Cohet: A CXL-Driven Coherent Heterogeneous Computing Framework with Hardware-Calibrated Full-System Simulation

Yanjing Wang*† Lizhou Wu*† Sunfeng Gao† Yibo Tang† Junhui Luo†
Zicong Wang† Yang Ou† Dezun Dong† Nong Xiao✉§† Mingche Lai✉†
†College of Computer Science and Technology, National University of Defense Technology, China
§School of Computer Science and Engineering, Sun Yat-sen University, China
{wangyanjing, lizhou.wu, mingchelai}@nudt.edu.cn, xiaon6@mail.sysu.edu.cn

Abstract*—Conventional heterogeneous computing systems built on PCIe interconnects suffer from inefficient fine-grained host-device interactions and complex programming models. In recent years, many proprietary and open cache-coherent interconnect standards have emerged, among which compute express link (CXL) prevails in the open-standard domain after acquiring several competing solutions. Although CXL-based coherent heterogeneous computing holds the potential to fundamentally transform the collaborative computing mode of CPUs and XPUs, research in this direction remains hampered by the scarcity of available CXL-supported platforms, immature software/hardware ecosystems, and unclear application prospects. This paper presents Cohet, the first CXL-driven coherent heterogeneous computing framework. Cohet decouples the compute and memory resources to form unbiased CPU and XPU pools which share a single unified and coherent memory pool. It exposes a standard malloc/mmap interface to both CPU and XPU compute threads, which share a single per-process page table for user applications, leaving the OS dealing with smart memory allocation, page auto-migration, and management of heterogeneous resources. This design significantly simplifies heterogeneous parallel programming to a level comparable to homogeneous programming. To facilitate Cohet research, we also present a full-system cycle-level simulator named SimCXL, which is capable of modeling all CXL sub-protocols and device types. SimCXL has been rigorously calibrated against a real CXL testbed with various CXL memory and accelerators, showing an average simulation error of 3%. Our evaluation reveals that CXL.cache reduces latency by 68% and increases bandwidth by 14.4× compared to DMA transfers at cacheline granularity. Building upon these insights, we demonstrate the benefits of Cohet with two killer apps, which are remote atomic operation (RAO) and remote procedure call (RPC). Compared to PCIe-NIC design, CXL-NIC achieves a 5.5 to 40.2× speedup for RAO offloading and an average speedup of 1.86× for RPC (de)serialization offloading.

I. Introduction

With the end of Dennard scaling and slowdown of Moore's law, homogeneous CPU architectures can no longer satisfy the explosive compute and memory demands of emerging data-driven workloads such as AI/ML and big data analytics [1]. This gap has spurred widespread adoption of heterogeneous systems that integrate CPUs with various XPUs (e.g., GPUs, SmartNICs/DPUs, and FPGAs) over high-speed interconnects such as PCIe to exploit their domain-specific accelerating capabilities [2]–[8], thus boosting overall computing performance. However, the conventional PCIe-based heterogeneous architecture suffers from two major limitations [1], [9]–[14]. First, PCIe favors occasional host-device transfers of bulky data sets, making it well suited to bandwidth-intensive coarse-grained workloads. In contrast, latency-sensitive applications that require frequent or fine-grained CPU-XPU interactions are severely hindered by PCIe's high latency and poor bandwidth in small data transfers. Second, PCIe is a noncoherent interconnect that has to rely on complex software stacks and significant programmer efforts to explicitly or implicitly manage data movement between host and accelerator, leading to a cumbersome programming model and poor development productivity.

In recent years, both proprietary [1], [15]–[17] and open [18]–[21] coherent interconnect technologies have emerged to address the limitations of PCIe. Proprietary solutions (e.g., NVIDIA's NVLink-C2C [15], AMD's Infinity Fabric [16], and Intel's UPI [17] and CMI [1]) deliver impressive performance, but lack hardware flexibility, software ecosystem diversity, and user code portability. Among open standards, CXL [18] has become the de facto industry standard with over 240 active consortium members [22] after consolidating competing proposals OpenCAPI [19], Gen-Z [20], and CCIX [21]. Building upon the PCIe physical layer, CXL defines three sub-protocols: CXL.io, CXL.cache, and CXL.mem. To date, the majority of academic and industrial research works are focused on CXL-based disaggregated memory systems merely using CXL.mem [23]–[28], which promise clear benefits in decoupling memory from compute and enabling independent scaling and sharing of memory resources. However, we contend that there is tremendous potential in exploiting CXL.cache together with CXL.mem to realize fully coherent heterogeneous computing. Such an approach can overcome the bottlenecks of the conventional heterogeneous architecture and fundamentally reshape the future cooperative computing paradigm.

