EMUCXL: an emulation and access library for CXL

Submitted By: Raja Gond (190050096)
under the guidance of Prof. Purushottam Kulkarni
Department of Computer Science and Engineering
Indian Institute of Technology Bombay, Mumbai, India
rajagond@cse.iitb.ac.in

***Abstract*—The emergence of CXL in the interconnects market holds great promise for transforming the architecture of host-device interconnects. With its low overhead, low latency, and memory coherency capabilities, CXL has the potential to improve the performance of existing devices while opening up new use cases that were previously unattainable. This report outlines a comprehensive approach to emulating CXL devices, along with a library design that provides user applications with a user-friendly interface to access CXL devices based on research papers [2], [8].**

***Index Terms*—Interconnect, Memory Pooling, Numa Node, Cache Coherency, PCI Express, QEMU, Emulation, Kernel**

I. Introduction

This is an era of data. As we are becoming advanced in technology, data is growing at a very faster rate. Every industry collects lots of data from the user and processes it for various applications. With the growth of data-intensive application workloads and dedicated devices to compute on the data, we need an interconnect that can move data between these devices with low overhead, low latency and high bandwidth.

New cache coherent interconnects such as Compute Express Link (CXL) have recently attracted great attention thanks to their excellent hardware heterogeneity management and resource disaggregation capabilities. Even though there is yet no real product or platform integrating CXL into memory disaggregation, it is expected to make memory resources practically and efficiently disaggregated much better than ever before [8].

One issue that may arise with the availability of hardware support for CXL is how user applications will be able to access CXL devices, such as loading or storing data on them. This is also a concern when considering emulated CXL devices in the absence of actual hardware. In essence, the challenge is to ensure that user applications can interface with CXL devices effectively and efficiently.

As part of this work, we have designed a library that enables user applications to perform load and store operations on emulated CXL devices. Our library is based on the concept presented in the paper titled *Direct Access, High-Performance Memory Disaggregation with DirectCXL*([8]). The emulation of CXL devices posed a significant challenge, and we have discussed various methods to emulate CXL devices in Section V. It is important to note that CXL devices are classified into three types based on their functionality, and we have discussed each of these types in detail in Subsection III-E.

The rest of the report is organized as follows, starting with the background and motivation in Section II, followed by a detailed discussion about the CXL in Section III and evolving use cases of cxl in Section IV. Emulation of cxl devices is briefly discussed in Section V and the design of EmuCxl API is presented in Section VI. Discussion about this work and challenges is in Section VII. Future extension of emucxl is discussed in Section VIII and Section IX concludes followed by acknowledgements.

II. Background & Motivation

CXL is emerging as the industry focal point for coherent IO and it is collaborating with industry organizations. CXL has successfully rallied the entire industry behind a common coherent interconnect standard with a common vision of how computing should evolve going forward. All commercial server systems are planning to deploy CXL-based solutions [6].

Following are the challenges, the computing world is facing currently and CXL promises to mitigate these problems [1]:

- Industry trends driving demand for faster data processing and next-gen data center performance
- Increasing demand for heterogeneous computing and server disaggregation
- Need for increased memory capacity and bandwidth
- Lack of open industry standard to address next-gen interconnect challenges

A. Heterogeneous and disaggregated computing

1) What is heterogeneous computing?: A heterogeneous computing platform is a system that combines different types of processors or computing elements, such as CPUs, GPUs, FPGAs, and ASICs, to perform specific tasks more efficiently. Heterogeneous computing platforms leverage the strengths of each type of processor to achieve higher performance and energy efficiency than a homogeneous system based on a single type of processor.

2) What is disaggregated computing?: Disaggregated computing is a computing architecture that separates compute, memory, and storage resources into distinct physical devices connected by a high-speed network.

CXL is an open standard industry-supported cache-coherent **interconnect** for processors, memory expansion, and accelerators with **PCIe** as the physical layer. Let us understand what is interconnect and PCI Express.

B. What is an interconnect?

In computing, an interconnect is a pathway that allows different components within a computer system to communicate with each other. It provides a mechanism for data, control signals, and power to flow between different devices such as processors, memory modules, storage devices, and input/output interfaces. Interconnects can be implemented in various ways, including buses, point-to-point links, switches, and networks. The design of an interconnect system plays a critical role in determining the performance, scalability, and reliability of a computer system. High-performance computing systems, data centers, and other complex computing systems require interconnects that can support high bandwidth, low latency, and efficient use of resources. Interconnect technology has evolved rapidly in recent years to meet the demands of modern computing, with innovations such as PCIe, InfiniBand, Ethernet, and other technologies.

