# A Comprehensive Simulation Framework for CXL Disaggregated Memory

Yanjing Wang† Lizhou Wu† Wentao Hong† Yang Ou† Zicong Wang† Sunfeng Gao† Jie Zhang
Sheng Ma† Dezun Dong† Xingyun Qi† Mingche Lai† Nong Xiao†
National University of Defense Technology†, Peking University



***Abstract*—Compute eXpress Link (CXL) has emerged as a key enabler of memory disaggregation for future heterogeneous computing systems to expand memory on-demand and improve resource utilization. However, CXL is still in its infancy stage and lacks commodity products on the market, thus necessitating a reliable system-level simulation tool for research and development. In this paper, we propose CXL-DMSim[1], an open-source full-system simulator to simulate CXL disaggregated memory systems with high fidelity at a gem5-comparable simulation speed. CXL-DMSim incorporates a flexible CXL memory expander model along with its associated device driver, and CXL protocol support with CXL.io and CXL.mem. It can operate in both app-managed mode and kernel-managed mode, with the latter using a dedicated NUMA-compatible mechanism. The simulator has been rigorously verified against a real hardware testbed with both FPGA-based and ASIC-based CXL memory prototypes, which demonstrates the qualification of CXL-DMSim in simulating the characteristics of various CXL memory devices at an average simulation error of 4.1%. The experimental results using LM-bench and STREAM benchmarks suggest that the CXL-FPGA memory exhibits a ∼2.88× higher latency than local DDR while the CXL-ASIC latency is ∼2.18×; CXL-FPGA achieves 45-69% of local DDR memory bandwidth, whereas the number for CXL-ASIC is 82-83%. We observed that the performance of CXL memory is 3× more sensitive to Rd/Wr patterns than local DDR, with the max. bandwidth at 74%:26% rather than 50%:50% due to the current compromised CXL+DDR controller design. The study also reveals that CXL memory can significantly enhance the performance of memory-intensive applications, improved by 23× at most with limited local memory for Viper and approximately 16% in bandwidth-sensitive scenarios such as MERCI. Moreover, the simulator's observability and expandability are showcased with detailed case-studies, highlighting its great potential for research on future CXL-interconnected hybrid memory pool.**

## I. Introduction

With the prevalence of massive data-driven applications such as AI/ML and big data analytics, the demand for larger memory is ever-increasing in today's heterogeneous parallel computing systems. Over the past two decades, the CPU performance has been boosted dramatically thanks to Moore's law and multi/many-core scaling. However, the memory capacity and bandwidth per core have been decreasing, which apparently poses a bottleneck for system performance [1]. In modern datacenters, the deployment unit is a monolithic server which contains closely-coupled computing and memory resources. This monolithic architecture for many years is always CPU-biased, leading to memory over-provision across the whole system. It has been observed that more than 50% of the aggregated memory is unused most of the time in production clusters at Google and Facebook [2]. Considering the rising DRAM chip prices in recent years, the under-utilization of memory resources becomes prohibitively expensive, which greatly boosts the TCO of datacenters [3].

With the advent of memory disaggregation technologies, new solutions can be explored to tackle the memory wall and memory under-utilization challenges [4]. Memory disaggregation technologies decouple memory resources from CPUs, providing a feasible option for memory pooling. Conventionally, Remote Direct Memory Access (RDMA) technology is exploited to realize memory disaggregation [2], [5]–[8]. But RDMA is based on networking IO semantics which requires specialized NICs and software intervention, leading to a latency multiple orders of magnitude longer than that of local memory access. In recent years, several low-latency and high-bandwidth memory-coherent interconnect protocols arise in industry, which have shown advantages over RDMA for memory disaggregation [9]–[15]. Among them, the CXL protocol [16] is very promising and embraced by an increasingly number of semiconductor vendors worldwide. With CXL, memory expansion becomes more flexible over the interconnect fabric while enabling coherent memory access via *load/store* instructions. Furthermore, the CXL protocol is independent on the underlying memory technology, which can be DRAM, Flash, or even emerging non-volatile memories such as MRAM and RRAM. This facilitates the construction of a unified heterogeneous memory pool for future energy- and cost-efficient computing systems.

Despite its attractive features disclosed in the protocol specifications, CXL is still in its infancy stage and lacks commodity products on the market. As a result, the prior research work on CXL-based memory disaggregation is conducted based on four main methods: software-based emulation [17], software-based simulation, [18]–[20], hardware-based emulation [21]–[23], and hardware prototyping [9], [24], [25]. However, the software-based emulation such as QEMU fails to model the physical characteristics and internal micro-architecture of real CXL memory devices. The simulation endeavors of gem5-CXL and CXLMemSim are both nascent, with gem5-CXL failing to accurately model the CXL protocol behavior and provide clear access interfaces, while CXLMemSim lacks full-system simulation capabilities and cycle-accurate fidelity,

[1]Open sourced at https://github.com/ferry-hhh/CXL-DMSim.

resulting in limited functionality and poor usability. The hardware-based emulation such as remote NUMA lacks CXL protocol support and there is a big difference in the memory access path and performance. As for the fourth method, there are currently no market-ready prototypes of CXL-based memory-disaggregated systems; CXL commodity products are also expensive to produce and purchase. Given the above limitations, there is a clear need for an accurate, cost-effective, and flexible tool for research on CXL-based disaggregated memory systems.

In this paper, we present CXL-DMSim, a full-system CXL Disaggregated Memory Simulator based on gem5 for cycle-accurate simulation, architectural exploration, and evaluation of CXL-interconnected memory systems. CXL-DMSim is as easily configurable as the original gem5 simulator and fits to a variety of CXL devices. It has been rigorously verified and calibrated by a real-world CXL1.1 testbed with both an in-house ASIC CXL memory expander and an FPGA-based CXL device prototype. To the best of our knowledge, CXL-DMSim is the first usable full-system disaggregated memory simulator. The main contributions of this paper are listed below.

- A flexible device model of CXL memory expander (Type 3 device) which currently supports both DRAM and Flash as underlying storage media.
- Supports for CXL.io and CXL.mem sub-protocols, which are used to enumerate, configure, and access our CXL memory device on CXL-DMSim.
- A driver for the device to operate in an application-managed mode and a NUMA-aware memory management mechanism to operate in a kernel-managed mode.
- An extensive evaluation that validates CXL-DMSim including performance tests, usability&fidelity tests, real-world app. tests, and observation&expandability tests; this verifies the system's feasibility and offers guidance for appropriate usage of CXL disaggregated memory.

## II. Background and Motivation

### A. CXL Protocol

Compute eXpress Link (CXL) is a cache-coherent high-speed interconnect for CPUs, memory expanders, and accelerators [16]. CXL features memory coherence between host memories and device-attached memories, enabling resource pooling and sharing for higher performance and reduced software complexity at lower cost. It was first introduced by Intel in 2019 and since then has been widely embraced by the industry. As of today, CXL protocol has undergone updates with three major versions: 1.0/1.1, 2.0, and 3.0/3.1.

CXL is built on the foundation of the PCIe physical layer. It consists of three sub-protocols, namely CXL.io, CXL.cache, and CXL.mem. CXL.io is functionally equivalent to PCIe in discovering and enumerating devices. CXL.cache enables IO devices and accelerators to directly access and cache the host memory. CXL.mem allows the host to access memories that are attached to CXL devices via load/store instructions.

The three sub-protocols identify three device types for distinct application scenarios. Type 1 devices employ CXL.io and

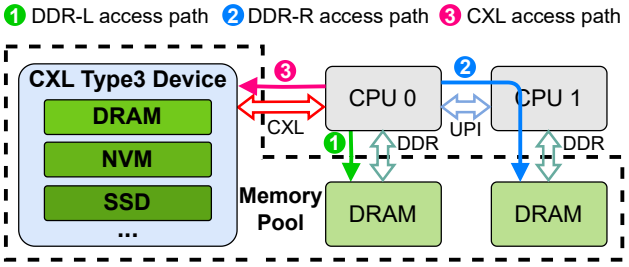


Fig. 1: Memory expansion via a CXL Type 3 device.

CXL.cache sub-protocols. This type of devices are typically referred to as IO devices such as smart NIC which has no memory attached to it. Type 2 devices (accelerators), such as GPU owning device memory and specialized compute units, utilize all three sub-protocols. Type 3 devices use the CXL.io and CXL.mem sub-protocols and are usually depicted as memory expanders, as illustrated in Fig. 1. This paper focuses on the CXL Type 3 devices for flexible memory expansion and pooling of future heterogeneous computing systems.

### B. Related Work

The prior work performs research on CXL-based memory disaggregation systems with the following four methods.

*1) Software-Based Emulation:* QEMU is a widely used and open-source emulator and virtualizer. As an emulator, it can perform full-system emulation which emulates all system components, including the processor and peripherals. In terms of CXL support, QEMU has begun to support CXL 2.0, with a few CXL components such as CXL Host Bridge, CXL Root Ports, CXL Switch and CXL Memory Devices [17]. A recent work named Mess [26] presented a novel memory benchmark which measures bandwidth-latency curves. Based on the curves, a one-size-fit-all memory emulator was proposed to emulate the real-time latency of different types of main memories covering DDR4, DDR5, HBM, and CXL memories.

*2) Software-Based Simulation:* gem5-CXL [18] is an open-source project we found on GitHub. It crudely adds a custom bus between the original gem5 MemBus and DDR controllers to introduce an extra delay. This extra delay actually models the deteriorated performance of CXL memory in comparison to DDR memory. Another project CXLMemSim [19], [20] is a lightweight trace-driven simulator for the CXL.mem sub-protocol. It allows users to define the desired topology of a modeled memory pool as well as the latency of each component. Basically, it relies on the execution of an unmodified program on a real server. The execution time is divided into multiple epochs, each of which is recalculated in terms of execution time based on the collected memory access events and the modeled latency of CXL memory.

