EmuCXL: A Emulation Framework for CXL-based Disaggregated Memory Applications

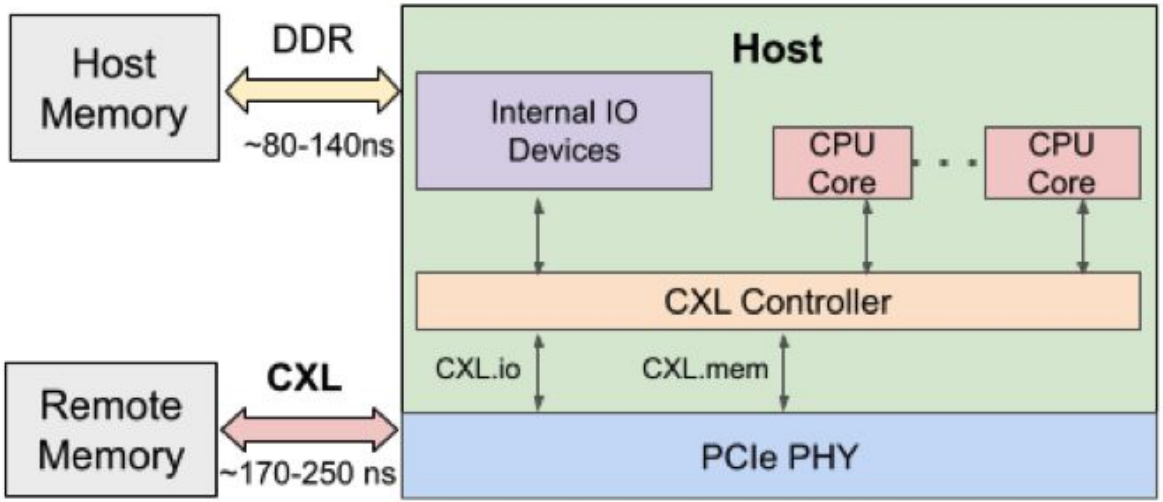
Raja Gond and Purushottam Kulkarni

IIT Bombay

Problems

1. No standard APIs for building applications on CXL-attached memory
2. APIs developed in industry but mostly proprietary
3. Difficulty in finding easy to use CXL memory emulation setups
4. No open-source way to get a end-to-end CXL-attached memory emulation going

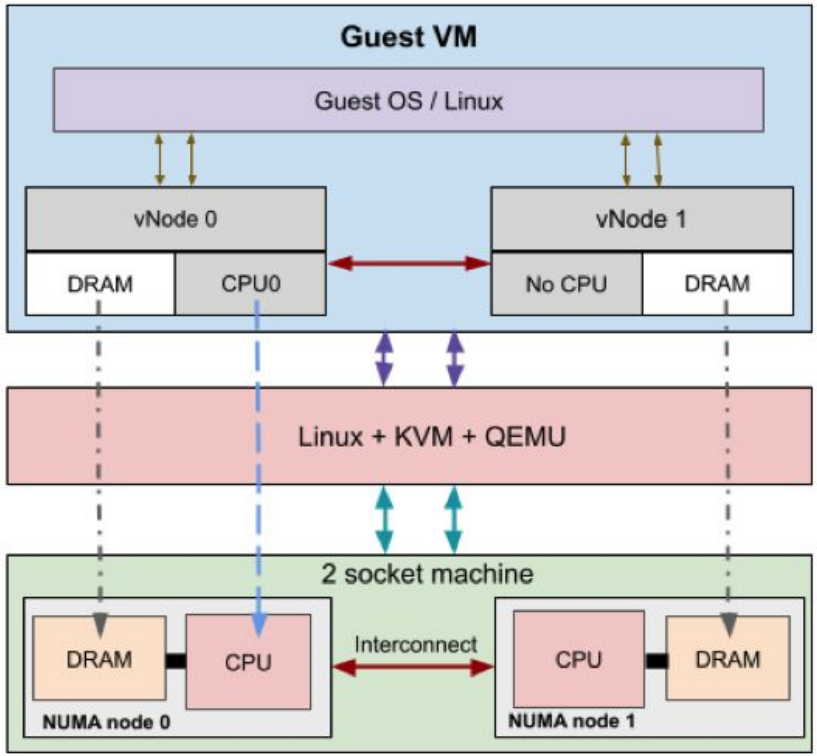
Architecture of CXL-based Disaggregated Memory