We are mostly interested in the interconnection between process and memory.

C. Introduction to PCIe: Its characteristics and interface provided by PCIe

PCIe (Peripheral Component Interconnect Express) is a high-speed serial computer expansion bus standard that provides a high-bandwidth data transfer pathway between the central processing unit (CPU) and the peripheral devices in a computer system. PCIe is a successor to the older PCI and AGP bus standards and it is the leading physical interconnect used currently.

PCIe uses a point-to-point topology, meaning that each peripheral device has a dedicated connection to the CPU. This results in a faster and more efficient data transfer as compared to shared buses like PCI, where multiple devices share a single connection to the CPU. PCIe also allows for the simultaneous transfer of data in both directions, making it a full-duplex technology.It provides high bandwidth, reaching up to 128 GB/c in a single direction. In PCIe, each device has a dedicated connection between the host and the device. It has separate lanes for the downward and upward direction of data flow. Typically DMA is used in conjunction with PCIe to move data to and from the device.

1) Topology of PCI Express: The PCI Express topology, illustrated in Figure 1, comprises three main components: the Root Complex, Switch, and Endpoint. The Root Complex serves as the foundation of an I/O hierarchy, linking the CPU/memory subsystem to the I/O. Each port is connected to either an endpoint device or a switch, creating a sub-hierarchy. A Switch offers fan-out capability and facilitates a series of connectors for high-performance I/O add-ins. Essentially, a Switch is made up of two or more logical PCI-to-PCI bridges, each of which is associated with a switch port. One port of the Switch, facing towards the root complex, is an upstream port, while all other ports facing away from the root complex are downstream ports. An Endpoint is an I/O device that connects to the PCI Express, such as a graphics, network, or storage controller attached to the PCI Express. Endpoints differ from switches in that they are requesters or completers of PCI Express transactions.

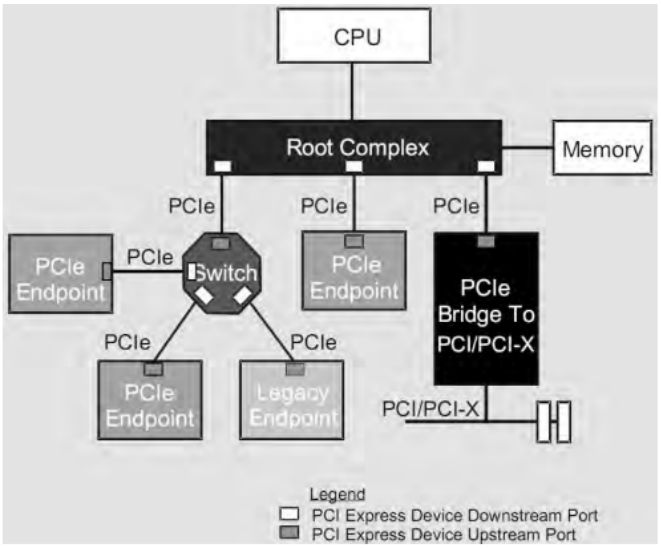


Fig. 1: PCIe System Topology [9]

D. Motivation for EMUCXL

Commercial CXL hardware is not available in the market at the moment, it is expected to become available in the near future, enabling broader adoption of the CXL standard. However, several research papers such as [11], [8], [14] have described in-house prototypes of CXL hardware and such as [10], [2], [12], [13] software-based emulations of CXL devices.

After enabling support for CXL-enabled devices on the processor, user applications may require access to CXL memory. This need extends even to the present day, where there is a desire to access memory on emulated CXL devices. In order to facilitate this access, it is user-friendly to provide interfaces that allow user applications to access memory on CXL-enabled devices. This is the primary objective of EMUCXL , which is focused on providing support for emulated CXL devices at present.

III. Compute Express Link

Compute Express Link([1]) is an open standard new class of **interconnect** for device connectivity and cache coherent interface using PCIe as the physical layer. It promises to provide high-bandwidth, low-latency communication between the host and connected devices, e.g., accelerators, as well as other devices like memory and smart I/O devices. The design and specifications of CXL are released by a **CXL consortium** that consists of 150+ member companies including all major CPU, GPU and memory vendors. CXL has a bright future and will be a game-changer in the industry!!

