory region they belong to (show later).

Next we have the decoders belonging to the host bridge:

```
    "decoders:port1":[
        {
            "decoder":"decoder1.0",
            "resource":825975898112,
            "size":274877906944,
            "interleave_ways":2,
            "interleave_granularity":256,
            "region":"region0",
            "nr_targets":2,
            "targets":[
                {
                    "target":"0000:d2:01.1",
                    "alias":"device:02",
                    "position":0,
                    "id":0
                },
                {
                    "target":"0000:d2:01.3",
                    "alias":"device:05",
                    "position":1,
                    "id":0
                }
            ]
        }
    ]
},
```

Host Bridge `port1` has a single decoder (`decoder1.0`) with two targets: `dport1` and `dport3` - which are attached to `endpoint5` and `endpoint6` respectively.

The host bridge decoder interleaves these devices at a 256 byte granularity.

The next chunk shows the three CXL host bridges without attached endpoints.

```
    {
        "port":"port2",
        "host":"pci0000:00",
        "depth":1,
        "nr_dports":2,
        "dports":[
            {
                "dport":"0000:00:01.3",
                "alias":"device:55",
                "id":2
            },
            {
                "dport":"0000:00:07.1",
                "alias":"device:5d",
                "id":113
            }
        ]
    },
    {
        "port":"port3",
        "host":"pci0000:a8",
        "depth":1,
        "nr_dports":1,
        "dports":[
            {
                "dport":"0000:a8:01.1",
                "alias":"device:c3",
                "id":0
            }
        ],
    },
    {
        "port":"port4",
        "host":"pci0000:2a",
        "depth":1,
        "nr_dports":1,
        "dports":[
            {
                "dport":"0000:2a:01.1",
                "alias":"device:d0",
                "id":0
            }
        ]
    }
],
```

Next we have the *Root Decoders* belonging to `root0`. This root decoder applies the interleave across the downstream ports `port1` and `port3` - with a granularity of 256 bytes.

This information is generated by the CXL driver reading the ACPI CEDT CMFWS.

```
"decoders:root0":[
    {
        "decoder":"decoder0.0",
        "resource":825975898112,
        "size":274877906944,
        "interleave_ways":1,
        "max_available_extent":0,
        "volatile_capable":true,
        "nr_targets":2,
        "targets":[
            {
                "target":"pci0000:a8",
                "alias":"ACPI0016:02",
                "position":1,
                "id":4
            },
        ],
```

Finally we have the *Memory Region* associated with the *Root Decoder* `decoder0.0`. This region describes the overall interleave configuration of the interleave set.

```
                "regions:decoder0.0":[
                    {
                        "region":"region0",
                        "resource":825975898112,
                        "size":274877906944,
                        "type":"ram",
                        "interleave_ways":2,
                        "interleave_granularity":256,
                        "decode_state":"commit",
                        "mappings":[
                            {
                                "position":1,
                                "memdev":"mem1",
                                "decoder":"decoder6.0"
                            },
                            {
                                "position":0,
                                "memdev":"mem0",
                                "decoder":"decoder5.0"
                            }
                        ]
                    }
                ]
            }
        ]
    }
]
```

Multi-Level Interleave

This cxl-cli configuration dump shows the following host configuration:

- A single socket system with one CXL root
- CXL Root has Four (4) CXL Host Bridges

- Two CXL Host Bridges have a two CXL Memory Expanders Attached each.
- The CXL root is configured to interleave across the two host bridges.
- Each host bridge with expanders interleaves across two endpoints.

This output is generated by `cxl list -v` and describes the relationships between objects exposed in `/sys/bus/cxl/devices/`.

```
[
  {
      "bus":"root0",
      "provider":"ACPI.CXL",
      "nr_dports":4,
      "dports":[
          {
              "dport":"pci0000:00",
              "alias":"ACPI0016:01",
              "id":0
          },
          {
              "dport":"pci0000:a8",
              "alias":"ACPI0016:02",
              "id":4
          },
          {
              "dport":"pci0000:2a",
              "alias":"ACPI0016:03",
              "id":1
          },
          {
              "dport":"pci0000:d2",
              "alias":"ACPI0016:00",
              "id":5
          }
      ],
```

This chunk shows the CXL “bus” (root0) has 4 downstream ports attached to CXL Host Bridges. The *Root* can be considered the singular upstream port attached to the platform’s memory controller - which routes memory requests to it.