Implementation of EmuCXL

1. A VM with NUMA-based emulation setup
    a. Similar to Pond
    b. A 2-socket server as the underlying hardware
    c. A 2-node QEMU + KVM virtual machine
        i. 1st vNode mapped to socket 1's CPU and Memory
        ii. 2nd vNode mapped to socket 2's Memory only
2. An software setup with a standardized API for CXL-style memory pooling
    a. A user-space library
    b. A kernel backend implemented as a Kernel module

Virtual Appliance Setup using QEMU



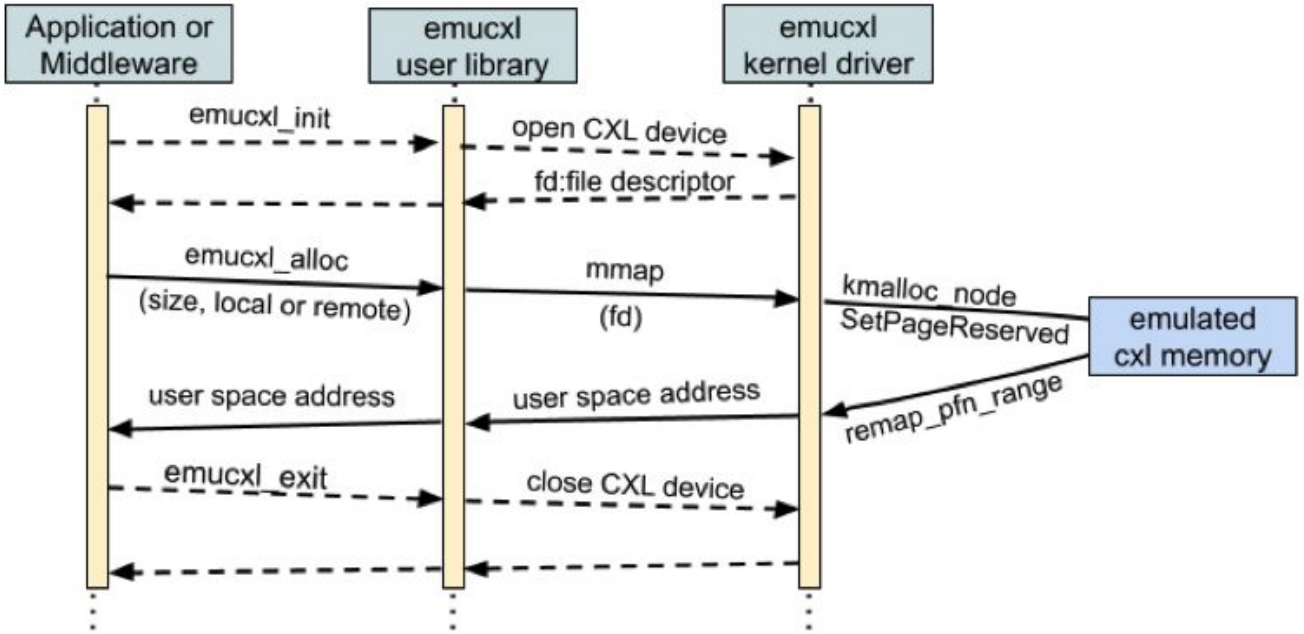
User-Space APIs in the EmuCXL library

Standardized API	***Description***
`void emucxl_init()`	open CXL device file, store file descriptor, initialized emulated memory sizing
`void emucxl_exit()`	free all allocated memory and close the device file
`void* emucxl_alloc(size_t size, int node)`	allocate memory either remotely or locally node = 0 for local memory, and 1 for remote memory returns virtual address to calling process
`void emucxl_free(void* address, size_t size)`	free allocated memory block of the specified size
`void* emucxl_resize(void* address, size_t size)`	allocate memory of new size on same node, copy, free earlier allocation, return address
`void* emucxl_migrate(void* address, int node)`	allocate memory on specified node, migrate all data to new address and return address
`bool emucxl_is_local(void* address)`	check if address is mapped from local or remote memory
`int emucxl_get_numa_node(void* address)`	return the NUMA node associated with the given memory address
`size_t emucxl_get_size(void* address)`	return size of memory allocated of the specified memory address
`size_t emucxl_stats(int node)`	return size of total memory allocated on given NUMA node
`bool emucxl_read(void* addr, int, void* buf, size_t)`	read/write specified number of bytes from/to the memory address and store in/from buffer
`bool emucxl_write(void* buf, int, void* addr, size_t)`	
`void* emucxl_memset(void* addr, int value, size_t)`	fill a block of memory with either 0 or -1
`void* emucxl_memcpy(void*, const void*, size_t)`	copy data from source to destination address
`void* emucxl_memmove(void*, const void*, size_t)`	move data from source to destination address unlike memcpy, memmove ensures correct handling of overlapping memory regions when moving the data.