A. Compute Express Link sounds like it might be a big deal, but what does it provide us exactly?

CXL provides the following features:

- CXL supports dynamic multiplexing of I/O (`CXL.io`), coherency (`CXL.cache`), and memory (`CXL.memory`) protocols [6]
- CXL maintains a unified, coherent memory space between the CPU(s) (host processor) and memory attached to CXL device(s). This allows both the CPU and CXL device to share memory resources coherently for higher performance [6]
- It promises to provide DRAM like latency and High Bandwidth reaching up to `128 GB/s` in each direction

B. How does CXL leverage PCIe?

CXL dynamically multiplexes its protocols on the physical PCIe connection. It defines a custom data link protocol to reduce connection overhead and reduce latency. CXL can offer one or more memory address spaces in the PCIe network domain coherently, which can consistently be accessed by different processors and hardware accelerators over its multi-protocol technology. CXL's multi-protocol can integrate PCIe storage into its cache coherent memory space, it can create a much bigger memory pool than DRAM-based or PMEM-based memory expansion technologies [9].

C. Overview of CXL standards

There are three standards released by the CXL consortium.

- **CXL 1.0/1.1**: It runs on PCIe 5.0 infrastructure makes it really easy for devices and platforms to adopt CXL without having to design and validate the PHY, channel, any channel extension devices such as Retimers, or the upper layers of PCIe, including the software stack and supports x16, x8, and x4 link widths natively and x2 and x1 widths in degraded mode. CXL 1.0 debuted at 32 GT/s, offering 64 GB/s bandwidth in each direction. CXL 1.0 also supports 16.0 GT/s and 8.0 GT/s data rates in degraded mode. It allowed device memory of accelerators to be mapped to the host. CXL 1.1 included the compliance testing details.
- **CXL 2.0**: CXL 2.0 defines Switches that can be used to connect multiple CXL devices to a host. It also allows a device to be used by multiple hosts. CXL 2.0 maintains full backward compatibility with CXL 1.1 and CXL 1.0 and enhances the CXL 1.1 experience by introducing three major areas: CXL Switch, support for persistent memory, memory pooling, and security. These features enable many devices in a platform to migrate to CXL, while maintaining compatibility with PCIe 5.0 and the low-latency characteristics of CXL.
- **CXL 3.0**: CXL 3.0 [5] is based on PCIe 6.0 technology, doubles the transfer rate to 64GT/s with no additional latency over previous generations. This allows for aggregate raw bandwidth of up to 256GB/s for x16 width link. For low-latency transfers, CXL 3.0 leverages PCIe 6.0's combination of lightweight FEC and strong CRC for error free transmission with 256B flits on PAM-4 signaling to achieve 64GT/s.

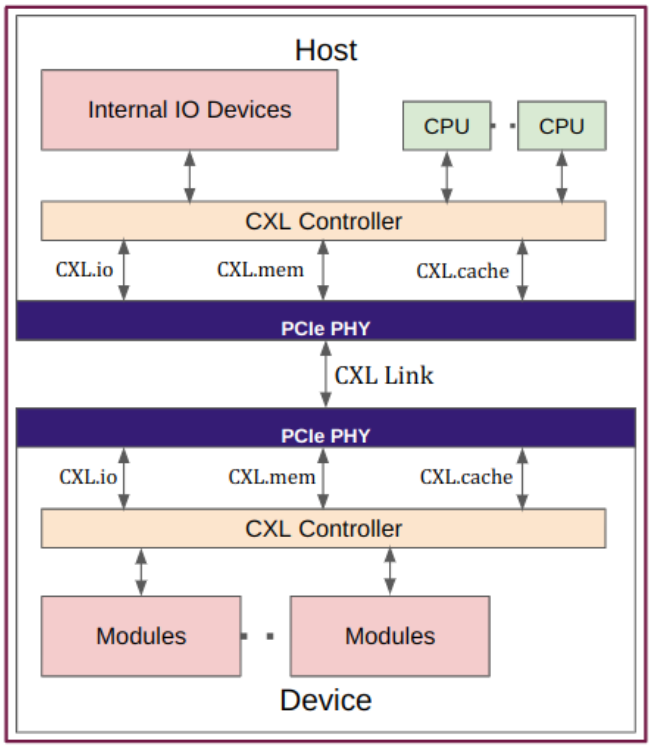


Fig. 2: CXL Architecture

D. Protocols defined by CXL standard

CXL defines three protocols that provide different functionalities.

