Coherent Accelerator Interface (CXL)

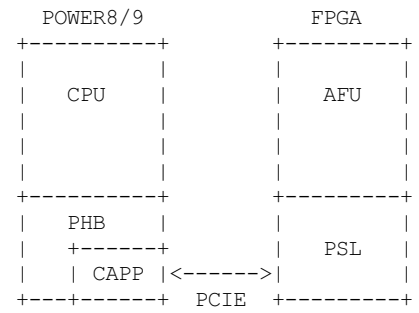
Introduction

The coherent accelerator interface is designed to allow the coherent connection of accelerators (FPGAs and other devices) to a POWER system. These devices need to adhere to the Coherent Accelerator Interface Architecture (CAIA).

IBM refers to this as the Coherent Accelerator Processor Interface or CAPI. In the kernel it's referred to by the name CXL to avoid confusion with the ISDN CAPI subsystem.

Coherent in this context means that the accelerator and CPUs can both access system memory directly and with the same effective addresses.

Hardware overview



The POWER8/9 chip has a Coherently Attached Processor Proxy (CAPP) unit which is part of the PCIe Host Bridge (PHB). This is managed by Linux by calls into OPAL. Linux doesn't directly program the CAPP.

The FPGA (or coherently attached device) consists of two parts. The POWER Service Layer (PSL) and the Accelerator Function Unit (AFU). The AFU is used to implement specific functionality behind the PSL. The PSL, among other things, provides memory address translation services to allow each AFU direct access to userspace memory.

The AFU is the core part of the accelerator (eg. the compression, crypto etc function). The kernel has no knowledge of the function of the AFU. Only userspace interacts directly with the AFU.

The PSL provides the translation and interrupt services that the AFU needs. This is what the kernel interacts with. For example, if the AFU needs to read a particular effective address, it sends that address to the PSL, the PSL then translates it, fetches the data from memory and returns it to the AFU. If the PSL has a translation miss, it interrupts the kernel and the kernel services the fault. The context to which this fault is serviced is based on who owns that acceleration function.

- POWER8 and PSL Version 8 are compliant to the CAIA Version 1.0.
- POWER9 and PSL Version 9 are compliant to the CAIA Version 2.0.

This PSL Version 9 provides new features such as:

- Interaction with the nest MMU on the P9 chip.
- Native DMA support.
- Supports sending ASB_Notify messages for host thread wakeup.
- Supports Atomic operations.
- etc.

Cards with a PSL9 won't work on a POWER8 system and cards with a PSL8 won't work on a POWER9 system.

AFU Modes

There are two programming modes supported by the AFU. Dedicated and AFU directed. AFU may support one or both modes.

When using dedicated mode only one MMU context is supported. In this mode, only one userspace process can use the accelerator at time.

When using AFU directed mode, up to 16K simultaneous contexts can be supported. This means up to 16K simultaneous userspace applications may use the accelerator (although specific AFUs may support fewer). In this mode, the AFU sends a 16 bit context ID with each of its requests. This tells the PSL which context is associated with each operation. If the PSL can't translate an operation, the ID can also be accessed by the kernel so it can determine the userspace context associated with an operation.

MMIO space

A portion of the accelerator MMIO space can be directly mapped from the AFU to userspace. Either the whole space can be mapped or just a per context portion. The hardware is self describing, hence the kernel can determine the offset and size of the per context portion.

Interrupts

AFUs may generate interrupts that are destined for userspace. These are received by the kernel as hardware interrupts and passed onto userspace by a read syscall documented below.

Data storage faults and error interrupts are handled by the kernel driver.

Work Element Descriptor (WED)

The WED is a 64-bit parameter passed to the AFU when a context is started. Its format is up to the AFU hence the kernel has no knowledge of what it represents. Typically it will be the effective address of a work queue or status block where the AFU and userspace can share control and status information.

User API

1. AFU character devices

For AFUs operating in AFU directed mode, two character device files will be created. /dev/cxl/afu0.0m will correspond to a master context and /dev/cxl/afu0.0s will correspond to a slave context. Master contexts have access to the full MMIO space an AFU provides. Slave contexts have access to only the per process MMIO space an AFU provides.