Despite its promise, CXL-based coherent heterogeneous computing faces three key challenges. First, the CXL hardware

*Co-first author. ✉Co-corresponding author.

and software ecosystem remains immature. On the hardware side, CXL memory expanders have only reached early commercial availability [29]–[33], and CXL-supported accelerators remain largely confined to FPGA platforms [34], [35] in experimental testbeds. On the software side, the OS kernel lacks native support for CXL accelerators [36], and higher-level heterogeneous programming frameworks must adapt to CXL cache coherence and memory semantics [37], [38]. Second, academic researchers suffer from a dearth of realistic CXL platforms. Previous works often used a remote NUMA node as an emulator of CXL accelerators [11], [39], [40], which can introduce inaccuracies and misleading conclusions in some cases [9]. Finally, owing to the lack of effective evaluation and comprehensive analysis of CXL accelerators, the specific application scenarios benefiting from coherent heterogeneous computing remain unexplored.

To address these challenges, this paper presents Cohet, the first CXL-driven coherent heterogeneous computing framework that is intended to deliver a software-hardware co-design solution targeting the open-source community. By interconnecting hosts and accelerators over CXL with cache-coherence, Cohet unifies their physical address spaces into a coherent memory pool. Cohet also provides a single shared memory view and a unified per-process page table to both CPUs and XPUs. The OS recognizes CPUs and XPUs as separate NUMA nodes, while providing standard malloc/mmap interfaces to programmers; memory allocation, page migration, and coherence management remain transparent. Moreover, we have developed SimCXL, an open-source full-system cycle-level simulator extended from gem5 that models all three CXL sub-protocols and three types of CXL devices. SimCXL has been rigorously calibrated against a real CXL-supported testbed with Intel FPGA accelerators and various memory expanders, achieving an average simulation error of 3%. Our evaluation shows that at cacheline granularity, CXL.cache reduces latency by 68% and increases bandwidth by 14.4× compared to DMA. To demonstrate the benefits of Cohet, we present a CXL-NIC design with two acceleration case-studies on SimCXL: 1) *remote atomic operation* (RAO), a key building block for distributed parallel computing [41]–[47], and 2) *remote procedure call* (RPC), a fundamental inter-process communication layer in modern cloud microservices [48]–[53]. Our experimental results suggest that CXL-based RAO outperforms its PCIe-based counterpart by 5.5 to 40.2× across different operation types, and that CXL-based RPC achieves an average 1.86× speedup in (de)serialization compared to PCIe-based RPC.

The main contributions are summarized as follows.

- A Cohet computing framework, which brings an optimized collaborative mode, decentralized control and synchronization, and a simplified programming model.
- A full-system cycle-level simulator named SimCXL, with complete support for all three CXL sub-protocols and three types of CXL devices.
- Comprehensive characterization of the latency and bandwidth of CXL.cache transactions; the measurements real new insights and are utilized to calibrate SimCXL.
- CXL-NIC design with two killer-app demonstrations; RAO for distributed parallel computing and RPC for inter-process communication in cloud microservices, demonstrating substantial speedups over PCIe-NIC.

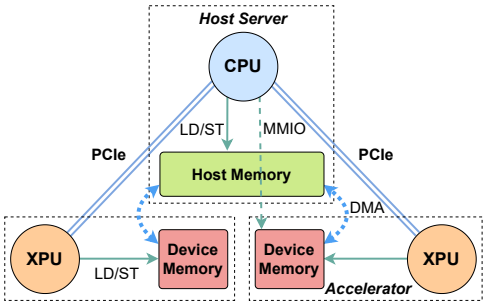


Fig. 1: PCIe-based heterogeneous computing architecture with CPU-biased computing and discrete physical memories.

II. Background

A. Conventional Heterogeneous Computing

Fig. 1 shows a typical PCIe-based heterogeneous architecture, with its CPU-XPU interactive interface and collaborative computing mode elaborated as follows.

1) CPU-XPU interactive interface: In PCIe-based systems, host and device memories reside in separate and non-coherent address spaces, with interactions realized by MMIO and DMA. MMIO exposes the base address registers (BARs) of a peripheral XPU device in the host address space, enabling the CPU to launch uncached loads and stores to the device BAR space. However, PCIe's high per-transaction latency and strict write-ordering, which allows only one outstanding write, limit the MMIO performance [1]. On the XPU side, its DMA engine allows to directly access the host memory and delivers a much higher throughput for bulky contiguous transfers. But, DMA operations incur substantial per-transfer setup overhead, making fine-grained and randomly distributed data movements highly inefficient [9], [54].

2) Collaborative computing mode: Conventional heterogeneous computing units employ a descriptor-driven producer-consumer model in which the CPU takes charge of the control plane for task dispatch, while the XPU serves as the computing plane for accelerated computing. This mode is used by today's mainstream XPUs, including GPUs, SmartNICs, and FPGAs, and its core process comprises a four-stage pipeline. Stage 1: The CPU allocates a DMA-safe memory region (e.g., pinned memory), copies the input data there, encapsulates the task in a structured descriptor and enqueues it, then rings a doorbell register via MMIO to notify the XPU of a new task. Stage 2: The XPU polls the doorbell and invokes its DMA engine to retrieve the pending descriptor from the host memory. Stage 3: Guided by the descriptor, the XPU issues DMA reads to fetch the input data, then performs the specified computation task. Stage 4: Upon completion, the XPU writes the results back to the host memory via DMA and then raises an event to notify the CPU of the results. Although this decoupled approach

maximizes throughput for transferring bulky contiguous data, the isolated address spaces on the CPU and XPU inevitably incur unnecessary data copies and hinder efficient sharing of complex data structures between compute units, thus significantly limiting collaborative computing efficiency.