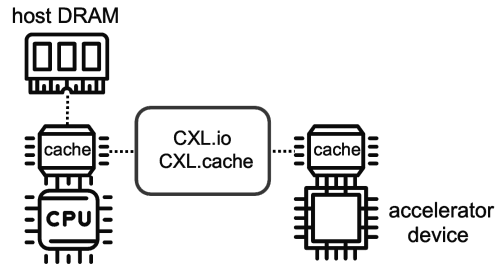
- **CXL.io**: This is a mandatory protocol that is used to set up the device connection with the host. It provides device discovery, device configuration and reconfiguration functionality.
- **CXL.cache**: This is an optional protocol that enables a device to cache host memory. The caching mechanism is provided by the host caching engine.
- **CXL.mem**: This is also an optional protocol that is used by a host to map a device memory in its own address space. Doing so, it can issue direct memory load/store instructions to access and modify device memory while ensuring memory coherency.

Figure 2 depicts a simplified CXL architecture that incorporates a variety of protocols.

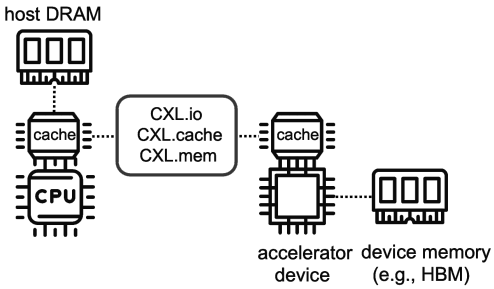
E. Types of CXL devices

Using three protocols defined in CXL standard, CXL consortium defines three types of CXL devices to provide different functionalities.

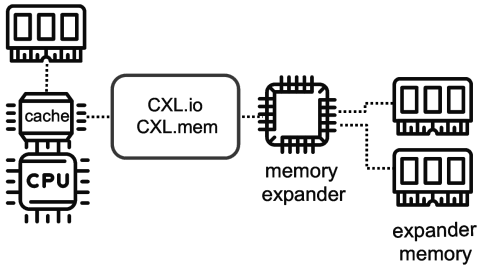
- **Type 1**(Figure 3a): It can access the host memory through `CXL.cache` transactions and maintain a local cache that is coherent with the host memory. It uses `CXL.io` and `CXL.cache` protocol. This class of devices accesses the unified memory, but they do not have large memories of their own. They are caching devices such as Accelerators



(a) Type-1 device(e.g. SmartNIC)



(b) Type-2 device(e.g. GPU)
HBM - High Bandwidth Memory



(c) Type-3 device(e.g. Memory Pool)

Fig. 3: Types of CXL devices [3]

and SmartNICs. SmartNIC needs to access host memory frequently to provide Intrusion Detection.

- **Type 2**(Figure 3b): The class of devices, accelerators with memory, is the category that includes GPU and FPGA accelerators. They use all three protocols, `CXL.io`, `CXL.cache`, and `CXL.memory` and both provide local memory to the unified memory space and access the unified space through a cache. Additionally, CXL Type 2 devices have local address space that is visible and accessible to the host CPU through `CXL.mem` transactions.
- **Type 3**(Figure 3c): These devices support only `CXL.io` and `CXL.mem`, and are targeted towards memory capacity and memory bandwidth expansion use-cases. It only have device memory that the host can map in its address space coherently. It provides memory for the unified memory space, but they do not use the unified memory. CXL memory devices like this fill the gap between DRAM and performance SSD and provide an easy interface to connect new emergent memory technologies [4]. A typical application of type 3 is memory expansion.

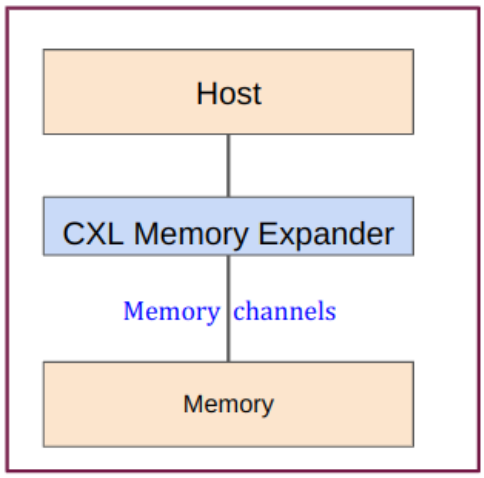


Fig. 4: Memory Expansion

IV. CXL EVOLVING USE CASES

CXL technology's cache coherence and low latency characteristics offer significant advantages for a broad range of use cases. Type-1 devices, for example, can provide faster and more efficient memory access, making them ideal for operations that require frequent access to host memory. Type-2 devices, on the other hand, can simplify the interface between the host and accelerators, enabling more efficient data transfer.