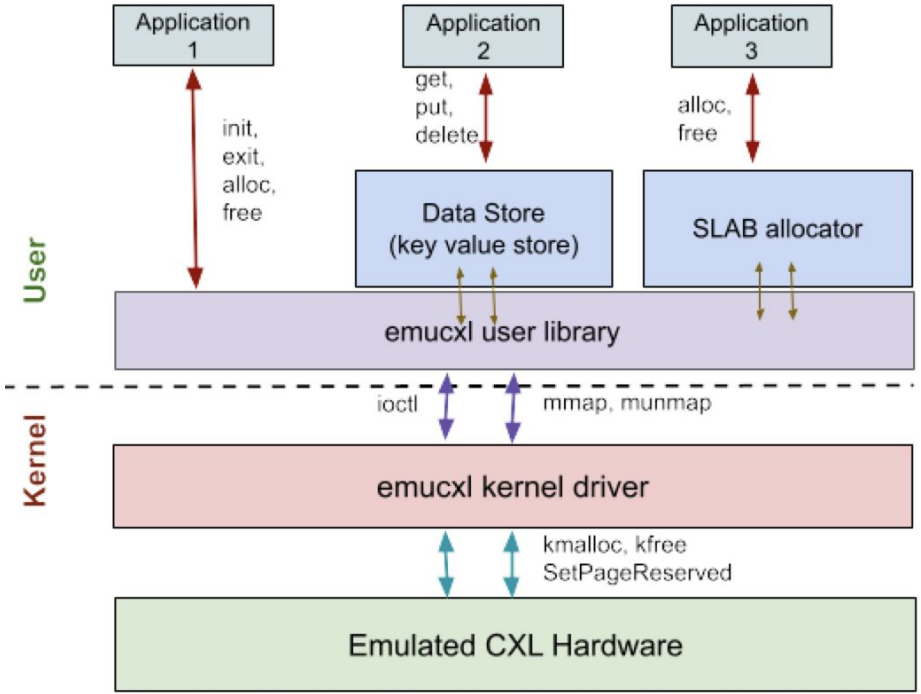
Kernel Backend

1. Implemented as a Loadable Kernel Module
2. Added IOCTL commands
   a. `EMUCXL_INIT, EMUCXL_EXIT, EMUCXL_ALLOC, EMUCXL_FREE`
3. Upon loading the kernel module:
   a. A char device (`/dev/emucxl`) is created and registered within the kernel
4. The char device implements `open`, `close`, `mmap`, and `ioctl` operations in the `file_operations` struct
5. In the NUMA-aware `mmap` operation, memory is allocated using `kmalloc_node` on the NUMA node and is mapped to a user address space using `remap_pfn_range`
6. To avoid swapping the pages out, the `PG_reserverd` bit is set on the mapped pages using `SetPageReserved`
7. `emucxl_alloc` uses this NUMA-aware `mmap`, `emucxl_free` uses `munmap`/ioctl(`EMUCXL_FREE)`, and `emucxl_init` and `_exit` uses `open`/ioctl(`EMUCXL_INIT)` and `close`/ioctl(`EMUCXL_EXIT)` respectively

Control Flow of EmuCXL



EmuCXL Application Layers



EmuCXL Applications

1. Direct Access
    a. Applications that make call directly to the user-space emucxl APIs
    b. The management of local and remote memory is embedded with the logic of the applications
    c. Authors implement a Queue using the EmuCXL library
        i. They report numbers for a Queue entirely in the local memory
        ii. And another queue entirely in the remote memory
        iii. They don't discuss a local + remote unified queue or any policies to transparently move elements between them

Execution Time (ms)	Enqueue		Dequeue	
	Local	Remote	Local	Remote
Mean	502.98	567.21	417.69	500.40
Std. Dev.	9.23	7.93	8.71	3.66

Reported timings for 15K operations each

EmuCXL Applications

1. Middleware Driven Usage
    a. Key Value Store:
        i. A Key-Value store uses the EmuCXL library and its API to provide GET, PUT, DELETE semantics to applications
        ii. Access Semantics:
            1. GET: First search for a key in local memory and if not found search in remote memory. If a key is frequently accessed, move it from remote to local memory using a LRU eviction policy for the local memory
            2. PUT: First added to the local memory, if local memory is exhausted, move to remote memory using LRU
            3. DELETE: Search for the key to delete in local first, then search in remote

Thank You !