B. Compute Express Link

Built on the PCIe physical layer, CXL comprises three sub-protocols: CXL.io provides PCIe-equivalent features; CXL.cache enables devices to coherently access host memory; CXL.mem allows the host to perform load/store operations directly on device-attached memory. By combining these sub-protocols, three device types are distinguished. Type-1 devices implement CXL.io and CXL.cache (e.g., SmartNICs without device memory). Type-2 devices support all three sub-protocols (e.g., GPUs with device memory and specialized compute units). Type-3 devices use CXL.io and CXL.mem as memory expanders.

This paper focuses on type-1/2 devices, both of which can be recognized as accelerators (XPUs). Each device includes a *host memory cache* (HMC) that caches host memory and acts as a peer to the CPU's L2 cache, with the CPU's last-level cache (LLC) serving as a coherence synchronization point. The host tracks coherence among peer caches, while the CXL device interacts with the host over the *device coherency engine* (DCOH) using a lightweight MESI protocol. In addition to standard load/store operations, CXL.cache introduces the *non-cacheable push* (NC-P) instruction [35], which allows a device to push a cacheline directly into the host LLC and invalidate it in the HMC, providing a low-latency path for returning computed results. Sections V-B will demonstrate how NC-P optimizes acceleration to boost application performance.

III. Coherent Heterogeneous Computing Framework

A. Limitations of Conventional Heterogeneous Computing

Traditional PCIe-based heterogeneous systems are becoming increasingly crippled in dealing with modern workloads such as LLM training/inference and big data analytics. We identify four fundamental limitations as follows.

L1: Compute-memory coupling leads to resource under-utilization. Current compute units, regardless of CPUs or XPUs, are tightly coupled to their own memory, as illustrated with the dotted boxes in Fig. 1. These host servers and accelerators form basic performance scaling units, leading to either compute or memory over-provisions in today's hyperscale datacenters [27], [55]. In addition, the distributed physical memories remain mutually isolated, resulting in mandatory data copies and infeasibility of direct sharing of complex data structures in heterogeneous computing.

L2: Inefficient collaborative mode. The four-stage interaction model is optimized for processing bulky and contiguous data streams, but it incurs severe latency penalties under frequent fine-grained or random access patterns. This forces heterogeneous computing units to operate in isolation, preventing efficient collaboration in accomplishing complicated computation tasks.

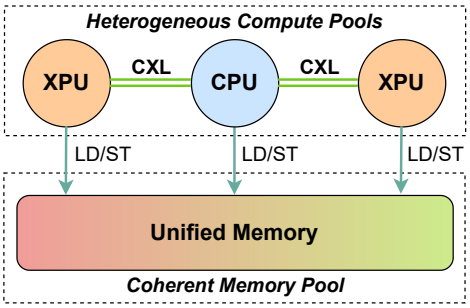


Fig. 2: Emerging CXL-based coherent heterogeneous computing architecture with unbiased cooperative computing and a coherent unified memory pool.

L3: CPU suffers from accelerator tax. The CPU-centric master-slave computing paradigm forces all control and synchronization through the CPU. The CPU overheads, also referred to as accelerator tax, in task dispatch and result collection may offset the raw acceleration gains.

L4: Complex programming model. The lack of hardware coherence results in complex programming models that depend on explicit/implicit memory management by programmers/software, further constrained by limited device memory capacity. Developers face a fundamental dilemma: manual data placement might yield high performance but impose significant programming efforts, whereas relying on software-managed unified memory mechanisms (e.g., CUDA unified memory (UM) [13]) simplifies development at the risk of performance loss due to expensive page fault handling.

B. Cohet Design Philosophy

To overcome the limitations of conventional heterogeneous computing and effectively leverage heterogeneous resources, CXL-based coherent heterogeneous computing offers a promising solution forward. Fig. 2 illustrates its architecture with the following four potential solutions.

S1: Resource disaggregation & pooling. CXL decouples compute and memory resources, enabling the construction of heterogeneous compute and coherent memory pools. These disaggregated compute and memory resources can be expanded on demand, thus greatly boosting utilization. A unified memory view with hardware-maintained coherence helps CPUs and XPUs directly share large-scale data sets, avoiding unnecessary deep copies of working data and the associated software overhead.

S2: Collaborative mode optimization. CXL optimizes the interaction mode between CPUs and XPUs by providing fine-grained, low-latency memory-semantic accesses in both the CPU-to-XPU (CXL.mem) and XPU-to-CPU (CXL.cache) directions. Looking ahead, this capability sets the stage for the unification of two complementary communication mechanisms, with CXL.mem/cache handling latency-sensitive operations while CXL.io serving throughput-oriented transfers.