Type-3 devices enable the dis-aggregation of CPU and memory, allowing for more efficient use of resources. Instead of deploying traditional servers with fixed amounts of CPU and memory resources, Type-3 devices allow CPU and memory to be scaled independently, providing the necessary resources as needed. This can result in significant cost savings and more efficient resource utilization in large-scale server deployments.

CXL technology also presents a promising solution for use cases such as large-scale in-memory databases, real-time analytics, and computationally intensive applications. It enables systems to work on significantly larger data pools while supporting tiering to better balance power, performance, and cost.

Following are the some of evolving use cases of cxl.

A. Memory Expander

In today's world, data sizes are increasing very rapidly. Limited memory in an in-memory can become a significant performance bottleneck. In - Memory databases(such as Redis) can be benefited from the CXL memory Expander(Figure 4).

Suppose, we have a host and we want to add more memory to it, to increase its capacity. CXL promises to provide increased capacity and improve bandwidth with lower server costs. One additional advantage is that we can also use different types of memory than what our host would maybe natively use.

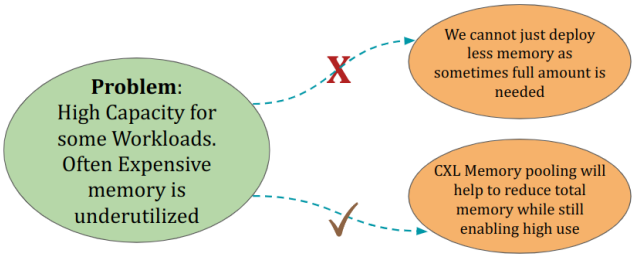


Fig. 5: Memory Pooling

B. Memory Pooling

Memory is the largest cost of data centres. Even today we are seeing cases where dram can be half of server cost. In the future, this might grow due to scaling and all. Many workloads only use a fraction of the available memory, which can result in wastage of resources and increased costs. However, for certain workloads, high memory capacity is necessary. This creates a challenge, as we can't simply reduce memory without impacting performance.

Memory pooling(Figure 5) with CXL can address this problem. CXL allows for memory to be pooled together from multiple devices. This enables high utilization while reducing the total amount of memory required. This is an effective solution for workloads that require both high capacity and efficient resource utilization.

With CXL, a server can access the memory resources of other servers connected through the interconnect, allowing for pooling of memory resources. This can be done through Type-3 CXL devices, which enable the dis-aggregation of CPU and memory resources. In this scenario, one server can be designated as a "memory pool," providing memory resources to other servers as needed.

The memory pooling process typically involves establishing a shared memory pool across multiple servers and then using CXL to access the memory resources in the pool. Applications can then request memory resources from the shared pool as needed, and the resources can be allocated dynamically based on the specific workload requirements.

C. Network Interface Cards (NICs)

CXL enables high-speed connectivity between the processor and network interface cards, allowing for low-latency and high-bandwidth networking.

D. Artificial Intelligence/Machine Learning

CXL can enable the development of more powerful and efficient AI and machine learning models by providing faster access to memory and accelerators. This can lead to improved performance and faster training times for complex AI/ML workloads.

V. Emulation of CXL devices

As CXL is a relatively new technology, there is currently no hardware available in the market for CXL devices. Emulating CXL devices has thus become essential for testing and development purposes.

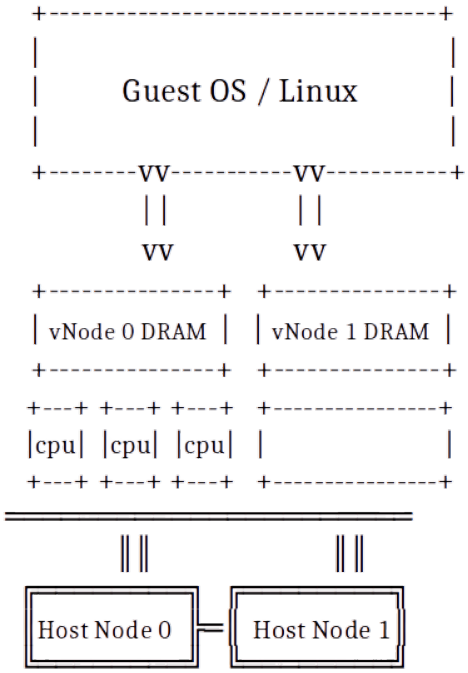


Fig. 6: CXL Emulation on regular 2-socket (2S) server systems(Pond-vm)