*3) Hardware-Based Emulation:* Due to the unavailability of commercial CXL hardware, many researchers have conducted emulations of CXL-expanded memory exploiting the NUMA mechanism. These studies [21]–[23] leverage NUMA-enabled hardware to emulate CXL disaggregated memory without directly handling the CXL protocol. Given the similar charac-

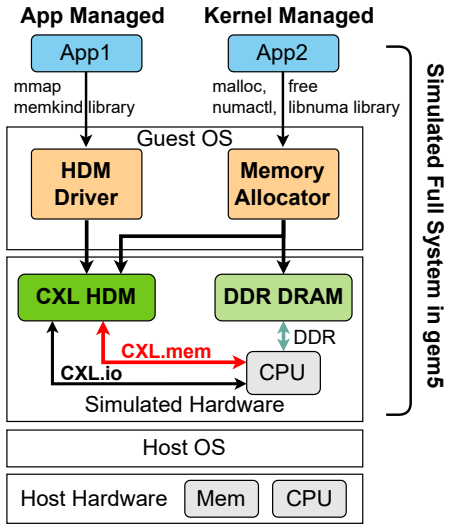


Fig. 2: Architecture of the proposed CXL-DMSim simulator.

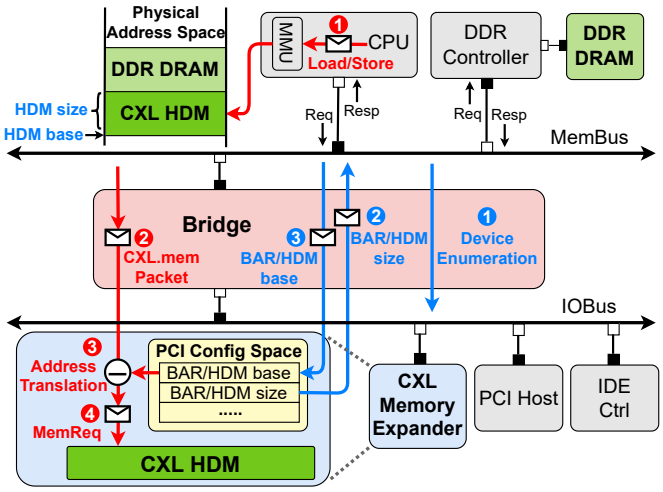


Fig. 3: System key component layout and CXL memory device access flow on the x86 platform of CXL-DMSim.

teristics in bandwidth and latency between CXL memory and remote NUMA memory, this approach provides a reference for the study of computing systems with real CXL memory.

*4) Hardware Prototyping:* There exist several ongoing projects to develop CXL hardware prototypes. For example, DirectCXL [9] utilizes FPGAs to implement a CXL memory controller and a RISC-V processor that supports CXL. Transparent Page Placement (TPP) [24] investigates page placement strategies for a tiered memory system using pre-production x86 CPUs with CXL 1.1 support and FPGA-based CXL memory expansion card. Sun et al. [25] perform a comprehensive characterization of CXL memory on a real CXL hardware platform.

### C. Motivation

Although QEMU serves as a generic system emulator, it is primarily used for functional-level emulation and verification. It does not model the physical characteristics and internal micro-architecture of CXL devices. Thus, QEMU is unsuitable for architectural exploration and accurate performance evaluation of CXL-based systems. Similarly, the Mess emulator targets at reproducing the bandwidth-latency curves of the SystemC model of a Micron's CXL memory expander; Real CXL memory devices may behave differently in a CXL-connected system. We also argue that the cpuBW value calculated from the most recently 1000 memory operations ($\sim$100 $\mu$s) may not be able to accurately capture the realistic memory bandwidth (messBW) under real-world applications.

Additionally, existing software-based simulation attempts are far away from readiness. gem5-CXL connects a simulated CXL device to the MemBus rather than mounting it on the IOBus to simulate peripherals. It does not model the behaviors of the CXL protocol (based on PCIe PHY) and lacks clear access interfaces with the modeled device. The project has not been updated for four years and is currently in a stagnant state. CXLMemSim is not a full-system simulator in essence, as neither the CXL protocol and system architecture nor the associated software stack are modeled. It lacks cycle accuracy and has limited functionalities, resulting in a poor usability.

As for the other emulation approach, the remote NUMA memory lacks realism and accuracy in mimicking CXL memory. Specifically, there is a significant difference in access latency and bandwidth between these two types of memory, as emphasized in [25]; they also have different memory access paths (see Fig. 1). In terms of the hardware prototype method, there are currently no market-ready hardware prototypes of CXL-based memory disaggregation systems. Existing hardware prototypes and commercial devices are expensive to produce, due to the long silicon development cycle. Moreover, hardware prototypes also lack flexibility in system configuration and observation.

The limitations of current emulators, simulators and CXL hardware call for a comprehensive and realistic simulation solution for agile design and evaluation of CXL-interconnected systems. This motivates us to propose CXL-DMSim, which is a configurable, scalable, and cost-efficient simulator.

## III. CXL-DMSim Design and Implementation

### A. Simulator Architecture

Fig. 2 shows the overall architecture of CXL-DMSim, with three main components built on top of gem5. First, we added a CXL memory expander model, illustrated as CXL *Host-managed Device Memory* (HDM) in the figure, to the gem5 simulator. It connects to the system as a PCI device with standalone memory space through IOBus (see Fig. 3). Second, we implemented the CXL sub-protocols CXL.io and CXL.mem, which are required for Type 3 devices. The CXL.io sub-protocol is used for the CPU to enumerate and configure CXL devices; it is achieved by reusing the original PCI protocol in gem5. The CXL.mem sub-protocol allows the CPU to access the CXL HDM directly; it is achieved by integrating new CXL.mem packets as well as by extending internal components such as bridge in gem5.

Third, we designed a dedicated device driver and a NUMA-compatible memory management mechanism in the guest OS, to manage the allocation and deallocation of CXL HDM as well as to provide interfaces to upper-level user applications.

Depending on the CXL HDM management mechanism in the guest OS, CXL-DMSim provides users with two ways to use our CXL memory expander: *Application Managed* (AM) and *Kernel Managed* (KM). In the AM mode, applications can allocate and free CXL HDM using the *memkind* library or the *mmap* system call interface provided by our device driver. This usage mode offers users more flexibility and granularity in managing CXL HDM but also imposes the pressure of modifying legacy programs. In the KM mode, the CXL memory expander is exposed to the OS kernel as a CPU-less (or memory-only) NUMA node, with the kernel managing the allocation and release of CXL HDM transparently to applications. Users can seamlessly utilize CXL HDM through existing tools such as *numactl*. However, it is evident that in this mode the difference from the DDR memory is only reflected in the "NUMA distance". Users are not able to perceive and exploit the unique characteristics (e.g., persistence, large volume, and low power consumption) of different memory technologies behind the CXL interface. Note that the KM mode is consistent with the mainstream usage of CXL memory expanders on real-world hardware platforms.

### B. CXL Memory Expander Model

gem5 is a modular, cycle-accurate computer system simulator. Its *Full-System* (FS) mode executes both user-level and kernel-level instructions and models a complete system including the OS and hardware devices, thereby simulating interactions between software and hardware more realistically [27]–[29]. Fig. 3 illustrates the structure of an x86 platform with classic memory subsystem used in the FS simulation mode. Note that the operating flow marked with red and blue arrows will be explained in Sec. III-C. The CPU, cache, and DDR controller are placed on the coherent MemBus, while devices such as PCI host and IDE controller are mounted on the non-coherent IOBus. These two buses are connected via a bridge. In our implementation, the CXL memory expander is accessed as a PCI device attached to the IOBus.

Fig. 4 shows our CXL memory expander model implemented in gem5. The model is structured from top to bottom, with a response port for communication, a memory controller for parsing read and write packets, and a memory medium for data storage. Since gem5 models external devices in a coarse-grained manner using the *atomic mode* (transactional behavior is represented as simple accumulated latency), we introduced the event-driven *timing mode* into the CXL memory expander to achieve accurate memory access behavior. The response port is connected to the IOBus, enabling it to receive request packets directed to the memory expander and send response packets to the request sources.

The memory controller in the device model has two main functions. First, it parses CXL transaction packets, which include read and write configuration packets for CXL.io and memory transaction packets for CXL.mem. Second, it translates the address in the incoming packets to the actual address for accessing the backend memory medium. For the CXL.io sub-protocol, we have implemented the PCI configuration space of the expander following the PCI device programming specifications in gem5. This includes basic device information and Base Address Register (BAR) space sizes, thereby enabling the device to respond to read and write transactions from the host to the configuration space. For the CXL.mem sub-protocol, the memory controller first identifies the command fields in the packets according to the protocol specifications. Then, it calculates the actual address based on the destination address of packet and the base address stored in the BARs. The derived read/write command and its associated address are subsequently encapsulated in a new request to the memory medium for programming and retrieving data.

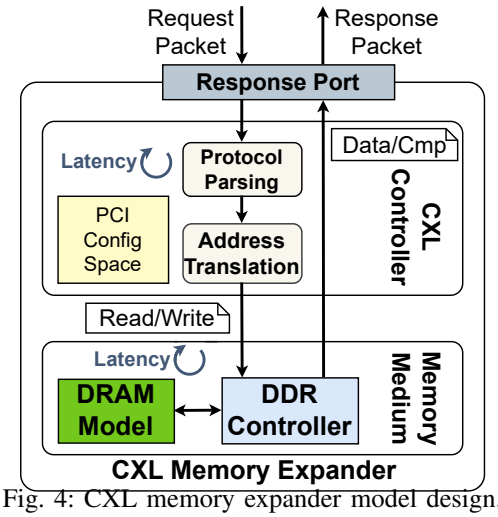

Fig. 4: CXL memory expander model design.

The memory medium is designed to be technology-agnostic, which means it can be the conventional charge-based memories such as DRAM and Flash, or emerging resistance-based non-volatile memories such as RRAM and MRAM. In the case of DRAM, the memory capacity of device is declared via the BARs. During the startup of simulation on CXL-DMSim, the host machine is requested to allocate memory in the heap space of the host process based on the BAR space capacity specified in the Python script. Afterwards, upon receiving read and write requests from the memory controller, the memory medium accesses the data at the corresponding address and generates responses if needed. For the memory medium, we currently offer two DRAM models. The first is a coarse-grained model that offers a faster simulation at the cost of some accuracy. The second is the native DRAM model from gem5, which provides higher simulation accuracy but operates at a slower speed. Additionally, thanks to the modular design of gem5, other existing memory models, such as the NVM model and DRAMSim model, can be easily integrated into our expander model to accommodate the research needs of various users.