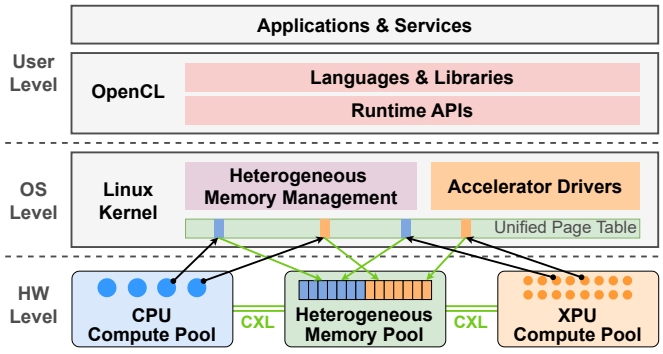


Fig. 3: Vertical stack of the Cohet computing framework.

S3: Decentralized control and synchronization. CXL promotes XPUs that are traditionally seen as subordinate devices to a peer position to CPUs. With access to coherent memory, CPUs and XPUs can now coordinate tasks and share data efficiently, without routing all controls to CPUs. Hardware-supported atomic operations across diverse devices replace traditional lock-based or software-coordinated synchronization, improving the performance of shared data workflows.

S4: Programming model simplification. Hardware-maintained coherence enables the OS to recognize heterogeneous compute units as standard NUMA nodes, providing unified memory allocation interfaces (e.g., malloc, mmap, C++ new), and allowing compute units to overcommit memory beyond the size of system memory. By simplifying heterogeneous programming to homogeneous while maintaining performance, this approach breaks through the developer dilemma in conventional heterogeneous computing. Moreover, existing applications can benefit seamlessly from high-performance heterogeneous acceleration without any refactoring or porting.

C. Cohet Computing Framework

In light of the above four guidelines, we propose Cohet, an open coherent heterogeneous computing framework. As shown in Fig. 3, the vertical stack of Cohet is organized into three levels: hardware, operating system (OS), and user.

1) Hardware level: One or more CXL switches compose a CXL fabric. A distributed resource scheduler (fabric manager) is implemented in each switch to allocate/release fabric-attached memory and XPU resources to a specific host. For each host (OS point of view), its CPUs and XPUs interconnect via the CXL fabric to form a unified physically-addressed heterogeneous memory pool, presenting a single memory view to all compute units. CXL's hardware-maintained coherence not only allows CPUs and XPUs to cache and access each other's memory at cacheline granularity but also supports cross-device atomic operations, enabling fine-grained and scalable synchronization across all running threads in the system. In this CXL-enabled architecture, the address translation service (ATS) lets CPUs and XPUs share a single per-process page table (denoted as Unified Page Table in the figure). When an XPU thread accesses a virtual address, it first looks up the mapping in its device-side address translation cache (ATC), analogous to the host TLB. Upon an ATC miss, the request is forwarded to the CPU-side IOMMU, which performs a page-table walk to resolve the physical address. The resolved mapping entry is then returned to the ATC, after which the XPU proceeds with its memory accesses [18]. At this level, tailoring the microarchitecture of endpoint devices (e.g., XPUs and memory expanders) for target applications is also crucial for maximizing the benefits of the coherent interconnect, as demonstrated with our CXL-NIC designs for accelerating two killer-apps in Section V.

2) OS level: The Linux kernel recognizes CPUs and XPUs as separate NUMA nodes. Heterogeneous memory management (HMM) merges device memory with host memory into the system memory pool, maintains the unified page table, and exposes a standard mmap/malloc interface to upper-layer software [56]. Specifically, the device driver probes the device memory size during system initialization, registers a device instance with HMM, and implements the callback functions required for unified page-table management, such as handling ATC invalidations and page migrations. Furthermore, we modify the kernel's *numa_init* routine, which inspects the available system memory, initializes the host and device memory as distinct NUMA nodes based on their types, and binds them to the corresponding CPU or XPU.

A malloc call allocates a page-table entry without assigning a physical frame, allowing memory overcommitment. On an XPU's first access to a given virtual address, an ATC miss triggers an IOMMU translation request. The kernel then updates the page-table entry to point to XPU physical memory. Once pages are allocated, both CPUs and XPUs access them at cacheline granularity under CXL's coherence guarantee. By contrast, PCIe-based non-coherent interconnects incur high-overhead OS page-fault interrupts and DMA-based page copies, which prior studies identify as the primary performance bottleneck in frameworks such as UM [57]. When the unified page table is about to be updated due to page migration or swapping, HMM invokes the registered driver callback. The driver then temporarily blocks the device from accessing the affected page-table entries, allowing HMM to safely perform the update and trigger the IOMMU invalidation process. According to the ATS protocol, this process invalidates the corresponding entries in the device-side ATC. Once the invalidation has been completed, HMM notifies the driver to resume device address translation. Moreover, adaptive page migration in Cohet is a potential performance optimization, left for future work.