A. Using Qemu and KVM

$$CXL + QEMU + Linux\ Kernel = YES$$

CXL devices can be emulated with the help of Qemu. But it also requires support from the Linux kernel. The good news is that support for CXL devices is already in active development. Once support is up, we can use Qemu to create virtual machines that emulate CXL devices with all functionalities.

Steps to emulate the cxl device using Qemu and KVM can be found here. Load and Store aren't yet supported on these emulated devices.

B. CXL Emulation on regular 2-socket (2S) server systems

In a NUMA setup, the individual processors in a computing system share local memory and can work together. NUMA can be thought of as a microprocessor cluster in a box. Microsoft's pond([2]) paper provides an idea to emulate cxl devices on 2 socket server systems based on two characteristics of cxl attached dram. The first one is latency 150 ns and the second is that no local cpu can directly access this cpuless node. Fig 6 shows the design of Pond's ideas.

Basically, they have created two virtual nodes mapped with two physical nodes respectively. One has cpus while the other is cpuless. They treated memory attached with a cpuless node as cxl-memory. Pond claims that the latency that comes with this setup is close to what cxl is promising.

Steps to set up a pond virtual machine with the above design can be found here.

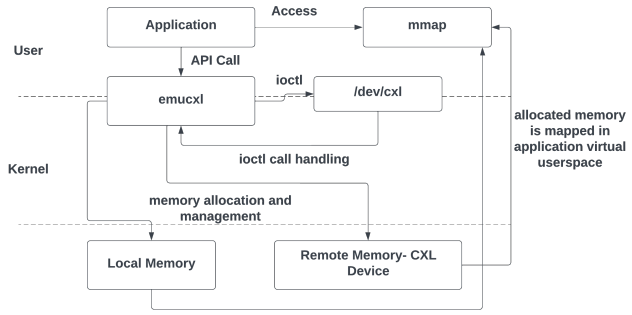


Fig. 7: Initial Design of EMUCXL

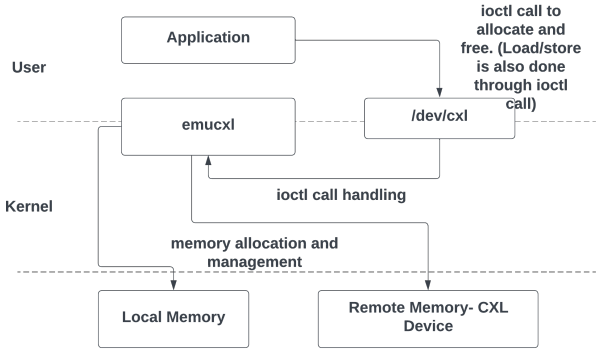


Fig. 8: Current Implementation Design of EMUCXL

VI. Design of EmuCxl

EMUCXL library is built on top of emulated cxl devices discussed in subsection V-B and the design idea is similar to what is discussed in the direct cxl paper [8]. Some of the key APIs that were planned for implementation but these APIs faced challenges discussed in section VII include:

- void emucxl_init(): This API is used to set up the device file and other configurations required for Emucxl operation.
- emucxl_exit(): This API is used to close the device file.
- void* emucxl_alloc(..., type tp, size_t size): This API is used to allocate memory either remotely or locally, depending on the type specified. The required size is also provided, and the starting address of the allocated memory is returned.
- bool emucxl_free(..., void* address, size_t size): This API is used to free allocated memory of a given size. It returns true if the operation is successful, and false otherwise.
- void* emucxl_migrate(..., void* address, type tp, size_t new): This API is used to transfer memory between local and remote locations, depending on the type specified. The required size and the starting address of the memory to be transferred are also provided. If the operation is unsuccessful, no changes are made.
- emucxl_resize(..., void* address, size_t curr, size_t new): This API is used to resize memory either in its original location or in the opposite location of the new size, if possible. If memory is allocated in the opposite location, the original memory data is copied and the memory in the original location is freed.