Fig. 5 shows the modeled key contributions of latency for a CXL.mem access request from CPU and its response from the CXL memory expander in CXL-DMSim. To capture the

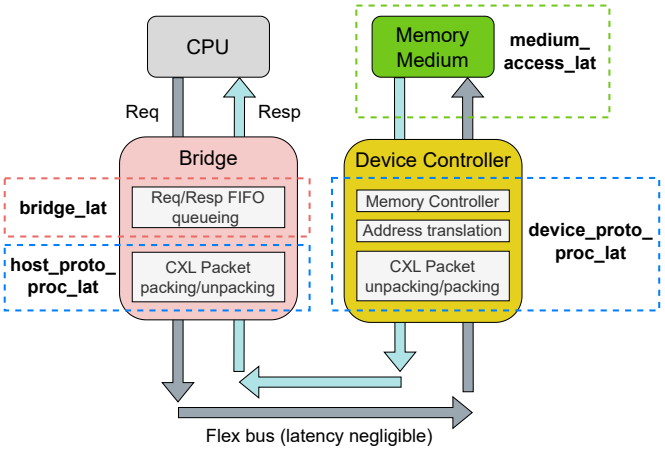


Fig. 5: Latency breakdown for a CXL memory access request from CPU to CXL memory and its response with a reversed path in CXL-DMSim.

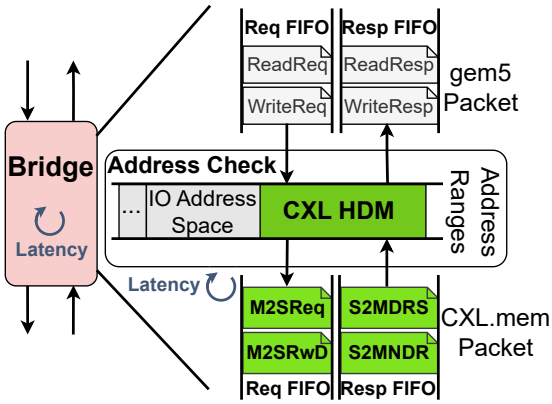


Fig. 6: Packet interception and transformation.

timing behavior of CXL memory expanders, our CXL memory expander model provides two parameters to simulate the processing latency of the CXL.mem sub-protocol and the access latency of the memory medium: *device_proto_proc_lat* and *medium_access_lat*. Note that the other two key latency parameters *bridge_1at* and *host_proto_proc_lat* associated with the bridge in the figure will be explained later in Sec. III-C.

### C. CXL Protocol Support

The CXL memory expander uses CXL.io and CXL.mem sub-protocols. Since CXL.io is functionally similar to the traditional PCI protocol in the transaction layer, we reused the existing PCI protocol in gem5 to implement device enumeration and configuration. This process is handled with three main steps during system startup, as shown with the blue arrows in Fig. 3. Step ①: When enumerating a CXL memory expander, its driver in OS first queries the internal memory size declared by the BARs using CXL.io configuration read transactions. Step ②: Based on the retrieved size, the HDM is then mapped into the host's physical address space. Step ③: The mapped base address is written into the expander configuration space via CXL.io configuration write transactions. By means of the above process, the CXL memory expander can be discovered and subsequently accessed by the host in the system.

The host accesses HDM via the CXL.mem sub-protocol with four main steps, as illustrated with the red arrows in Fig. 3. Step ①: Memory access requests are first initiated towards HDM through load/store instructions. Step ②: These requests are then routed to downstream devices via the bridge, where gem5-internal packets are converted into CXL.mem packets in order to communicate with the CXL memory expander. Step ③: When the CXL.mem packets arrive at the expander, the CXL memory controller converts the request address to the expander's internal memory address based on the HDM base address. Step ④: The controller forwards the converted request to the backend memory medium for actual memory access.

The CXL.mem sub-protocol defines two types of communication endpoints. One is called master, typically a local agent in the host processor. The other is subordinate such as a memory controller in the CXL memory expander. This protocol specifies two types of packets for communication between the master and subordinate agents: those from the master to the subordinate are called Master-to-Subordinate (M2S) packets, and those in the opposite direction are called Subordinate-to-Master (S2M) packets [16].

CXL-DMSim extends the packet-based point-to-point communication mechanism in gem5 to support the simulation of the CXL.mem sub-protocol. First, when the CPU launches a ReadReq/WriteReq gem5 packet, its command field is updated to include a read/write transaction command used by the CXL.mem sub-protocol. Subsequently, the bridge connecting the memory bus and I/O bus intercepts the packet if destined to the CXL memory expander. Fig. 6 shows that the bridge contains two pairs of FIFO queues, one pair for buffering upstream gem5 packets and the other pair for buffering downstream CXL.mem packets. The bridge checks the address and captures the gem5 packet in the upstream Req FIFO, then transforms it into a M2SReq/M2SRwD packet. Thereafter, the CXL.mem packet is buffered in the downstream Req FIFO and then sent out to the CXL memory expander. Similarly, when a response packet (S2MDRS or S2MNDR) from the CXL memory expander arrives at the bridge, it is buffered in the Resp FIFO and then converted into a ReadResp/WriteResp packet. The gem5 packet is buffered in the upstream Resp FIFO before being transmitted to the memory bus.

Note that the bridge module includes four configuration parameters (see Fig. 5): *bridge_lat*, *host_proto_proc_lat*, *req_fifo_depth*, and *resp_fifo_depth*. The bridge_lat represents the inherent latency through the bridge module, which is already present in the original gem5 code. The host_proto_proc_lat represents the processing latency for CXL.mem sub-protocol packets. The req_fifo_depth and resp_fifo_depth determine the depths of the FIFO queues used for buffering CXL.mem request and response packets, respectively. These queue depths impact the CXL memory expander's capability to handle concurrent requests.

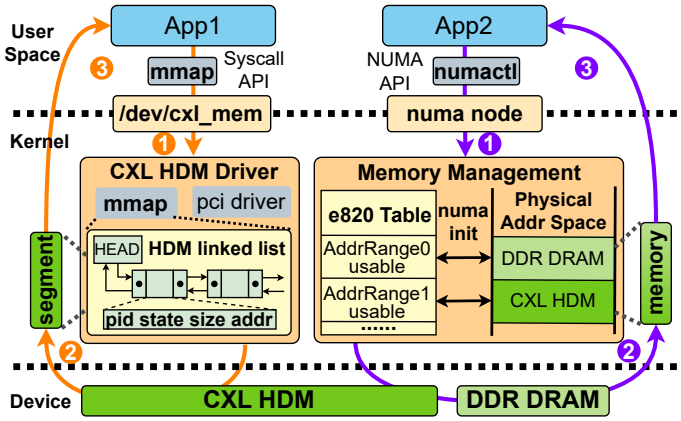


Fig. 7: Two ways of managing CXL HDM in the OS.

### D. CXL HDM Management in OS

Fig. 7 shows the software stacks of operating the CXL memory expander in both the AM and KM modes. In the AM mode, the CXL HDM driver serves not only for device enumeration and configuration but also furnishes upper-layer applications with a suite of system call interfaces to access and manage HDM. As the memory expander is mounted on the PCI bus as an external device to the host, its driver adheres to the PCI driver programming paradigm. During the enumeration phase of the system, the "pci driver" component of the driver identifies the CXL memory expander and creates a character device file named */dev/cxl_mem*.The "file operations" component of the driver implements system calls (such as mmap) for accessing HDM and initializes a doubly linked list for managing HDM allocation. Each node in the list records the process *PID* that owns the memory block, the current *state* of the memory block (FREE or BUSY), the *size* of the memory block, and the offset *address* of the memory block. An application accesses HDM via the AM mode in three main steps, as shown with the orange arrows in Fig. 7. Step ①: The application first opens the device file */dev/cxl_mem* using the driver's *open* system call to acquire the device file descriptor. Then, it uses the *mmap* system call to request for a fixed-size HDM region and maps it into the virtual address space of processor. Step ②: Considering the potential for multiple processes to access the HDM concurrently, the driver implements a mutex lock pertaining to the HDM, ensuring that only one process has exclusive access at any given time. Once the application process gets the mutex, the memory allocator examines the HDM and allocates a contiguous segment of physical memory. If the allocation is successful, the allocator obtains the offset of the physical address for this segment, maps the virtual memory area to it, and updates the linked list tracking HDM allocations. Step ③: The application can use the pointer returned by the *mmap* function to read from or write to the CXL HDM directly. Furthermore, the pointer can also be passed to the *memkind* library for the purpose of unified memory management of DDR DRAM and CXL HDM.

Although the AM mode provides a straightforward method to access the CXL HDM, it requires modifications to applications' source code and does not utilize the kernel's existing

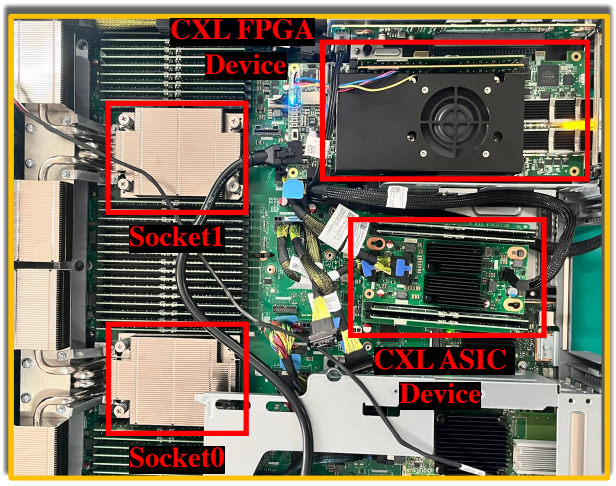


Fig. 8: Top view of our CXL hardware testbed.

tiered memory management infrastructure. As a result, the prevalent approach in real-world hardware platforms treats the CXL memory device as a CPU-less NUMA node. This approach remains transparent to applications and enables NUMA-aware memory management, which facilitates flexible migration between DDR DRAM and CXL HDM. Therefore, CXL-DMSim also provides a NUMA API for accessing HDM, which corresponds to the KM mode. To achieve this, we added an e820 table entry to provide the kernel with the size and range of the CXL HDM, enabling the kernel to recognize the CXL HDM as part of the available system memory during boot time. Note that this approach mirrors the configuration process used on real hardware platforms for recognizing and configuring CXL devices. Additionally, we adjusted the kernel *numa_init* routine to initialize both DDR DRAM and CXL HDM as two separate NUMA nodes. An application accesses HDM via KM mode in three main steps, as shown with the purple arrows in Fig. 7. Step ①: The application first uses the *numactl* tool to bind memory allocations to the HDM node or opt for other NUMA strategies, such as memory interleaving. Step ②: The kernel's memory management subsystem then handles memory allocation transparently, with the allocated physical memory varying based on the specified NUMA strategy. Step ③: The application can access HDM through standard memory allocation interfaces such as *malloc*; the kernel is responsible for the allocation and migration of memory pages across DDR DRAM and CXL HDM.