3) User level: Cohet embraces OpenCL 3.0 [37], the most pervasive cross-vendor open standard for low-level heterogeneous parallel programming, to deliver a fully transparent programming model. While the standard OpenCL requires special APIs for explicit (such as *clHostMemAllocINTEL*) or implicit memory management (such as *clSharedMemAllocINTEL*) before execution, Cohet allows applications to use mmap, malloc, and C++ new exactly as in CPU-only systems, without incurring page-fault penalties. Fig. 4 compares Cohet with CUDA's explicit and implicit copy programming models, using *AXPY* operation ($\mathbf{Y} = \alpha\mathbf{X} + \mathbf{Y}$) as a pseudocode

```cpp
int main() {
    // 1.Allocate host memory for X and Y
    float *h_X = malloc(N);
    float *h_Y = malloc(N);
    cpu_init_data(h_X, h_Y, N);
    // 2.Allocate device memory for X and Y
    float *d_X, *d_Y;
    cudaMalloc(&d_X, N);
    cudaMalloc(&d_Y, N);
    cudaMemcpy(d_X, h_X, N, H2D);
    cudaMemcpy(d_Y, h_Y, N, H2D);
    // 3.Launch AXPY kernel to GPU
    axpy_kernel<<<...>>>(N, a, d_X, d_Y);
    cudaDeviceSynchronize();
    cudaMemcpy(h_Y, d_Y, N, D2H);
    // 4.CPU consumes Y
    cpu_use_data(h_Y);
    free(h_X); free(h_Y);
    cudaFree(d_X); cudaFree(d_Y);
}
```

(a) Explicit copy

```cpp
int main() {
    // 1.Allocate unified memory
    // for X and Y
    float *X, *Y;
    cudaMallocManaged(&X, N);
    cudaMallocManaged(&Y, N);
    cpu_init_data(X, Y, N);



    // 2.Launch AXPY kernel to GPU
    // (H2D implicit copy)
    axpy_kernel<<<...>>>(N, a, X, Y);
    cudaDeviceSynchronize();
    // 3.CPU consumes Y
    // (D2H implicit copy)
    cpu_use_data(Y);

    cudaFree(X); cudaFree(Y);
}
```

(b) Implicit copy (UM)

```cpp
int main() {
    // 1.Allocate coherent memory
    // for X and Y
    float *X = malloc(N);
    float *Y = malloc(N);
    init_data(X, Y, N);



    // 2.Launch AXPY kernel to
    // a designated XPU
    clEnqueueNDRangeKernel(
        queue, axpy_kernel, ...);
    clFinish(queue);
    // 3.CPU consumes Y
    cpu_use_data(Y);

    free(X); free(Y);
}
```

(c) Cohet

Fig. 4: Comparison between CUDA and Cohet programming models using AXPY operation as an example.

example. Fig. 4(a) relies heavily on programmers for manual memory management, resulting in the highest code complexity (16 lines). Fig. 4(b) employs UM to reduce programming effort (10 lines), but it incurs non-negligible overhead from implicit data copies. In contrast, Cohet in Fig. 4(c) further simplifies heterogeneous programming to a level approaching homogeneous programming (9 lines), while achieving performance comparable to manually managed data.

IV. SimCXL Design and Implementation

A. Simulator Architecture

SimCXL is a cycle-level simulator intended for modeling the proposed Cohet computing framework with a CXL interconnect fabric connecting CPUs, XPUs, and a memory pool in a cache-coherent manner. Fig. 5 illustrates its modular architecture built on gem5's *full-system* (FS) mode [58], [59], modeling both the guest OS and hardware architecture to enable realistic simulation of software-hardware interactions.

We build three main components on gem5 for SimCXL. First, we add a PCIe device model called XPU which comprises processing units, device-attached memory, and a local cache. Second, we implement the complete CXL sub-protocols: CXL.io supports device enumeration and configuration, register access, and DMA by extending gem5's native PCIe protocol; CXL.cache and CXL.mem enable coherent load and store operations in device-to-host (D2H) and host-to-device (H2D) directions, respectively. These extensions unify host memory and device memory into a unified coherent memory pool through a custom MESI coherency protocol. Third, we develop a dedicated device driver in the guest OS to allow the host to discover and configure CXL devices; it also cooperates with the HMM module to present a unified mmap and malloc interface to user applications.

B. CXL Protocol Supports

Fig. 6 shows an x86 platform with a Ruby subsystem used in the FS mode. Ruby [60] supports the flexible modeling of various cache coherence protocols by specifying state transitions of multi-level caches using the domain-specific

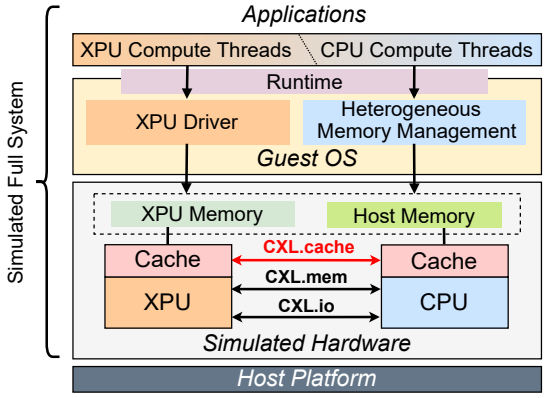


Fig. 5: Architecture of the proposed SimCXL simulator.