The *ports:root0* section lays out how each of these downstream ports are configured. If a port is not configured (id’s 0 and 1), they are omitted.

```
"ports:root0":[
    {
        "port":"port1",
        "host":"pci0000:d2",
        "depth":1,
        "nr_dports":3,
        "dports":[
            {
                "dport":"0000:d2:01.1",
                "alias":"device:02",
                "id":0
            },
            {
                "dport":"0000:d2:01.3",
                "alias":"device:05",
                "id":2
            },
            {
                "dport":"0000:d2:07.1",
                "alias":"device:0d",
                "id":113
            }
        ],
```

This chunk shows the available downstream ports associated with the CXL Host Bridge `port1`. In this case, `port1` has 3 avaiable downstream ports: `dport0`, `dport2`, and `dport113`.

```
"endpoints:port1":[
    {
        "endpoint":"endpoint5",
        "host":"mem0",
        "parent_dport":"0000:d2:01.1",
        "depth":2,
        "memdev":{
            "memdev":"mem0",
            "ram_size":137438953472,
            "serial":0,
            "numa_node":0,
            "host":"0000:d3:00.0"
        },
        "decoders:endpoint5":[
            {
                "decoder":"decoder5.0",
                "resource":825975898112,
                "size":549755813888,
                "interleave_ways":4,
                "interleave_granularity":256,
                "region":"region0",
                "dpa_resource":0,
                "dpa_size":137438953472,
                "mode":"ram"
            }
        ]
    },
    {
        "endpoint":"endpoint6",
        "host":"mem1",
        "parent_dport":"0000:d2:01.3",
        "depth":2,
        "memdev":{
            "memdev":"mem1",
            "ram_size":137438953472,
            "serial":0,
            "numa_node":0,
            "host":"0000:d3:00.0"
        },
        "decoders:endpoint6":[
            {
                "decoder":"decoder6.0",
                "resource":825975898112,
                "size":549755813888,
                "interleave_ways":4,
                "interleave_granularity":256,
                "region":"region0",
                "dpa_resource":0,
                "dpa_size":137438953472,
                "mode":"ram"
            }
        ]
    }
],
```

This chunk shows the endpoints attached to the host bridge `port1`.

`endpoint5` contains a single configured decoder `decoder5.0` which has the same interleave configuration as `region0` (shown later).

`endpoint6` contains a single configured decoder `decoder5.0` which has the same interleave configuration as `region0` (shown later).

Next we have the decoders belonging to the host bridge:

```
    "decoders:port1":[
        {
            "decoder":"decoder1.0",
            "resource":825975898112,
            "size":549755813888,
            "interleave_ways":2,
            "interleave_granularity":512,
            "region":"region0",
            "nr_targets":2,
            "targets":[
                {
                    "target":"0000:d2:01.1",
                    "alias":"device:02",
                    "position":0,
                    "id":0
                },
                {
                    "target":"0000:d2:01.3",
                    "alias":"device:05",
                    "position":2,
                    "id":0
                }
            ]
        }
    ]
},
```

Host Bridge `port1` has a single decoder (`decoder1.0`), whose targets are `dport0` and `dport2` - which are attached to `endpoint5` and `endpoint6` respectively.