## IV. Experiments and Evaluation

### A. Experimental Setup

We have conducted comprehensive experiments on both real hardware testbed and our proposed CXL-DMSim. The detailed configurations of these two test platforms are as follows.

*1) Hardware Testbed:* The testbed is a high-performance dual-socket x86 server shown in Fig. 8; its internal structure is illustrated in Fig. 9. The server has dual-socket Intel Xeon Platinum 8468V processors, each of which has 4 integrated

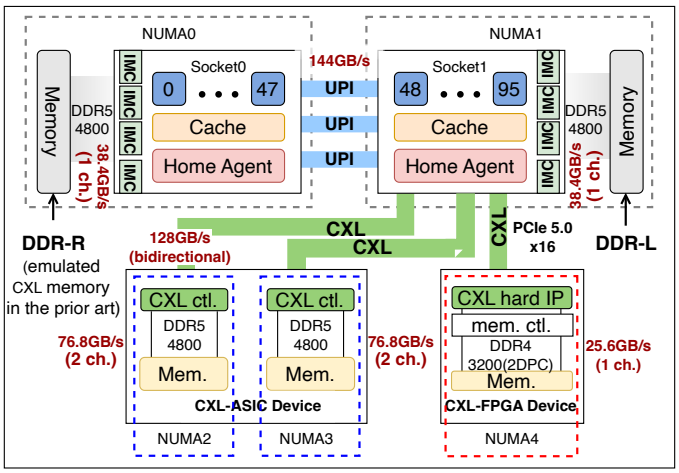


Fig. 9: Architecture diagram of our CXL hardware testbed.

TABLE I: Host configurations for our hardware testbed and CXL-DMSim simulator.

| Config. Parameter | CXL Testbed Host | CXL-DMSim Host |
|---|---|---|
| Linux kernel version | v6.5.0 | Modified v5.4.49 |
| CPU type | Xeon® Platinum 8468V | X86O3CPU |
| CPU cores | 48 | 48 |
| Local DRAM type | DDR5 4800 | DDR5 4400 |
| #Memory channels | 1 | 1 |
| Local DRAM size | 32GB | 32GB |
| L1 dcache size | 48KB | 48KB |
| L1 icache size | 32KB | 32KB |
| L2 cache size | 2MB | 2MB |
| LLC size | 97.5MB | 96MB |

memory controllers with 8 memory channels. For the purpose of fair comparison with a CXL link, we only enabled a single memory channel (i.e., a 32GB DDR5 4800MT/s DIMM) for each NUMA node in our characterization experiments. The detailed host configurations are listed in Table I.

On our hardware testbed, we measured two types of CXL memory devices: a real CXL FPGA memory device (denoted as CXL-FPGA) and a real CXL ASIC memory device (denoted as CXL-ASIC); the details are listed in Table II. Both of these two devices are connected to Socket1, as illustrated in Fig. 9. The CXL-FPGA device is an Intel Agilex I-Series FPGA Development Kit with an integrated hard CXL IP [30]. It is equipped with 16GB 3200MT/s DDR4 memory, connected to the CPU via an CXL 1.1 interface. The CXL-FPGA device is recognized as a separate CPU-less NUMA node (denoted as NUMA4); it can be used and managed using existing NUMA software infrastructure. The in-house CXL-ASIC device has two CXL memory controllers, each of which is composed of a *CXL protocol controller plus a DDR5 controller* driving two DIMMs. As each DIMM is populated with a 32GB DDR5 4800MT/s memory module, the CXL-ASIC device owns a total memory capacity of 128GB. It appears in the OS as two separate NUMA nodes without CPUs (denoted as NUMA2 and NUMA3 respectively).

We also measured the DDR5 memory on the remote NUMA node (denoted as DDR-R on NUMA0) as an *emulated CXL memory* device, as done in many of the prior works [21]–[23]. For better comparing the performance of the above-mentioned three types of memory, we also measured the local DDR5 memory (denoted as DDR-L) on NUMA1 as a baseline.

TABLE II: Real CXL memory expander configurations for CXL-FPGA and CXL-ASIC devices.

| Config. Parameter | CXL-FPGA | CXL-ASIC |
|---|---|---|
| CXL memory size | 16GB | 64GB |
| #Backend memory channels | 1 | 2 |
| Type of backend DRAM | DDR4 3200 | DDR5 4800 |
| Latency | 375ns | 284ns |

TABLE III: Modeled CXL memory expander configurations on CXL-DMSim for both FPGA type (CXL-DMSim$_F$) and ASIC type (CXL-DMSim$_A$).

| Config. Parameter | CXL-DMSim$_F$ | CXL-DMSim$_A$ |
|---|---|---|
| CXL memory size | 16GB | 64GB |
| bridge_lat | 50ns | 50ns |
| host_proto_proc_lat | 14ns | 14ns |
| device_proto_proc_lat | 60ns | 15ns |
| medium_access_lat | 50ns | 50ns |
| req_fifo_depth | 48 | 52 |
| resp_fifo_depth | 48 | 52 |

*2) CXL-DMSim Simulator:* CXL-DMSim is implemented based on gem5 v23.1 in the full-system mode. The CPU type is X86O3CPU with 32GB 4400MT/s single DDR5 memory (denoted as CXL-DMSim$_L$). The CXL memory device driver was developed based on the Linux v5.4.49 kernel. CXL-DMSim simulates a CXL FPGA memory device (denoted as CXL-DMSim$_F$) with 16GB and a CXL ASIC memory device (denoted as CXL-DMSim$_A$) with 64GB, with configurations listed in Table III. To simulate different CXL memory devices from different vendors, we introduce six configurable parameters and their typical ranges in our simulator, according to the latency breakdown of memory access paths provided in the CXL spec. [16] and relevant literature [3], [9], [31]–[35]. For our CXL FPGA and ASIC devices, the values of these parameters in the Table III have been meticulously calibrated and fine-tuned using end-to-end latency measurements on our hardware testbed. For those who are interested in using our simulator, different values can be configured to match the performance of their own CXL memory devices.

### B. Usability and Fidelity of CXL-DMSim

There exist many popular benchmarks to characterize memory performance in industry, such as Intel's MLC [36], LMbench [37], STREAM [38]. Considering the ease of use and compatibility on both hardware and simulator platforms, we selected LMbench, STREAM, and Redis-YCSB [39], [40] to evaluate the latency, bandwidth, and real-world application performance of DDR-L, DDR-R, CXL-FPGA, CXL-ASIC and CXL-DMSim$_F$, and CXL-DMSim$_A$.

*1) LMbench Test for Memory Latency:* To measure the access latency of CXL HDM, we utilized a memory latency benchmark *lat_mem_rd* in LMbench [37]. It can be used to

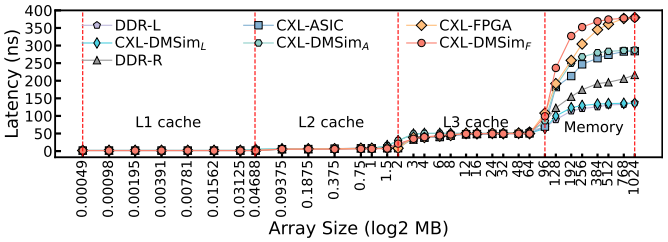


Fig. 10: Measured random memory access latency of seven memory devices: DDR-L, DDR-R, CXL-ASIC, CXL-FPGA, CXL-DMSim$_L$,CXL-DMSim$_A$ and CXL-DMSim$_F$.

measure random read latency of different memory layers, covering L1, L2, and L3 caches as well as DDR-L, DDR-R, and CXL memories. The random read pattern typically reflects the system's real latency [41], [42]. Hardware prefetch mechanisms are all disabled in BIOS settings to provide more accurate measurements of memory latencies. Note that we only present the read latency as the CXL memory is symmetric in read and write performance. The benchmark involves two key parameters: array size and stride. To ensure that the entire memory hierarchy is accurately measured, avoiding the disturbance of caching on the results meanwhile properly representing the data access patterns in real-world applications, the array size should be set to at least four times as large as LLC and no more than 80% of physical memory size [43]. Given the above considerations, we set the array size to 1024MB and the stride to 64 bytes in our experiments.

Fig. 10 shows the measurement results using LMbench. The curves contain a series of plateaus, each of which represents a level in the memory hierarchy. The right-most plateau represents the random read latency of DDR-L (130 ns), DDR-R (200 ns), CXL-FPGA (375 ns), CXL-ASIC (284 ns), CXL-DMSim$_F$ (375 ns), and CXL-DMSim$_A$ (284 ns). The latency of CXL-FPGA is approximately 2.88 times higher than that of DDR-L, while the latency of CXL-ASIC is about 2.18 times. The two types of CXL memory devices exhibit much higher latencies, because the access path of CXL memory is much longer than that of DDR-L. The latency of CXL-FPGA is 91 ns higher than CXL-ASIC, because FPGA-based CXL memory fails in fully utilizing DRAM chip performance due to FPGA's lower operating frequency compared to ASICs [44]. It is worth noting that the latencies between 192MB and 384MB for CXL-DMSim are consistently higher than those measured on real hardware. This discrepancy is primarily attributed to: 1) poor CPU model in gem5 lagging behind sophisticated real-world CPUs, 2) smaller LLC size (96MB due to restricted value selection) in the gem5 setting than that (97.5MB) on the real hardware platform. When using remote NUMA to emulate CXL memory, the read latency is relatively lower compared to real CXL memory devices, being about 1.53 times higher than that of DDR-L. Clearly, the experimental results show that CXL-DMSim can accurately simulate the access latency of real CXL memory devices, while the remote NUMA emulation cannot reflect the real latency of the devices.