language SLICC. On the host side, the CPU cores and their MMUs connect to the Ruby sequencer through cache and Walker ports, respectively. The sequencer routes memory access requests by analyzing packet destination addresses: memory access requests are converted into *RubyRequest* packets and forwarded to the cache hierarchy (red arrows), while IO accesses are directed to target devices via the IO bus (orange arrows). The memory interface coordinates requests from LLC and DMA controllers while converting *RubyRequest* packets into gem5 memory packets for memory controllers. The device side includes the CXL accelerator with its HMC and device memory. The accelerator utilizes three CXL sub-protocols. CXL.io supports MMIO and DMA through PIO and DMA ports, respectively. CXL.cache enables coherent memory access in cacheline granularity via the cache port. CXL.mem incorporates device memory into the host memory address space, managed uniformly by the memory interface, allowing cores to perform standard load/store operations with full cache coherence.

1) CXL.io support: The CXL.io sub-protocol handles device enumeration and configuration during system initialization. The BIOS performs CXL.io configuration reads to determine the size of each BAR register space, maps the corresponding physical address range, and writes the base

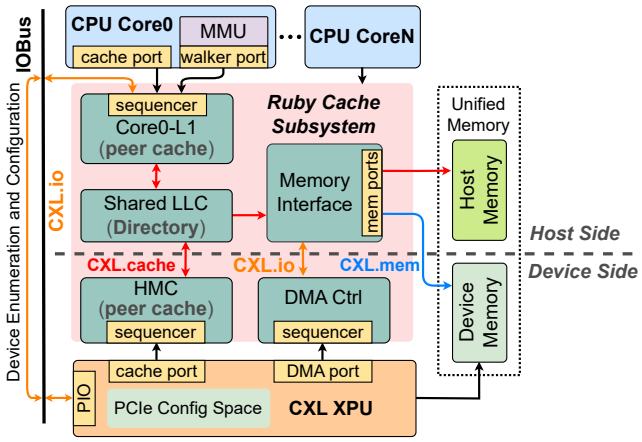


Fig. 6: CXL protocol implementation in SimCXL.

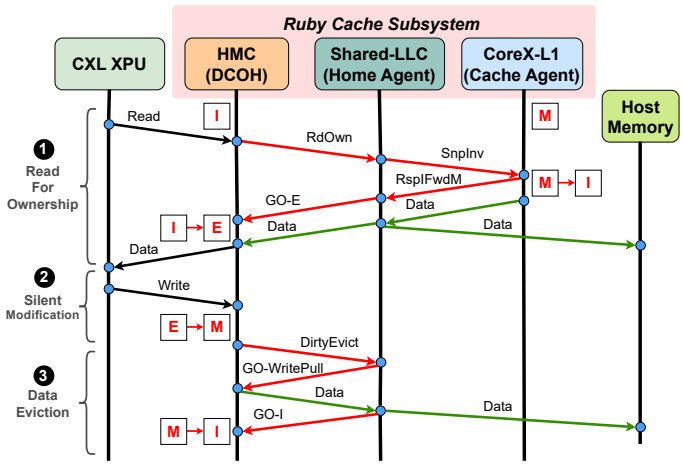


Fig. 7: An example of CXL.cache request-response flow when an XPU performs a store operation on a host memory address.

addresses back via configuration writes. A kernel driver then creates */dev/cxl_acc* and exposes open, mmap and release syscalls, allowing the CPU to read and write the BAR space of the CXL device via MMIO to control the device. The device also performs DMA operations through its DMA port.

2) CXL.cache support: A cache port is implemented and connects to the HMC within Ruby for the CXL accelerator. Using SLICC, we developed a directory-based two-level MESI protocol optimized for heterogeneous systems. Core0-L1 and HMC serve as peer caches which are privately owned by Core0 and the accelerator, respectively, and they both share the LLC. The metadata of each LLC cacheline embeds directory information for coherence management, including a *CacheState* field for transient or stable states, an *ID* field (the unique identifier of each cache controller in Ruby) tracking the exclusive holder, and a *bit vector* recording all sharers.

Fig. 7 illustrates an example of CXL.cache sub-protocol workflows when an accelerator exclusively modifies and writes back a dirty cacheline, in three phases. ❶ Read For Ownership: The accelerator requests exclusive access to a target cacheline, which then issues a *RdOwn* request to the LLC upon HMC miss. LLC sends *SnpInv* to invalidate (*I*) CoreX-L1's modified (*M*) copy, writes back dirty data to memory, and forwards data with exclusive (*E*) state to HMC. ❷ Silent Modification: Accelerator modifies cacheline locally, upgrading state to *M* without coherence messages. ❸ Data Eviction: HMC issues *DirtyEvict*. LLC verifies state and authorizes write-back with *GO-WritePull*, then sends *GO-I* to invalidate HMC's cacheline.