The following chunk shows a similar configuration for Host Bridge `port3`, the second host bridge with a memory device attached.

```
{
    "port":"port3",
    "host":"pci0000:a8",
    "depth":1,
    "nr_dports":1,
    "dports":[
        {
            "dport":"0000:a8:01.1",
            "alias":"device:c3",
            "id":0
        },
        {
            "dport":"0000:a8:01.3",
            "alias":"device:c5",
            "id":0
        }
    ],
    "endpoints:port3":[
        {
            "endpoint":"endpoint7",
            "host":"mem2",
            "parent_dport":"0000:a8:01.1",
            "depth":2,
            "memdev":{
                "memdev":"mem2",
                "ram_size":137438953472,
                "serial":0,
                "numa_node":0,
                "host":"0000:a9:00.0"
            },
            "decoders:endpoint7":[
                {
                    "decoder":"decoder7.0",
                    "resource":825975898112,
                    "size":549755813888,
                    "interleave_ways":4,
                    "interleave_granularity":256,
                    "region":"region0",
                    "dpa_resource":0,
                    "dpa_size":137438953472,
                    "mode":"ram"
                }
            ]
        },
        {
            "endpoint":"endpoint8",
            "host":"mem3",
            "parent_dport":"0000:a8:01.3",
            "depth":2,
            "memdev":{
                "memdev":"mem3",
                "ram_size":137438953472,
                "serial":0,
                "numa_node":0,
                "host":"0000:a9:00.0"
            },
            "decoders:endpoint8":[
                {
                    "decoder":"decoder8.0",
                    "resource":825975898112,
                    "size":549755813888,
                    "interleave_ways":4,
                    "interleave_granularity":256,
                    "region":"region0",
                    "dpa_resource":0,
                    "dpa_size":137438953472,
                    "mode":"ram"
                }
            ]
        }
    ],
    "decoders:port3":[
        {
            "decoder":"decoder3.0",
            "resource":825975898112,
            "size":549755813888,
            "interleave_ways":2,
            "interleave_granularity":512,
            "region":"region0",
            "nr_targets":1,
            "targets":[
                {
                    "target":"0000:a8:01.1",
                    "alias":"device:c3",
                    "position":1,
                    "id":0
                },
                {
                    "target":"0000:a8:01.3",
                    "alias":"device:c5",
                    "position":3,
                    "id":0
                }
            ]
        }
    ]
},
```

The next chunk shows the two CXL host bridges without attached endpoints.

```
    {
        "port":"port2",
        "host":"pci0000:00",
        "depth":1,
        "nr_dports":2,
        "dports":[
            {
                "dport":"0000:00:01.3",
                "alias":"device:55",
                "id":2
            },
            {
                "dport":"0000:00:07.1",
                "alias":"device:5d",
                "id":113
            }
        ]
    },
    {
        "port":"port4",
        "host":"pci0000:2a",
        "depth":1,
        "nr_dports":1,
        "dports":[
            {
                "dport":"0000:2a:01.1",
                "alias":"device:d0",
                "id":0
            }
        ]
    }
],
```

Next we have the *Root Decoders* belonging to `root0`. This root decoder applies the interleave across the downstream ports `port1` and `port3` - with a granularity of 256 bytes.

This information is generated by the CXL driver reading the ACPI CEDT CMFWS.

```
"decoders:root0":[
    {
        "decoder":"decoder0.0",
        "resource":825975898112,
        "size":549755813888,
        "interleave_ways":2,
        "interleave_granularity":256,
        "max_available_extent":0,
        "volatile_capable":true,
        "nr_targets":2,
        "targets":[
            {
                "target":"pci0000:a8",
                "alias":"ACPI0016:02",
                "position":1,
                "id":4
            },
            {
                "target":"pci0000:d2",
                "alias":"ACPI0016:00",
                "position":0,
                "id":5
            }
        ],
```

Finally we have the *Memory Region* associated with the *Root Decoder* `decoder0.0`. This region describes the overall interleave configuration of the interleave set. So we see there are a total of `4` interleave targets across 4 endpoint decoders.

```
              "regions:decoder0.0":[
                {
                    "region":"region0",
                    "resource":825975898112,
                    "size":549755813888,
                    "type":"ram",
                    "interleave_ways":4,
                    "interleave_granularity":256,
                    "decode_state":"commit",
                    "mappings":[
                        {
                            "position":3,
                            "memdev":"mem3",
                            "decoder":"decoder8.0"
                        },
                        {
                            "position":2,
                            "memdev":"mem1",
                            "decoder":"decoder6.0"
                        }
                        {
                            "position":1,
                            "memdev":"mem2",
                            "decoder":"decoder7.0"
                        },
                        {
                            "position":0,
                            "memdev":"mem0",
                            "decoder":"decoder5.0"
                        }
                    ]
                }
              ]
          }
      ]
  }
]
```

DAX Driver Operation

The *Direct Access Device* driver was originally designed to provide a memory-like access mechanism to memory-like block-devices. It was extended to support CXL Memory Devices, which provide user-configured memory devices.