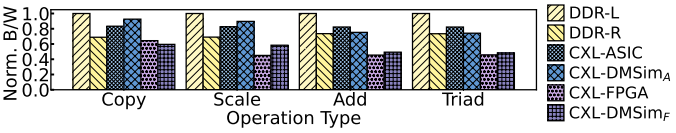


Fig. 11: Measured memory b/w normalized to DDR-L with copy, scale, add, and triad tests for different memory types.

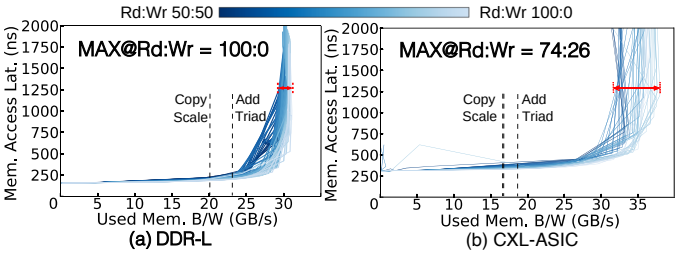


Fig. 12: Bandwidth-latency curves of DDR-L and CXL-ASIC memories with respect to varying Rd/Wr ratios.

*2) STREAM Test for Memory Bandwidth:* To measure the sustainable memory bandwidth of our prototyped and simulated systems, we used STREAM [38] as a benchmark. As a well-known benchmark, STREAM provides four distinct tests, i.e., Copy, Scale, Add, and Triad with different memory access and calculation patterns.

To ensure that STREAM measures realistic bandwidth between memory and processor without interference from caches, the data size has to be at least 8 times of the LLC size; thus we set the data size to 1024MB in our experiments. Fig. 11 shows the measurement results of the six different memory devices. Note that all the values are normalized to the bandwidth of DDR-L for the purpose of better comparison. It can be seen that the bandwidths derived from CXL-DMSim closely align with those obtained from the CXL hardware prototype; the modeling error rate is about 6% on average. The CXL-FPGA memories achieve approximately 45%-69% of the DDR-L bandwidth. Comparatively, the CXL-ASIC memories exhibit a better performance in bandwidth, achieving about 82%-83% of the DDR-L bandwidth. One can also see that the bandwidth of DDR-R is 68%-74% of the DDR-L bandwidth. This indicates that the bandwidth of DDR-R memory lies between CXL-ASIC and CXL-FPGA memories.

The reason why the measured bandwidth of the emulated CXL memory device using remote DDR5 memory (i.e., DDR-R) outperforms the real CXL FPGA memory device can be explained as follows. While CXL 1.1 is based on PCIe 5.0 x16 achieving a maximum theoretical bi-directional bandwidth of 128GB/s, the on-board DDR4 memory on our FPGA platform is limited to a maximum bandwidth of 25.6GB/s. The comparatively low bandwidth of DDR4 memory actually poses the bottleneck of the real CXL memory device. When it comes to the CXL-ASIC device, it is equipped with two channels of DDR5 memory with 76.8GB/s, which significantly enhances the CXL link's actual bandwidth in comparison to the DDR4 memory on the FPGA board. Furthermore, the ASIC's sophisticated architecture and elevated operating frequency also contribute to its higher bandwidth.

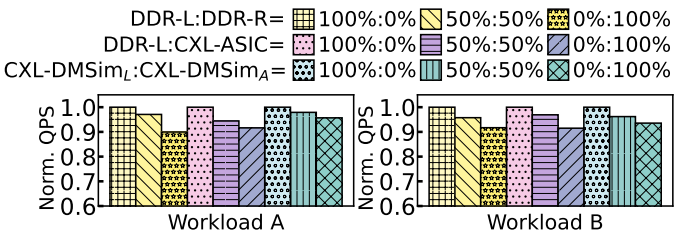


Fig. 13: QPS of Redis with different ratios of pages allocated to DDR-L and DDR-R/CXL-HDM.

*3) Mess Test With Varying Rd/Wr Patterns:* We also conducted tests on our platform using the newly-proposed Mess benchmark [26]. Fig. 12(a) illustrates that DDR-L memory exhibits a marginal variation in bandwidth across different Rd/Wr patterns, whereas Fig. 12(b) demonstrates that CXL-ASIC memory is much more sensitive to Rd/Wr patterns (approximately 3× as marked with red two-way arrows). This discrepancy arises from the fact that the DDR protocol utilizes a bi-directional parallel bus for both reads and writes in a time-multiplexed manner, while the CXL protocol features SerDes TX/RX expressways for separated read and write traffics.

In addition, one can observe that the bandwidth of DDR-L maximizes at the 100% read pattern, and it gradually decreases until reaching at Rd:Wr=50%:50%. In contrast, CXL-ASIC exhibits the best performance at Rd:Wr=74%:26%, instead of 50%:50% as claimed in the Mess work [26]. we argue that this phenomenon is caused by the compromised CXL memory controller design, where the CXL protocol has to be translated to DDR for accessing DRAM storage arrays. This CXL+DDR design not only prolongs memory access paths leading to increased access latency, but also compromises the bandwidth utilization of CXL link and power consumption. Hence, we believe it is necessary to eliminate DDR in the CXL memory controller by directly exposing backend DRAM arrays to the CXL protocol, similar to what NVMe has done to SATA SSDs.

*4) Real-World Application Test:* The benchmarks above show that the CXL-FPGA, operating at a lower frequency, cannot accurately replicate the actual performance of real CXL memories. Due to limitations on space and simulation time, future real-world applications will only use the CXL-ASIC devices. With the above characterization results, we can see that the performance of CXL-FPGA is much poorer than the real CXL-ASIC device. Due to the simulation overhead as well as page limitations, we will limit our real-world application experiments to CXL-ASIC only. To better understand the system performance of CXL-based disaggregated memory, we selected a representative in-memory database named Redis [39] to evaluate the system throughput in diverse application scenarios. The performance of Redis is evaluated using YCSB, a popular and well-known NoSQL and SQL database benchmark suite with six predefined workloads [40].

Fig. 13 shows the performance metric *Queries Per Second* (QPS) of Redis under two workloads with different memory allocation strategies. For both workloads, using only DDR-L memory results in the highest QPS, while 100% CXL or DDR-R memory leads to smallest QPS. This is because mixing poor-performance (prominently in latency as observed previously) memory into DDR-L pulls down the overall performance of Redis. In addition, the QPS achieved with a certain combination of DDR-L and DDR-R is slightly higher than that achieved with the same combination of DDR-L and CXL memories. Redis is a memory-latency-sensitive application which operates with $\mu$s-level precision [25], [45], [46]. For such applications, even a minor allocation of their working set to high-latency memory can lead to significant performance degradation. Therefore, latency-bound applications such as Redis, DDR-R may deliver superior performance compared to CXL-ASIC due to its moderate latency characteristics.

### C. Exploration of CXL Memory Benefits

*1) Memory Capacity Expansion:* Viper is a hybrid PMEM-DRAM key-value database [47]. We used Viper to evaluate the impact of memory expansion strategies on application performance when the local memory capacity is inadequate. To create this scenario meanwhile speeding up our experiments, we limited the DDR-L capacity accessible to Viper to ensure that it is less than the amount of data inserted into Viper. To evaluate the QPS of the system, we inserted an equal number of key-value pairs with two different types: <16,200>(0.216KB) and <100,900>(1KB). The system employs a preferred memory allocation strategy, which attempts to allocate memory from DDR-L first.

Fig. 14 shows the test results of system performance in normalized QPS. When inserting key-value pairs of <16,200>, the QPS remains relatively consistent across various memory configurations. This is because the total data volume inserted into Viper did not exceed the maximum capacity of DDR-L during the execution of the four operations. However, when the size of the inserted key-value pairs increased to <100,900>, the system's QPS with DDR-L alone dropped significantly. The decline is attributed to the total volume of inserted data surpassing the capacity threshold that DDR-L can accommodate. It cannot completely accommodate the data intended to be inserted into Viper, necessitating frequent swapping of pages in and out using swap space by default. It can be seen that expanding memory with CXL memory can enhance system throughput significantly with at most 25 times or 23 times improvement compared to DDR-L alone. This suggests that CXL memory can be an effective means of expanding system memory on top of existing DDR memory when needed, as it does not occupy DIMM slots and simply relies on PCIe 5.0/6.0 x16 interface.

*2) Memory Bandwidth Expansion:* To evaluate the benefits of CXL memory to bandwidth-sensitive applications, we performed inference tasks using Facebook's *Deep Learning Recommendation Model* (DLRM) configured similar to the prior work MERCI [48]. We conducted tests with various memory mixing configurations, covering DDR-L, DDR-R, and CXL HDM with an interleaved memory allocation strategy.

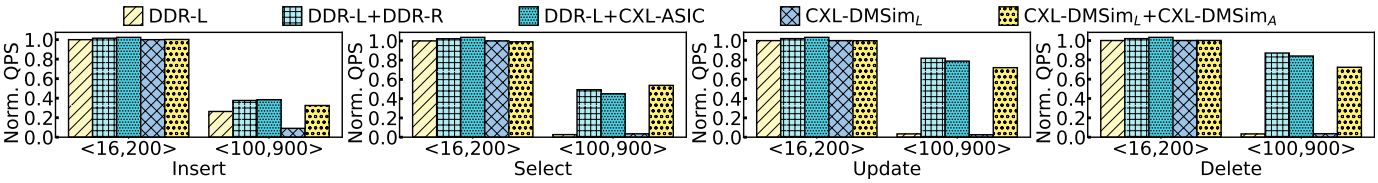


Fig. 14: QPS of Viper with various memory-type expansion strategies for different database operations.