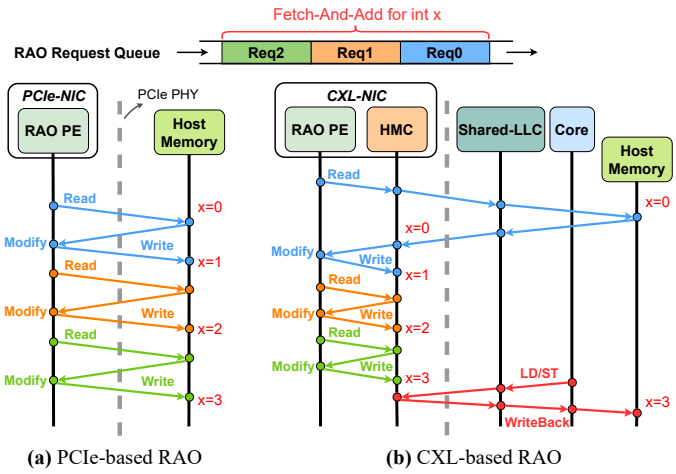
3) CXL.mem support: We developed a dedicated memory interface module for organizing the unified memory in our Ruby implementation. This module routes memory access requests from the shared LLC to either the host memory or the device memory based on address ranges configured by the BIOS and subsequently returns response data to the requester. The device memory can directly leverage various existing memory models in gem5, including DDR3/4/5, non-volatile memory (NVM), and high bandwidth memory (HBM). Furthermore, we modified the memory management module in the OS kernel to recognize CXL device memory as a CPU-less NUMA node during system initialization. This modification enables transparent utilization of CXL-expanded memory by upper-layer applications.

V. CoHet Killer Apps

A. Remote Atomic Operations

Atomic operations play a pivotal role in parallel programming, as they are widely used to ensure consistency of shared data accesses [61] and implement lock-free algorithms [62] as well as synchronization primitives such as spinlocks [46], [63], [64]. As a result, modern processors have been highly optimized for efficient atomic operations within a single multi-core node. However, the emergence of data-intensive workloads (e.g., machine learning and graph analytics) has driven the distribution of massive shared data sets across multiple computing nodes in order to exploit data parallel processing. This shift gives rise to the demand for frequent cross-node *remote atomic operations* (RAOs) [41]–[47]. RAOs enforce atomic updates to shared data residing in a remote memory location, which is an extension of local atomics to distributed computing systems by means of remote direct memory access (RDMA). The study in [42] shows that when the number of computing nodes exceeds 64 and global memory accesses follow a random indexing pattern, RAOs can constitute as much as 98.44% of all atomic operations. Hence, the design and implementation efficiency of RAOs are crucial to the performance of large-scale high-performance computing systems.

1) Conventional RAO offloading design in PCIe-NIC: Traditional software-based implementations of RAOs rely on parallel programming libraries such as MPI and OpenSHMEM, but these implementations often suffer from performance overhead due to redundant software routines (e.g., multi-layer communication protocol stacks, software locks/semaphores, and redundant memory copies) [41], [42]. To address this problem, modern supercomputers typically employ hardware offloading techniques in RDMA networks [65] to optimize common RAO primitives such as *compare-and-swap* (CAS) and *fetch-and-add* (FAA). In such systems, RAOs are of-



(a) PCIe-based RAO **(b)** CXL-based RAO

Fig. 8: PCIe-based RAO vs. CXL-based RAO in NIC design.

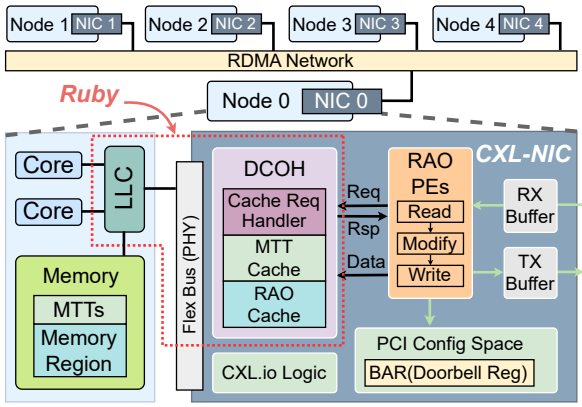


Fig. 9: RAO offloading design in a CXL-NIC of SimCXL.

floaded to the RDMA network interface controllers (NICs) and executed as indivisible *read-modify-write* operations across the PCIe interface, significantly reducing inter-node traffics. However, PCIe-NICs still face critical limitations when accelerating RAOs. That is, PCIe does not intrinsically guarantee memory consistency and atomicity, as well as timely visibility of data modifications to the host. Each RAO requires two DMA transfers, one for the read and another for the write, which must be issued consecutively without interruption for the same target address. This can be explained with an example illustrated in Fig. 8(a). When three FAA requests on a shared int variable x are queued for processing, the NIC must launch six costly DMA transactions across PCIe. Moreover, due to PCIe's relaxed ordering and split-transaction mechanisms [12], a later read request may arrive before a prior write, leading to potential read-after-write (RAW) hazards. To avoid such hazards, each RAO must wait for an acknowledgment of the previous write before proceeding, which significantly limits the RAO throughput [66].