Figure 7 illustrates the original design of our EMUCXL library that supports all the API calls we discussed earlier. The library allows the application to call any supported API. When the application requests memory from local or remote memory (a combination of both memories like %x local and %y remote is a possible future extension) through an API, EMUCXL handles the API call and initiates the `ioctl` call to the already created device driver file, set up through the emucxl_init API. This `ioctl` request is then handled by the EMUCXL kernel module, which allocates memory on the local NUMA node or remote NUMA node, depending on the API call and `ioctl` argument, through the `kmalloc_node` or `vmalloc_node` kernel api, which is memory mapped to the user application's virtual userspace memory space. This can be done through the `remap_pfn_range` kernel api. The application can directly access the memory-mapped region, and the emucxl_free API call frees the memory mapped and allocated physical memory.

The current implementation of our EMUCXL library, as depicted in Figure 8, is a simplified version of the original design presented in Figure 7. In this version, the library does not support any API calls or memory-mapped access. Instead, the application directly sends an `ioctl` request to allocate memory on either the local or remote NUMA node, which is handled by the EMUCXL kernel module. The kernel module allocates memory using the kmalloc_node function and returns the address to the application. The application can then access this memory using load/store operations with `ioctl` calls, and when the memory is no longer needed, the application can free it using the `free ioctl` call, which uses the kfree function.

VII. Discussion and Challenges

```
vmalloc_node(size, NUMA_NODE);
// contiguous allocation
kmalloc_node(size, GFP_KERNEL, NUMA_NODE);
// contiguous allocation and
// initialise with zero
kzalloc_node(size, GFP_KERNEL, NUMA_NODE);
// initialise with zero
vzalloc_node(Asize, NUMA_NODE);
```

Above kernel API can be used for NUMA-aware memory allocation.

```
#define EMUCXL_ALLOCATE_MEMORY
    _IOWR('e', 4, emucxl_arg_t *)
#define EMUCXL_FREE_MEMORY
    _IOW('e', 5, emucxl_arg_t *)
```

The above read/write commands can be used in the `ioctl` command. copy_from_user and copy_to_user API is used to pass to and fro data.

All the codes are available at https://github.com/rajagond/pmem_cxl/tree/main/cxl_rnd2.

Designing and implementing EMUCXL presented several challenges that needed to be overcome. One challenge was managing allocation for a combination of local and remote memory. This required dealing with two starting virtual addresses, which was not a trivial task.

Another challenge was allocating memory in kernel space with kmalloc_node and then mapping it to application user space. Although remap_pfn_range was tried, it resulted in program crashes. The `mmap` API is considered as a possible solution, but no numa-aware allocation is found.

While kmalloc_node was able to allocate memory, vmalloc_node was not working in the EMUCXL setup. It did work in a separate kernel module, but not in the EMUCXL implementation. Additionally, creating and implementing the API interface was not an easy task due to the aforementioned difficulties. Building a library required a good deal of background knowledge in kernel programming, which took up a significant amount of time in the process.

VIII. Future Work

The development of EMUCXL , a library to provide an interface on emulated CXL devices, was a challenging task that required a significant amount of background research and experimentation. As CXL is a relatively new technology, much of the initial effort was spent on building familiarity with the underlying principles and exploring various methods for emulating CXL devices, as no hardware was available for testing.

In addition, the implementation of EMUCXL required a deep understanding of Linux kernel programming, including kernel memory management, ioctl calls, and NUMA node memory allocation techniques, as the design of the library was based on NUMA node allocation. Discussions were also held to determine the appropriate APIs and their implementation. Due to these constraints, the initial implementation of EMUCXL only included basic functionalities.

To further enhance EMUCXL , additional testing of all API calls discussed in the design phase is necessary. This will require mapping the memory allocated using `kmalloc_node` to userspace virtual memory and creating header files which list all the API calls and their implementation in `C` files.

An alternative approach to implementing EMUCXL can be using the `libnuma` library for all NUMA node-related operations and on top of that we can build our EMUCXL library and provide the interface. The `libnuma` library provides a simple programming interface to the NUMA policy supported by the Linux kernel, which is important for architectures with non-uniform memory access. While this approach has not been tested, it holds exploratory potential for the future development of EMUCXL .