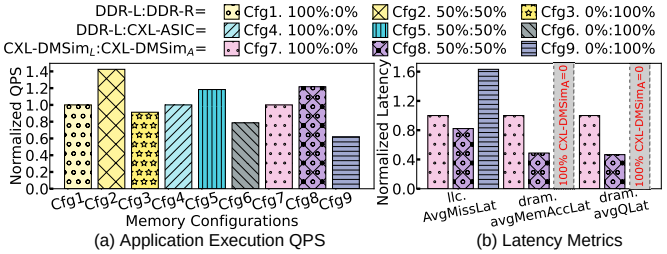


Fig. 15: Impact of memory mixing strategies on the inference performance of the DLRM model.

Fig. 15(a) presents the test results of DLRM inference in various memory mixing scenarios. One can see that the system performance improves 29% and 16% separately when using a 50%:50% mix of DDR-L and DDR-R/CXL-HDM in comparison to the 100% DDR-L case. This is because interleaved memory can effectively expand the bandwidth of the existing DDR memory, thereby increasing the overall system's QPS. When using a 50%:50% mix of DDR-L and CXL HDM, the QPS of the system is lower than the QPS achieved with a 50%:50% mix of DDR-L and DDR-R. This can be explained by the fact that the emulated CXL memory using remote DDR5 offers a relatively higher performance considering the combined effects of both latency and bandwidth when compared to the CXL memory.

The simulation results on CXL-DMSim suggest a similar phenomenon, despite the QPS is slightly lower than its counterpart on real hardware. The variation can be attributed to the fact that the X86O3CPU model we used on CXL-DMSim cannot absolutely match today's advanced Xeon CPU. We further analyzed the root causes of the phenomenon using detailed output statistics of the simulator. In Fig.15(b), the metric *dram.avgQLat* represents the average queue latency per DRAM burst for the DDR memory, whereas *dram.avgMemAccLat* is the average memory access latency per DRAM burst. *l3cache.overallAvgMissLatency* measures the average cache miss latency. The statistics reveal that hybrid DDR and CXL memories deliver the lowest latency across all metrics, due to reduced queuing delays and access contention. Note that in the CXL-ASIC exclusive configuration, there is no DDR DRAM traffic; thus *dram.avgMemAccLat* and *dram.avgQLat* are both 0.

### *D. Congestion Analysis Using CXL-DMSim*

During our experiments, we observed an interesting phenomenon that after a certain point the QPS for MERCI drops as the number of cores continuously increases. Using performance counter monitor (PCM) [49], we found that the average L3 cache miss latency of the 48-core system is $2.4\,\mu$s, whereas the data is $568\,$ns for the 12 cores. We intuitively speculate that under high-concurrent memory access pressure in the 48-core system, the CXL link may experience congestion, resulting in performance degradation. Fortunately, CXL-DMSim's high observability allows us to dive into the system to further investigate what is happening and its root causes.

Table IV shows the QPS and key statistics from CXL-DMSim under different configurations, where stats.1 shows that the aggregate QPS of the 12-core system is twice that of the 48-core system. Stats.2-7 present various latency parameters for a single core. Stats.2 and 3 reveal that in the 48-core system, the average core load-to-use latency and its standard deviation are significantly higher than those in the 12-core system. Stat.4 shows the proportion of load-to-use latency within 0-9 cycles, while stats.5 and stats.6 reflect the minimum and maximum load-to-use latency, respectively. These three metrics suggest that although 94% of the loads in the 48-core system can respond quickly (within 0-9 cycles), a small fraction of the loads exhibit long-tail latency, leading to a larger standard deviation in the load-to-use latency distribution. Stats.7 records core stall events caused by a full load/store queue (LSQ). The data shows that congestion from a small number of load requests in the 48-core system leads to more frequent core stalls, resulting in a lower QPS. In contrast, while only 63% of the loads in the 12-core system can respond quickly, the smaller variance in load-to-use latency suggests a more pipelined system, which results in a higher QPS. We further localized the congestion point on the CXL path. Stats.9 records the number of transmission retries caused by full request/response queues in the bridge module, with results showing that the retry count in the 48-core system is three times higher than the 12-core system. Stats.10 reflects the average response time of the CXL device, showing that the device response times in both systems are nearly identical. In summary, the high degree of parallel memory access pressure in the 48-core system leads to severe queuing in the bridge module, resulting in long-tail latency.

With the above analysis, we speculate that because the CXL link is longer than DDR links, it is more prone to congestion under high memory access pressure, thereby causing the system performance to degrade rapidly. With the help of CXL-DMSim, researchers can easily pinpoint congestion locations and quickly explore various optimization strategies, such as smarter CPU prefetch mechanisms, customized flow control strategies, and more advanced CXL memory controller.

TABLE IV: Statistics from 12-core and 48-core DLRM experiments on the CXL-DMSim$_A$ memory (Cfg9).

| No. | Statistics | 12-Core | 48-Core |
|---|---|---|---|
| 1 | Aggregate QPS | 1.8e6 | 8.8e5 |
| 2 | core.loadToUse::mean (Cycle) | 97 | 198 |
| 3 | core.loadToUse::stdev (Cycle) | 217 | 1107 |
| 4 | core.loadToUse::0-9 (Cycle) | 63.6% | 94.8% |
| 5 | core.loadToUse::min_value (Cycle) | 2 | 2 |
| 6 | core.loadToUse::max_value (Cycle) | 2.7e3 | 2.5e4 |
| 7 | core.lsqFullEvents (Count) | 4.6e6 | 9.0e6 |
| 8 | l3.overallAvgMissLat. (Tick) | 1.3e5 | 1.2e6 |
| 9 | bridge.reqRetryCounts (Count) | 1.8e7 | 6.1e7 |
| 10 | cxl.rsp::mean (Tick) | 8.1e4 | 8.1e4 |

### E. Expandability of CXL-DMSim

The CXL protocol is agnostic to the underlying memory technologies, enabling seamless support for a range of media, including DRAM, Flash, and emerging non-volatile memories. CXL-DMSim also adheres to this principle, allowing for the convenient extension of internal components.

As an example, Fig. 16 shows the device model of CXL-SSD, which is a newly added component to CXL-DMSim. The red dashed blocks highlight the added modules to our CXL device model. In the memory medium module, we utilized an open-source SSD simulator called SimpleSSD [50], [51] to model the SSD backend of the CXL interface. However, two issues must be addressed. First, SSD exhibits significantly higher latency ($\mu$s level) compared to DRAM (ns level). Second, CXL.mem is based on memory semantics at a byte granularity, while SSD operates with I/O semantics at a page (e.g., 4KB) granularity. A simple replacement of the memory medium from DRAM to SSD would inevitably result in a dramatic performance decline. Thus, we designed a cache between the device controller and memory medium to address the aforementioned issues. Additionally, we implemented a cache prefetching and replacement module to manage cache lines. Specifically, we first considered that, due to the larger access granularity of SSDs, prefetching is more likely to result in cache pollution. Our prefetching decisions are based on the history of cache misses, with prefetching only executed when there is a high confidence level that it will yield benefits. Furthermore, given the higher access latency of SSDs, we employ the Best-Offset prefetching algorithm [52], which prioritizes timeliness in retrieving data from the backend SSD. When a memory access request reaches the CXL-SSD, it is directly returned if the SSD cache hits; otherwise, it is translated into SimpleSSD data packets for backend retrieval.

We ran the Viper test on the CXL-SSD to evaluate its performance. Unlike the previous tests, we utilized the AM mode in this case to demonstrate its usage. As illustrated in Fig. 17, the test results indicate that the QPS of CXL-SSD across four operations are considerably lower than CXL-DRAM, due to the higher access latency of Flash medium. To evaluate the effects of cache on system performance, we implemented and compared different cache replacement policies including Least Recently Used (LRU) and First In First Out (FIFO). It can be seen that the QPS of LRU or FIFO

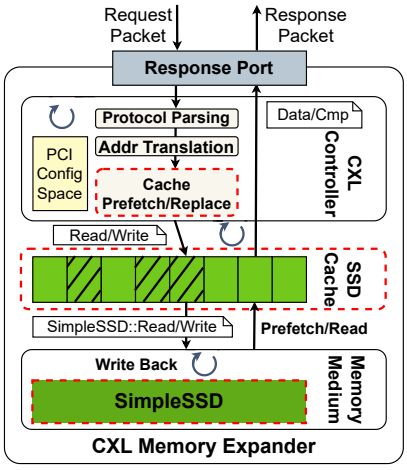


Fig. 16: CXL-SSD device model design.

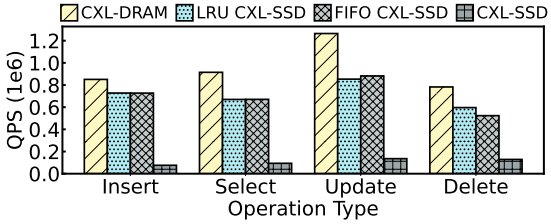


Fig. 17: QPS values of Viper with CXL-DRAM and CXL-SSD for different operations.

CXL-SSD with an additional cache has been significantly enhanced, reaching 72%-88% of CXL-DRAM. The results of this test not only prove the effectiveness of the AM mode, but also demonstrate the good expandability of our simulator. This means that CXL-DMSim can simulate various CXL memory devices integrated with different storage medium, providing a broad exploration and design space for the research of future heterogeneous memory-pooled systems.

## V. Discussion

### A. Simulation Overhead of CXL-DMSim

To assess the simulation overhead covering two aspects which are *simulation speed* and *occupied host memory* introduced by CXL-DMSim in comparison to the raw gem5, we conducted various experiments on our simulator and collected statistics of the execution time (HostSeconds) and physical memory occupation (HostPhyMemory) on the host machine. Fig 18(a) compares the program execution time between raw gem5, CXL-DMSim$_L$, and CXL-DMSim$_A$ with respect to three workloads. The execution time of CXL-DMSim$_L$ is nearly identical to that of raw gem5, while the average execution time of CXL-DMSim$_A$ is 16% higher. This is because CXL-DMSim$_A$ involves more components and longer latency in the simulation process. Notably, LMBench, a benchmark focused on evaluating memory access latency, exhibits slightly higher execution times compared to STREAM and MERCI.

The comparison of HostPhyMemory overhead is shown in Fig 18(b). CXL-DMSim exhibits an average increase of only 3.5% in host memory usage, compared to raw gem5.