The CXL subsystem depends on the DAX subsystem to generate either:

- A file-like interface to userland via `/dev/daxN.Y`, or
- Engaging the memory-hotplug interface to add CXL memory to page allocator.

The DAX subsystem exposes this ability through the *cxl_dax_region* driver. A *dax_region* provides the translation between a CXL *memory_region* and a *DAX Device*.

DAX Device

A *DAX Device* is a file-like interface exposed in `/dev/daxN.Y`. A memory region exposed via dax device can be accessed via userland software via the `mmap()` system-call. The result is direct mappings to the CXL capacity in the task's page tables.

Users wishing to manually handle allocation of CXL memory should use this interface.

kmem conversion

The `dax_kmem` driver converts a *DAX Device* into a series of *hotplug memory blocks* managed by `kernel/memory-hotplug.c`. This capacity will be exposed to the kernel page allocator in the user-selected memory zone.

The `memmap_on_memory` setting (both global and DAX device local) dictate where the kernell will allocate the `struct folio` descriptors for this memory will come from. If `memmap_on_memory` is set, memory hotplug will set aside a portion of the memory block capacity to allocate folios. If unset, the memory is allocated via a normal `GFP_KERNEL` allocation - and as a result will most likely land on the local NUM node of the cpu executing the hotplug operation.

Memory Hotplug

The final phase of surfacing CXL memory to the kernel page allocator is for the *DAX* driver to surface a *Driver Managed* memory region via the memory-hotplug component.

There are four major configurations to consider

1. Default Online Behavior (on/off and zone)
2. Hotplug Memory Block size
3. Memory Map Resource location
4. Driver-Managed Memory Designation

Default Online Behavior

The default-online behavior of hotplug memory is dictated by the following, in order of precedence:

- `CONFIG_MHP_DEFAULT_ONLINE_TYPE` Build Configuration
- `memhp_default_state` Boot parameters
- `/sys/devices/system/memory/auto_online_blocks` value

These dictate whether hotplugged memory blocks arrive in one of three states:

1. Offline
2. Online in `ZONE_NORMAL`
3. Online in `ZONE_MOVABLE`

`ZONE_NORMAL` implies this capacity may be used for almost any allocation, while `ZONE_MOVABLE` implies this capacity should only be used for migratable allocations.

`ZONE_MOVABLE` attempts to retain the hotplug-ability of a memory block so that it the entire region may be hot-unplugged at a later time. Any capacity onlined into `ZONE_NORMAL` should be considered permanently attached to the page allocator.

Hotplug Memory Block Size

By default, on most architectures, the Hotplug Memory Block Size is either 128MB or 256MB. On x86, the block size increases up to 2GB as total memory capacity exceeds 64GB. As of v6.15, Linux does not take into account the size and alignment of the ACPI CEDT CFMWS regions (see Early Boot docs) when deciding the Hotplug Memory Block Size.

Memory Map

The location of `struct folio` allocations to represent the hotplugged memory capacity are dicated by the following system settings:

- `/sys_module/memory_hotplug/parameters/memmap_on_memory`
- `/sys/bus/dax/devices/daxN.Y/memmap_on_memory`

If both of these parameters are set to true, `struct folio` for this capacity will be carved out of the memory block being onlined. This has performance implications if the memory is particularly high-latency and its `struct folio` becomes hotly contended.

If either parameter is set to false, `struct folio` for this capacity will be allocated from the local node of the processor running the hotplug procedure. This capacity will be allocated from `ZONE_NORMAL` on that node, as it is a `GFP_KERNEL` allocation.

Systems with extremely large amounts of `ZONE_MOVABLE` memory (e.g. CXL memory pools) must ensure that there is sufficient local `ZONE_NORMAL` capacity to host the memory map for the hotplugged capacity.

Driver Managed Memory

The DAX driver surfaces this memory to memory-hotplug as “Driver Managed”. This is not a configurable setting, but it’s important to not that driver managed memory is explicitly excluded from use during kexec. This is required to ensure any reset or out-of-band operations that the CXL device may be subject to during a functional system-reboot (such as a reset-on-probe) will not cause portions of the kexec kernel to be overwritten.