For AFUs operating in dedicated process mode, the driver will only create a single character device per AFU called /dev/cxl/afu0.0d. This will have access to the entire MMIO space that the AFU provides (like master contexts in AFU directed).

The types described below are defined in include/uapi/misc/cxl.h

The following file operations are supported on both slave and master devices.

A userspace library libcxl is available here:

https://github.com/ibm-capi/libcxl

This provides a C interface to this kernel API.

open

Opens the device and allocates a file descriptor to be used with the rest of the API.

A dedicated mode AFU only has one context and only allows the device to be opened once.

An AFU directed mode AFU can have many contexts, the device can be opened once for each context that is available.

When all available contexts are allocated the open call will fail and return -ENOSPC.

Note:

> IRQs need to be allocated for each context, which may limit the number of contexts that can be created, and therefore how many times the device can be opened. The POWER8 CAPP supports 2040 IRQs and 3 are used by the kernel, so 2037 are left. If 1 IRQ is needed per context, then only 2037 contexts can be allocated. If 4 IRQs are needed per context, then only 2037/4 = 509 contexts can be allocated.

ioctl

CXL_IOCTL_START_WORK:

> Starts the AFU context and associates it with the current process. Once this ioctl is successfully executed, all memory mapped into this process is accessible to this AFU context using the same effective addresses. No additional calls are required to map/unmap memory. The AFU memory context will be updated as userspace allocates and frees memory. This ioctl returns once the AFU context is started.
>
> Takes a pointer to a struct cxl_ioctl_start_work

```
struct cxl_ioctl_start_work {
        __u64 flags;
        __u64 work_element_descriptor;
        __u64 amr;
        __s16 num_interrupts;
        __s16 reserved1;
        __s32 reserved2;
```

```
                __u64 reserved3;
                __u64 reserved4;
                __u64 reserved5;
                __u64 reserved6;
        };
```

flags:
: Indicates which optional fields in the structure are valid.

work_element_descriptor:
: The Work Element Descriptor (WED) is a 64-bit argument defined by the AFU. Typically this is an effective address pointing to an AFU specific structure describing what work to perform.

amr:
: Authority Mask Register (AMR), same as the powerpc AMR. This field is only used by the kernel when the corresponding CXL_START_WORK_AMR value is specified in flags. If not specified the kernel will use a default value of 0.

num_interrupts:
: Number of userspace interrupts to request. This field is only used by the kernel when the corresponding CXL_START_WORK_NUM_IRQS value is specified in flags. If not specified the minimum number required by the AFU will be allocated. The min and max number can be obtained from sysfs.

reserved fields:
: For ABI padding and future extensions

CXL_IOCTL_GET_PROCESS_ELEMENT:
: Get the current context id, also known as the process element. The value is returned from the kernel as a __u32.

mmap

An AFU may have an MMIO space to facilitate communication with the AFU. If it does, the MMIO space can be accessed via mmap. The size and contents of this area are specific to the particular AFU. The size can be discovered via sysfs.

In AFU directed mode, master contexts are allowed to map all of the MMIO space and slave contexts are allowed to only map the per process MMIO space associated with the context. In dedicated process mode the entire MMIO space can always be mapped.

This mmap call must be done after the START_WORK ioctl.

Care should be taken when accessing MMIO space. Only 32 and 64-bit accesses are supported by POWER8. Also, the AFU will be designed with a specific endianness, so all MMIO accesses should consider endianness (recommend endian(3) variants like: le64toh(), be64toh() etc). These endian issues equally apply to shared memory queues the WED may describe.

read

Reads events from the AFU. Blocks if no events are pending (unless O_NONBLOCK is supplied). Returns -EIO in the case of an unrecoverable error or if the card is removed.

read() will always return an integral number of events.

The buffer passed to read() must be at least 4K bytes.