TABLE V: Comparison between CXL-DMSim and other CXL system simulators: pros & cons.

| Attribute | CXL-DMSim | QEMU [17] | Mess+gem5 [26] | gem5-cxl [18] | CXLMemSim [19] | Remote NUMA [21]–[23] |
|---|---|---|---|---|---|---|
| CXL protocol support | Yes | Yes | No | No | No | No |
| Full-system CXL support | Yes | Yes | No | No | No | No |
| Silicon validation | Yes | No | No | No | No | N/A |
| Expandability | High | High | Low | Low | Low | Low |
| Sw/hw co-design | Yes | Yes | No | No | No | No |
| Cycle accuracy | Yes | No | Yes | Yes | No | Yes |
| Development maturity | Yes | Yes | Yes | No | No | Yes |
| Configurability | High | High | High | Low | Medium | Low |
| Simulation error | Medium | High | Medium | High | Unknown | High |
| Simulation speed | Low | Medium | Low | Low | Medium | High |

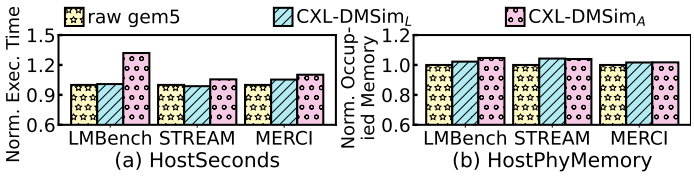


Fig. 18: Simulation overhead in terms of simulation speed and occupied host memory for CXL-DMSim.

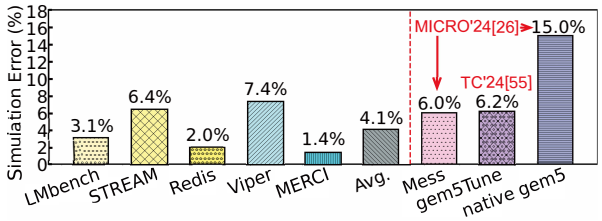


Fig. 19: Simulation error of CXL-DMSim and related simulators in the prior art.

This increase results from the additional memory allocation necessary on the host machine to model the CXL device memory space. Moreover, the specific magnitude of memory overhead is closely correlated with the amount of physical memory utilized by different applications.

### B. Simulation Error of CXL-DMSim

We also analyzed the simulation error stats of CXL-DMSim for a large variety of workloads against the results on our hardware testbed. The collected error data is compared with that of other CXL simulators as depicted in Fig. 19. It can be seen that the minimum simulation error is 1.4% for MERCI while the maximum goes to Viper. The averaged simulation error is measured at 4.1% across all workloads in our experiments. CXL-DMSim exhibits the best accuracy in comparison to the two latest works Mess (6.0%) and gem5Tune (6.2%) as revealed in their respective papers [26], [53]. The native gem5 shows the highest average error of 15.0% [26].

The sources of simulation errors for CXL-DMSim can be attributed to the cumulative effects of modeling inaccuracies in the entire system. These include but not limited to the following. (1) The CPU model lags behind of the latest CPU we used on our testbed. (2) The poor modeling of disk in gem5 when storage layers are involved, as can be observed with the increased errors for Viper when some data was pushed out to disk. Note that it is always a dilemma in pursuit of simulation accuracy and speed.

### C. CXL-DMSim vs. Other Simulators

Table V compares CXL-DMSim and other prominent simulators such as QEMU, Mess, gem5-CXL, CXLMemSim and remote NUMA at various aspects. First, CXL-DMSim and QEMU offer comprehensive support for CXL protocol, which is critical for accurately simulating the behavior of real CXL memory systems. In contrast, the other four simulators actually do not support CXL protocol. Second, CXL-DMSim is a full-system CXL simulator that provides the most realistic interaction between the operating system and simulated hardware, enabling a range of opportunities for software-hardware co-design. In contrast, Mess, gem5-CXL and CXLMemSim cannot boot the operating system that supports CXL devices. Third, CXL-DMSim is the only one which has undergone extensive silicon validation, surpassing the other simulators in terms of accuracy and realism. CXL-DMSim also excels in expandability, cycle accuracy, development maturity, configurability, with acceptable simulation error. Nevertheless, these advantages come along at the cost of slow simulation speed, high modeling complexities for new features, and limited system scale.

### D. Future Work

Currently, CXL-DMSim only supports single-host simulation mode. Plans are on our schedule to enhance its capabilities to support multi-hosts via CXL switches with various CXL topologies, akin to dist-gem5 [54]. We are actively developing and integrating the CXL.cache sub-protocol into CXL-DMSim. Furthermore, we will leverage CXL-DMSim to explore heterogeneous memory management; e.g., efficient allocation and release of memory capacity and bandwidth within a CXL memory pool with tied and hybrid memory resources. These days, many researchers claim that the deployment of CXL-disaggregated memory in data centers offers both performance improvements and cost reductions. But, forecasting TCO savings and calculating Return on Investment (ROI) for the CXL technology are complex. We believe that leveraging CXL-DMSim for cost modeling, system efficiency evaluation, and optimization in a multi-host CXL-disaggregated memory environment are all interesting research topics.

## VI. Conclusion

In this paper, we have presented CXL-DMSim, an open-source and silicon-calibrated full-system simulator for CXL disaggregated memory systems. CXL-DMSim can be used

either in a NUMA-compatible kernel-managed mode or in an app-managed mode depending on users' preference. Our experimental results on both hardware and CXL-DMSim suggest that CXL memory is very effective in memory expansion of both capacity and bandwidth to boost system performance. We also found the current CXL memory controller design composed of a CXL protocol controller and a DDR controller compromises both the latency and bandwidth of CXL memory and also leads to a high sensitivity to Rd/Wr patterns. This calls for a CXL-oriented memory controller design without DDR to fully unleash the performance of CXL links. With CXL-DMSim, the system performance bottlenecks can be quickly identified thanks to its high observability with substantial simulation statistics. In the future, we will continue to enhance the capabilities of CXL-DMSim, and we also welcome the community to join us to build a solid simulation platform for architectural research on memory-pooled computing systems.

## REFERENCES


[1] Meta, "Reimagining memory expansion for single socket servers with cxl," https://146a55aca6f00848c565-a7635525d40ac1c70300198708936b4e.ssl.cf1.rackcdn.com/images/fa0cc66ccd41ff51dcbb4a7b5b311c8e338b482a.pdf.

[2] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *NSDI*, 2017, pp. 649–667, doi:10.5555/3154630.3154683.

[3] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: CXL-based memory pooling systems for cloud platforms," in *ASPLOS*, 2023, pp. 574–587, doi:10.1145/3575693.3578835.

[4] H. Al Maruf and M. Chowdhury, "Memory disaggregation: advances and open challenges," *ACM SIGOPS Operating Systems Review*, pp. 29–37, 2023, doi:10.1145/3606557.3606562.

[5] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, "Remote regions: a simple abstraction for remote memory," in *USENIX ATC*, 2018, pp. 775–787, doi:10.5555/3277355.3277430.

[6] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *USENIX ATC*, 2015, pp. 291–305, doi:10.5555/2813767.2813789.

[7] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating persistent memory and controlling them remotely: an exploration of passive disaggregated key-value stores," in *USENIX ATC*, 2020, pp. 33–48, doi:10.5555/3489146.3489149.

[8] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "AIFM: High-performance, application-integrated far memory," in *OSDI*, 2020, pp. 315–332, doi:10.1145/3606557.3606562.

[9] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, high-performance memory disaggregation with DirectCXL," in *USENIX ATC*, 2022, pp. 287–294, https://www.usenix.org/system/files/atc22-gouk.pdf.

[10] A. Geyer, D. Ritter, D. H. Lee, M. Ahn, J. Pietrzyk, A. Krause, D. Habich, and W. Lehner, "Working with disaggregated systems. what are the challenges and opportunities of RDMA and CXL?" *Datenbanksysteme für Business, Technologie und Web*, pp. 751–755, 2023, doi:10.18420/BTW2023-47.

[11] Y. Guo and G. Li, "A CXL-powered database system: Opportunities and challenges," in *IEEE International Conference on Data Engineering*, 2024, pp. 5593–5604, doi:10.1109/ICDE60146.2024.00447.

[12] OpenCAPI Consortium, "OpenCAPI specification," https://computeexpresslink.org/resource/opencapi-specification-archive/.

[13] Gen-Z Consortium, "Gen-Z specification," https://computeexpresslink.org/resource/gen-z-specification-archive/.

[14] CCIX Consortium, "CCIX specification," https://computeexpresslink.org/resource/ccix-specification-archive/.

[15] NVIDIA, "What is NVIDIA NVLink," https://blogs.nvidia.com/blog/what-is-nvidia-nvlink/.

[16] CXL Consortium, "Compute express link specification," https://www.computeexpresslink.org/download-the-specification.

[17] T. Q. P. Developers, "QEMU," https://www.qemu.org/docs/master/system/devices/cxl.html.

[18] L. Wang, X. Zhang, T. Lu, and M. Chen, "gem5-CXL," https://github.com/zxhero/gem5-CXL?tab=readme-ov-file.

[19] Y. Yang, P. Safayenikoo, J. Ma, T. A. Khan, and A. Quinn, "CXLMemSim: A pure software simulated CXL.mem for performance characterization," *ArXiv*, 2023, doi:10.48550/arXiv.2303.06153.

[20] Yiwei Yang and Pooneh Safayenikoo and Jiacheng Ma and Tanvir Ahmed Khan and Andrew Quinn, "CXLMemSim," unpublished.

[21] Y. Fridman, S. Mutalik Desai, N. Singh, T. Willhalm, and G. Oren, "CXL memory as persistent memory for disaggregated HPC: A practical approach," in *International Conference for High Performance Computing, Networking, Storage, and Analysis Workshops*, 2023, pp. 983–994, doi:10.1145/3624062.3624175.

[22] M. Arif, K. Assogba, M. M. Rafique, and S. Vazhkudai, "Exploiting CXL-based memory for distributed deep learning," in *ICPP*, 2023, pp. 1–11, doi:10.1145/3545008.3545054.

[23] K. Song, J. Yang, S. Liu, and G. Pekhimenko, "Lightweight frequency-based tiering for cxl memory systems," *ArXiv*, 2023, doi:10.48550/arXiv.2312.04789.