IX. Conclusion

The emergence of CXL represents a significant technological advancement that has the potential to transform data centre architectures. With support from industry players across the board, including cloud service providers, chip makers, manufacturers of processors and IP providers, CXL's development reflects the enormous value it can offer. As a once-in-a-decade technological force, CXL's rapid adoption is expected to provide significant benefits to various use cases, from large-scale in-memory databases to real-time analytics and computationally intensive applications. By providing low latency and cache coherence, CXL can enable more efficient and cost-effective resource utilization in large-scale server deployments. With its potential to work on vastly larger pools of data while supporting tiering to better balance power, performance, and cost, CXL is poised to be a game-changer in the field of high-performance computing.

EMUCXL provides a simplified interface for user applications to get a feel of reading/writing on emulated cxl devices. The development of EMUCXL was a challenging task that required significant research, experimentation, and programming expertise. Despite its initial limitations, EMUCXL holds promise for basic read/write on emulated cxl-devices from user space and can be further enhanced with additional testing and exploration of alternative implementation approaches.

Acknowledgements

I would like to thank `Prof. Purushottam Kulkarni` for his guidance and encouragement during the course of my project. I would also like to thank `Sameer Ahmad` for the insightful discussion on Compute Express Link(CXL) technology.

References

[1] Compute Express Link Consortium. 2020. Compute Express Link Specification.

[2] Li, Huaicheng, et al. "Pond: CXL-based memory pooling systems for cloud platforms." Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. 2023.

[3] D. Boles, D. Waddington and D. A. Roberts, "CXL-Enabled Enhanced Memory Functions," in IEEE Micro, vol. 43, no. 2, pp. 58-65, 1 March-April 2023, doi: 10.1109/MM.2022.3229627.

[4] Anthony M Cabrera, Aaron R Young, and Jeffrey S Vetter. 2022. Design and analysis of CXL performance models for tightly-coupled heterogeneous computing. In Proceedings of the 1st International Workshop on Extreme Heterogeneity Solutions (ExHET '22). Association for Computing Machinery, New York, NY, USA, Article 1, 1–6. https://doi.org/10.1145/3529336.3530817

[5] Compute Express Link™ 3.0 White Paper https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf

[6] D. D. Sharma, "Compute Express Link (CXL): Enabling Heterogeneous Data-Centric Computing With Heterogeneous Memory Hierarchy," in IEEE Micro, vol. 43, no. 2, pp. 99-109, 1 March-April 2023, doi: 10.1109/MM.2022.3228561.

[7] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22). Association for Computing Machinery, New York, NY, USA, 45–51. https://doi.org/10.1145/3538643.3539745

[8] Gouk, D., Lee, S., Kwon, M. & Jung, M. Direct Access, High-Performance Memory Disaggregation with DirectCXL. *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. pp. 287-294 (2022,7), https://www.usenix.org/conference/atc22/presentation/gouk

[9] Q. Wu, J. Xu, X. Li and K. Jia, "The research and implementation of interfacing based on PCI express," 2009 9th International Conference on Electronic Measurement & Instruments, Beijing, China, 2009, pp. 3-116-3-121, doi: 10.1109/ICEMI.2009.5274345.

[10] Wahlgren, J., Gokhale, M. & Peng, I. Evaluating Emerging CXL-enabled Memory Pooling for HPC Systems. *ArXiv Preprint ArXiv:2211.02682*. (2022)

[11] Wang, C., He, K., Fan, R., Wang, X., Kong, Y., Wang, W. & Hao, Q. CXL over Ethernet: A Novel FPGA-based Memory Disaggregation Design in Data Centers. *ArXiv Preprint ArXiv:2302.08055*. (2023)

[12] Arif, M., Assogba, K., Rafique, M. & Vazhkudai, S. Exploiting CXL-Based Memory for Distributed Deep Learning. *Proceedings Of The 51st International Conference On Parallel Processing*. (2023), https://doi.org/10.1145/3545008.3545054

[13] Ahn, M., Chang, A., Lee, D., Gim, J., Kim, J., Jung, J., Rebholz, O., Pham, V., Malladi, K. & Ki, Y. Enabling CXL Memory Expansion for In-Memory Database Management Systems. *Data Management On New Hardware*. (2022), https://doi.org/10.1145/3533737.3535090

[14] Jung, M. Hello Bytes, Bye Blocks: PCIe Storage Meets Compute Express Link for Memory Expansion (CXL-SSD). *Proceedings Of The 14th ACM Workshop On Hot Topics In Storage And File Systems*. pp. 45-51 (2022), https://doi.org/10.1145/3538643.3539745