The result of the read will be a buffer of one or more events, each event is of type struct cxl_event, of varying size:

```
struct cxl_event {
        struct cxl_event_header header;
        union {
                struct cxl_event_afu_interrupt irq;
                struct cxl_event_data_storage fault;
                struct cxl_event_afu_error afu_error;
        };
};
```

The struct cxl_event_header is defined as

```
    struct cxl_event_header {
            __u16 type;
            __u16 size;
            __u16 process_element;
            __u16 reserved1;
    };
```

type:

This defines the type of event. The type determines how the rest of the event is structured. These types are described below and defined by enum cxl_event_type.

size:
This is the size of the event in bytes including the struct cxl_event_header. The start of the next event can be found at this offset from the start of the current event.

process_element:
Context ID of the event.

reserved field:
For future extensions and padding.

If the event type is CXL_EVENT_AFU_INTERRUPT then the event structure is defined as

```
struct cxl_event_afu_interrupt {
        __u16 flags;
        __u16 irq; /* Raised AFU interrupt number */
        __u32 reserved1;
};
```

flags:
These flags indicate which optional fields are present in this struct. Currently all fields are mandatory.

irq:
The IRQ number sent by the AFU.

reserved field:
For future extensions and padding.

If the event type is CXL_EVENT_DATA_STORAGE then the event structure is defined as

```
struct cxl_event_data_storage {
        __u16 flags;
        __u16 reserved1;
        __u32 reserved2;
        __u64 addr;
        __u64 dsisr;
        __u64 reserved3;
};
```

flags:
These flags indicate which optional fields are present in this struct. Currently all fields are mandatory.

address:
The address that the AFU unsuccessfully attempted to access. Valid accesses will be handled transparently by the kernel but invalid accesses will generate this event.

dsisr:
This field gives information on the type of fault. It is a copy of the DSISR from the PSL hardware when the address fault occurred. The form of the DSISR is as defined in the CAIA.

reserved fields:
For future extensions

If the event type is CXL_EVENT_AFU_ERROR then the event structure is defined as

```
struct cxl_event_afu_error {
        __u16 flags;
        __u16 reserved1;
        __u32 reserved2;
        __u64 error;
};
```

flags:
These flags indicate which optional fields are present in this struct. Currently all fields are Mandatory.

error:
Error status from the AFU. Defined by the AFU.

reserved fields:
For future extensions and padding

2. Card character device (powerVM guest only)

In a powerVM guest, an extra character device is created for the card. The device is only used to write (flash) a new image on the FPGA accelerator. Once the image is written and verified, the device tree is updated and the card is reset to reload the updated image.

open

Opens the device and allocates a file descriptor to be used with the rest of the API. The device can only be opened once.

ioctl

CXL_IOCTL_DOWNLOAD_IMAGE / CXL_IOCTL_VALIDATE_IMAGE:

Starts and controls flashing a new FPGA image. Partial reconfiguration is not supported (yet), so the image must contain a copy of the PSL and AFU(s). Since an image can be quite large, the caller may have to iterate, splitting the image in smaller chunks.

Takes a pointer to a struct cxl_adapter_image:

```
struct cxl_adapter_image {
    __u64 flags;
    __u64 data;
    __u64 len_data;
    __u64 len_image;
    __u64 reserved1;
    __u64 reserved2;
    __u64 reserved3;
    __u64 reserved4;
};
```

flags:
: These flags indicate which optional fields are present in this struct. Currently all fields are mandatory.

data:
: Pointer to a buffer with part of the image to write to the card.

len_data:
: Size of the buffer pointed to by data.

len_image:
: Full size of the image.

Sysfs Class

A cxl sysfs class is added under /sys/class/cxl to facilitate enumeration and tuning of the accelerators. Its layout is described in Documentation/ABI/testing/sysfs-class-cxl

Udev rules

The following udev rules could be used to create a symlink to the most logical chardev to use in any programming mode (afuX.Yd for dedicated, afuX.Ys for afu directed), since the API is virtually identical for each:

```
SUBSYSTEM=="cxl", ATTRS{mode}=="dedicated_process", SYMLINK="cxl/%b"
SUBSYSTEM=="cxl", ATTRS{mode}=="afu_directed", \
                  KERNEL=="afu[0-9]*.[0-9]*s", SYMLINK="cxl/%b"
```