[24] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "TPP: transparent page placement for CXL-enabled tiered-memory," in *ASPLOS*, 2023, pp. 742–755, doi:10.1145/3582016.3582063.

[25] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim, "Demystifying CXL memory with genuine CXL-ready systems and devices," in *MICRO*, 2023, pp. 105–121, doi:10.1145/3613424.3614256.

[26] P. Esmaili-Dokht, F. Sgherzi, V. Soldera Girelli, I. Boixaderas, M. Carmin, A. Monemi, A. Armejach, E. Mercadal, G. Llort, P. Radojkovic, M. Moreto, J. Gimenez, X. Martorell, E. Ayguade, J. Labarta, E. Confalonieri, R. Dubey, and J. Adlard, "A mess of memory system benchmarking, simulation and application profiling," in *MICRO*, 2024, pp. 1–18, http://export.arxiv.org/abs/2405.10170.

[27] gem5 community, "gem5," https://www.gem5.org/.

[28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, pp. 1–7, 2011, doi:10.1145/2024716.2024718.

[29] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, "The gem5 simulator: Version 20.0+," *ArXiv*, 2020, doi:10.48550/arXiv.2007.03152.

[30] Intel, "Intel CXL IP," https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/cxl-ip.html.

[31] D. D. Sharma, R. G. Blankenship, and D. S. Berger, "An introduction to the compute express link (CXL) interconnect," *ACM Computing Surveys*, 2023, doi:10.48550/arxiv.2306.11227.

[32] J. Liu, X. Wang, J. Wu, S. Yang, J. Ren, B. Shankar, and D. Li, "Exploring and evaluating real-world CXL: Use cases and system adoption," 2024, doi:10.48550/arxiv.2405.14209.

[33] H. Ham, J. Hong, G. Park, Y. Shin, O. Woo, W. Yang, J. Bae, E. Park, H. Sung, E. Lim, and G. Kim, "Low-overhead general-purpose near-data processing in CXL memory expanders," in *MICRO*, 2024, doi:10.5281/ZENODO.13283894.

[34] D. D. Sharma, "Novel composable and scaleout architectures using Compute Express Link," *IEEE Micro*, pp. 9–19, 2023, doi:10.1109/MM.2023.3235972.

[35] P. Levis, K. Lin, and A. Tai, "A case against CXL memory pooling," in *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, 2023, pp. 18–24, doi:10.1145/3626111.3628195.

[36] Intel, "Intel memory latency checker," https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[37] C. S. Larry McVoy, "LMbench," https://lmbench.sourceforge.net/.

[38] J. McCalpin, "STREAM," https://www.cs.virginia.edu/stream/ref.html.

[39] Redis Ltd., "Redis," https://redis.io/.

[40] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput. (SOCC)*, 2010, pp. 143–154, doi:10.1145/1807128.1807152.

[41] G. Wu, P. Huang, and X. He, "Reducing SSD access latency via NAND flash program and erase suspension," *Journal of Systems Architecture*, pp. 345–356, 2014, doi:10.1016/j.sysarc.2013.12.002.

[42] K. Kim and T. Kim, "HMB in DRAM-less NVMe SSDs: Their usage and effects on performance," *PLoS ONE*, p. e0229645, 2020, doi:10.1371/journal.pone.0229645.

[43] C. S. Larry McVoy, "LMbench," https://github.com/foss-for-synopsys-dwc-arc-processors/lmbench/blob/master/scripts/config-run.

[44] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *FPGA*, 2006, pp. 21–30, doi:10.1145/1117201.1117205.

[45] Y. Zhong, D. S. Berger, C. Waldspurger, R. Wee, I. Agarwal, R. Agarwal, F. Hady, K. Kumar, M. D. Hill, M. Chowdhury, and A. Cidon, "Managing memory tiers with CXL in virtualized environments," in *OSDI*, 2024, pp. 37–56, https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong.

[46] M. Ahn, T. Willhalm, N. May, D. Lee, S. M. Desai, D. Booss, J. Kim, N. Singh, D. Ritter, and O. Rebholz, "An examination of CXL memory use cases for in-memory database management systems using SAP HANA," *VLDB*, 2024, doi:10.14778/3685800.3685809.

[47] L. Benson, H. Makait, and T. Rabl, "Viper: an efficient hybrid PMem-DRAM key-value store," *Proc. VLDB Endow.*, pp. 1544–1556, 2021, doi:10.14778/3461535.3461543.

[48] Y. Lee, S. H. Seo, H. Choi, H. U. Sul, S. Kim, J. W. Lee, and T. J. Ham, "MERCI: efficient embedding reduction on commodity hardware via sub-query memoization," in *ASPLOS*, 2021, pp. 302–313, doi:10.1145/3445814.3446717.

[49] Intel, "Intel performance counter monitor," https://www.intel.cn/content/www/cn/zh/developer/articles/tool/performance-counter-monitor.html.

[50] CAMELab, "SimpleSSD," https://docs.simplessd.org/en/v2.0.12/.

[51] M. Jung, J. Zhang, A. Abulila, M. Kwon, N. Shahidi, J. Shalf, N. S. Kim, and M. Kandemir, "SimpleSSD: modeling solid state drives for holistic system simulation," *IEEE Comput. Archit. Lett.*, pp. 37–41, 2018, doi:10.1109/LCA.2017.2750658.

[52] P. Michaud, "Best-offset hardware prefetching," in *HPCA*, 2016, pp. 469–480, doi:10.1109/HPCA.2016.7446087.

[53] Y. Qiu, T. Huang, Y. Tang, Y. Liu, Y. Kong, X. Yu, X. Zeng, and Y. Fan, "Gem5Tune: A parameter auto-tuning framework for gem5 simulator to reduce errors," *IEEE Trans. Comput.*, pp. 902–914, 2023, doi:10.1109/TC.2023.3347675.

[54] A. Mohammad, U. Darbaz, G. Dozsa, S. Diestelhorst, D. Kim, and N. S. Kim, "dist-gem5: distributed simulation of computer clusters," in *ISPASS*, 2017, pp. 153–162, doi:10.1109/ISPASS.2017.7975287.

## APPENDIX

### A. Abstract

The artifact consists of three components: 1) the complete CXL-DMSim source code based on gem5 v23.1; 2) a Linux kernel with the added CXL expander driver; 3) a disk image containing test programs. To initiate the full-system simulation of CXL-DMSim, the Linux kernel and disk image need to be loaded. This appendix describes the process of compiling and installing our simulator, CXL-DMSim, and reproducing the results shown in Fig. 10 and Fig. 15 in this paper. The code must be executed on a Linux system with at least 64GB of main memory and 50GB of disk space.

### B. Artifact Checklist (meta-information)

- **Program:** We provide a disk image parsec.img that includes benchmarks LMBench and DLRM.
- **Compilation:** GCC/G++ 9.4.0 and Python3.8.
- **Binary:** The Linux 5.4.49 kernel binary with added CXL memory expender driver support.
- **Run-time environment:** Ubuntu 20.04 or Ubuntu 22.04.
- **Hardware:** Intel x86-64
- **Output:** Files with the results of the program's execution.
- **Experiments:** Manual invocation of scripts, which launch corresponding experiments and generate outputs in designated folders.
- **How much disk space required (approximately)?:** 40–50GB
- **How much time is needed to prepare workflow (approximately)?:** The compilation for CXL-DMSim takes approximately 10-20 minutes, and the full-system bootup for CXL-DMSim takes about 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** We primarily control memory allocation on DDR-L, CXL memory, or interleave both using the numactl. For a single test of a specific memory allocation, LMBench requires 5 hours and Merci requires 20 hours. You can accelerate the testing progress by executing as many tests in parallel as possible, based on the size of the machine's main memory.
- **Publicly available?:** Yes
  - CXL-DMSim: https://github.com/ferry-hhh/CXL-DMSim.
  - Kernel and disk image: https://drive.google.com/drive/folders/1sxZBsedT19ntJdzN8MTkcbczMGXNXgrM?usp=sharing.
- **Code licenses (if publicly available)?:** BSD 3-Clause "New" or "Revised" License

### C. Description

*1) How to access:* All the source code of CXL-DMSim is available on GitHub, and the linux Kernel and disk image can be downloaded from GoogleDrive.

*2) Hardware dependencies:* It is recommended to run this on a machine with a sufficiently large main memory (>64GB) because modeling the operations of 48 O3 CPUs along with the CXL memory expender requires a considerable amount of memory.

*3) Software dependencies:* Linux systems that support building gem5. Ubuntu 20.04 and Ubuntu 22.04 have been verified to run CXL-DMSim successfully. The GCC version used is 9.4.0, and the Python version is 3.8.10.

### D. Installation

1) First, run the following command on Ubuntu 20.04 to install the required dependencies:

```
$ sudo apt install build-essential git m4
  scons zlib1g zlib1g-dev libprotobuf-dev
  protobuf-compiler libprotoc-dev
  libgoogle-perftools-dev python3-dev
  python-is-python3 libboost-all-dev
  pkg-config
```

2) Download the CXL-DMSim source code, linux kernel and disk image from the provided links.
3) Run the following command to compile:

```
$ scons build/X86/gem5.opt -j{cpus}
```

Note that the '-j' flag is optional and enables parallelization of the compilation process, with 'cpus' specifying the number of threads to use. It is recommended to use as many threads as possible to accelerate the compilation.

### E. Experiment Workflow

Next, we will proceed to complete the experiments for Fig. 10 and Fig. 15.

1) Before starting, please check the correct paths for the `kernel` and `disk_image` within the file `configs/example/gem5_library/x86-cxl-run.py` on your local machine.
2) At the terminal prompt, start the test script with the following command.

```
$ ./run_fs.sh
```

The script will sequentially execute the LMBench and DLRM tests. However, you may also manually duplicate each test command within `run_fs.sh` to execute them in parallel, if the memory capacity of your host machine permits.

### F. Evaluation and Expected Results

Upon execution of the command to initiatethe the full system, the system runtime logs can be found within the `board.pc.com_1.device` file located in the corresponding test folder under the `output` directory. Subsequently, the experimental results will also be appended to this file. We provide the expected data for each test in the `expected_result` directory of the GitHub repository for your reference.

### G. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html