DAX Devices

CXL capacity exposed as a DAX device can be accessed directly via mmap. Users may wish to use this interface mechanism to write their own userland CXL allocator, or to managed shared or persistent memory regions across multiple hosts.

If the capacity is shared across hosts or persistent, appropriate flushing mechanisms must be employed unless the region supports Snoop Back-Invalidate.

Note that mappings must be aligned (size and base) to the dax device's base alignment, which is typically 2MB - but maybe be configured larger.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

#define DEVICE_PATH "/dev/dax0.0" // Replace DAX device path
#define DEVICE_SIZE (4ULL * 1024 * 1024 * 1024) // 4GB

int main() {
    int fd;
    void* mapped_addr;

    /* Open the DAX device */
    fd = open(DEVICE_PATH, O_RDWR);
    if (fd < 0) {
        perror("open");
        return -1;
    }

    /* Map the device into memory */
    mapped_addr = mmap(NULL, DEVICE_SIZE, PROT_READ | PROT_WRITE,
                       MAP_SHARED, fd, 0);
    if (mapped_addr == MAP_FAILED) {
        perror("mmap");
        close(fd);
        return -1;
    }

    printf("Mapped address: %p\n", mapped_addr);

    /* You can now access the device through the mapped address */
    uint64_t* ptr = (uint64_t*)mapped_addr;
    *ptr = 0x1234567890abcdef; // Write a value to the device
    printf("Value at address %p: 0x%016llx\n", ptr, *ptr);

    /* Clean up */
    munmap(mapped_addr, DEVICE_SIZE);
    close(fd);
    return 0;
}
```

The Page Allocator

The kernel page allocator services all general page allocation requests, such as `kmalloc`. CXL configuration steps affect the behavior of the page allocator based on the selected *Memory Zone* and *NUMA node* the capacity is placed in.

This section mostly focuses on how these configurations affect the page allocator (as of Linux v6.15) rather than the overall page allocator behavior.

NUMA nodes and mempolicy

Unless a task explicitly registers a mempolicy, the default memory policy of the linux kernel is to allocate memory from the *local NUMA node* first, and fall back to other nodes only if the local node is pressured.

Generally, we expect to see local DRAM and CXL memory on separate NUMA nodes, with the CXL memory being non-local. Technically, however, it is possible for a compute node to have no local DRAM, and for CXL memory to be the *local* capacity for that compute node.

Memory Zones

CXL capacity may be onlined in `ZONE_NORMAL` or `ZONE_MOVABLE`.

As of v6.15, the page allocator attempts to allocate from the highest available and compatible ZONE for an allocation from the local node first.

An example of a *zone incompatibility* is attempting to service an allocation marked `GFP_KERNEL` from `ZONE_MOVABLE`. Kernel allocations are typically not migratable, and as a result can only be serviced from `ZONE_NORMAL` or lower.

To simplify this, the page allocator will prefer `ZONE_MOVABLE` over `ZONE_NORMAL` by default, but if `ZONE_MOVABLE` is depleted, it will fallback to allocate from `ZONE_NORMAL`.

Zone and Node Quirks

Lets consider a configuration where the local DRAM capacity is largely onlined into `ZONE_NORMAL`, with no `ZONE_MOVABLE` capacity present. The CXL capacity has the opposite configuration - all onlined in `ZONE_MOVABLE`.

Under the default allocation policy, the page allocator will completely skip `ZONE_MOVABLE` has a valid allocation target. This is because, as of Linux v6.15, the page allocator does approximately the following:

```
for (each zone in local_node):

  for (each node in fallback_order):

    attempt_allocation(gfp_flags);
```

Because the local node does not have `ZONE_MOVABLE`, the CXL node is functionally unreachable for direct allocation. As a result, the only way for CXL capacity to be used is via *demotion* in the reclaim path.

This configuration also means that if the DRAM ndoe has `ZONE_MOVABLE` capacity - when that capacity is depleted, the page allocator will actually prefer CXL `ZONE_MOVABLE` pages over DRAM `ZONE_NORMAL` pages.

We may wish to invert these configurations in future Linux versions.

If *demotion* and *swap* are disabled, Linux will begin to cause OOM crashes when the DRAM nodes are depleted. This will be covered amore in depth in the reclaim section.

CGroups and CPUSets

Finally, assuming CXL memory is reachable via the page allocation (i.e. onlined in `ZONE_NORMAL`), the `cpusets.mems_allowed` may be used by containers to limit the accessibility of certain NUMA nodes for tasks in that container. Users may wish to utilize this in multi-tenant systems where some tasks prefer not to use slower memory.

In the reclaim section we'll discuss some limitations of this interface to prevent demotions of shared data to CXL memory (if demotions are enabled).

Reclaim

Another way CXL memory can be utilized *indirectly* is via the reclaim system in `mm/vmscan.c`. Reclaim is engaged when memory capacity on the system becomes pressured based on global and cgroup-local *watermark* settings.

In this section we won't discuss the *watermark* configurations, just how CXL memory can be consumed by various pieces of reclaim system.

Demotion

By default, the reclaim system will prefer swap (or zswap) when reclaiming memory. Enabling `kernel/mm/numa/demotion_enabled` will cause vmscan to opportunistically prefer distant NUMA nodes to swap or zswap, if capacity is available.

Demotion engages the `mm/memory_tier.c` component to determine the next demotion node. The next demotion node is based on the `HMAT` or `CDAT` performance data.

cpusets.mems_allowed quirk

In Linux v6.15 and below, demotion does not respect `cpusets.mems_allowed` when migrating pages. As a result, if demotion is enabled, vmscan cannot guarantee isolation of a container's memory from nodes not set in mems_allowed.

In Linux v6.XX and up, demotion does attempt to respect `cpusets.mems_allowed`; however, certain classes of shared memory originally instantiated by another cgroup (such as common libraries - e.g. libc) may still be demoted. As a result, the mems_allowed interface still cannot provide perfect isolation from the remote nodes.

ZSwap and Node Preference

In Linux v6.15 and below, ZSwap allocates memory from the local node of the processor for the new pages being compressed. Since pages being compressed are typically cold, the result is a cold page becomes promoted - only to be later demoted as it ages off the LRU.

In Linux v6.XX, ZSwap tries to prefer the node of the page being compressed as the allocation target for the compression page. This helps prevernt thrashing.

Demotion with ZSwap

When enabling both Demotion and ZSwap, you create a situation where ZSwap will prefer the slowest form of CXL memory by default until that tier of memory is exausted.

Huge Pages

Contiguous Memory Allocator

CXL Memory onlined as SystemRAM during early boot is eligible for use by CMA, as the NUMA node hosting that capacity will be *Online* at the time CMA carves out contiguous capacity.

CXL Memory deferred to the CXL Driver for configuration cannot have its capacity allocated by CMA - as the NUMA node hosting the capacity is *Offline* at `__init` time - which CMA carves out contiguous capacity.

HugeTLB

2MB Huge Pages

All CXL capacity regardless of configuration time or memory zone is eligible for use as 2MB huge pages.

1GB Huge Pages

CXL capacity onlined in `ZONE_NORMAL` is eligible for 1GB Gigantic Page allocation.

CXL capacity onlined in `ZONE_MOVABLE` is not eligible for 1GB Gigantic Page allocation.

Memory Tiering

todo

Memory Tiers

todo

Transparent Page Placement

todo

Data Access MONitor

to be updated

References

- Self-tuned Memory Tiering RFC prototype and its evaluation
- SK Hynix HMSDK Capacity Expansion
- kernel documentation
- project website

Memory Expansion

todo

As Page Cache

todo

As DAX Device

todo

Dynamic Capacity

todo

For Virtual Machines

todo

For Workload Orchestration

todo

For Shared Memory

todo

Virtual Machines

todo

NUMA Passthrough

todo

Flexible Shapes

todo

Datacenter Efficiency

todo

Shared Memory

todo

Coherence

todo

Fabric Attached Memory FileSystem (FAMFS)

todo

Contributors

- Gregory Price
- Joshua Hahn