2) Cohet-driven RAO offloading design in CXL-NIC: In contrast, Fig. 8(b) shows the CXL-based RAO design guided by Cohet to address this bottleneck. The CXL-NIC and the host CPU share a unified memory view. This allows the CXL-NIC to cache the target variable in the HMC, enabling subsequent RAOs to be serviced directly within the HMC without redundant PCIe transfers. The processing element (PE) locks the target RAO cacheline to prevent any invalidation during the RMW operation and thus preserve atomicity. Meanwhile, hardware-maintained cache coherence ensures that the host always perceives and retrieves the most recent data. This mechanism substantially improves the efficiency of RAO-intensive operations such as sequencers [43], lock services [46], [64], and synchronization barriers [63], which often involve many-to-one contention on a small set of hot memory locations. Furthermore, most data-intensive applications use pointer-based data structures such as graphs, unbalanced trees, unstructured grids, and sparse matrices. These structures generate fine-grained and irregular memory access patterns [67]. As discussed in Section II-A, traditional DMA mechanisms are unsuitable for such access patterns. In contrast, CXL.cache efficiently supports such workloads through low-latency, coherent access to shared memory.

As shown in Fig. 9, we implemented a CXL-NIC with RAO offloading design in SimCXL. To boost concurrency, multiple RAO PEs are added to handle incoming RAO requests from remote servers in parallel. The DCOH module is responsible for caching data and maintaining coherence with the host's cache hierarchy. When an RAO request arrives at the RX buffer, the RAO PEs parse the request and perform the read-modify-write operation in three stages. (1) In the "read" stage, a PE first queries the DCOH to check whether the target variable resides in the cache. In case of a cache hit, the operation proceeds directly to the "modify" stage. Otherwise, the DCOH requests the latest data from the host's LLC. (2) In the "modify" stage, the PE executes the atomic operation according to the request type, such as FAA or CAS. (3) In the "write" stage, the result is written back to the cache without immediate eviction to the host. When the host later accesses the same variable, the DCOH ensures that any dirty cachelines are written back to host memory (as shown in Fig. 8). After the RAO operation completes, the CXL-NIC sends the response back to the remote server.

B. Remote Procedure Calls

Remote procedure calls (RPCs) serve as the fundamental inter-process communication layer for modern cloud services, supporting diverse distributed applications (e.g., microservices [48], cloud storage [68], and machine learning [69]). An RPC abstracts remote compute resources by making a function call to a remote machine similar to a local call. Its underlying stack (typically implemented as a user-space library [70], [71]) transparently handles connection management, network protocols, parameter (de)serialization, and thread scheduling. Industry measurements reveal significant RPC overheads: Google reports that RPCs account for 7.1% of CPU cycles in production clusters [51]; Meta observes that parameter (de)serialization alone consumes 6.7% of CPU cycles in seven critical microservices [53]. These findings highlight the importance of accelerating the RPC stack especially (de)serialization to reclaim CPU resources for application workloads.

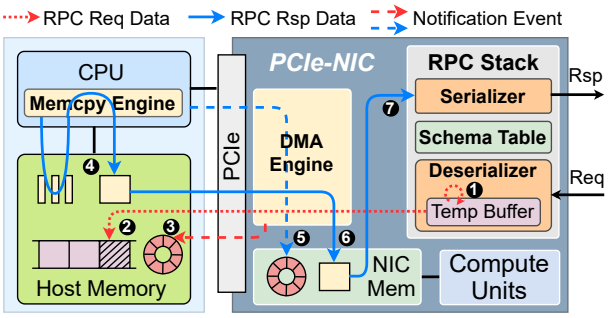


Fig. 10: Implementation of an RPC offloading design [49] in a PCIe-NIC of SimCXL.

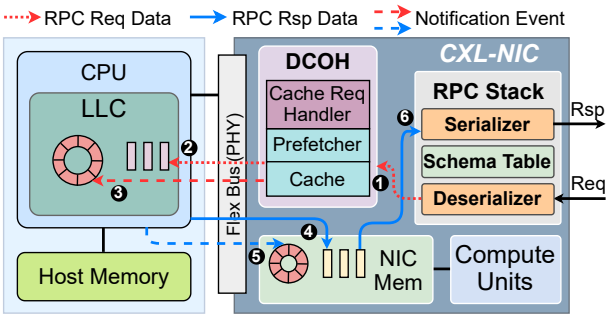


Fig. 11: RPC offloading design in a CXL-NIC of SimCXL.

1) Conventional RPC offloading design in PCIe-NIC: Among existing serialization libraries [72]–[74], we focus on Google's Protobuf [72] in this paper, as it has widespread adoption in today's cloud applications. Two key characteristics of RPC messages critically impact acceleration designs. (1) The vast majority of messages are extremely small; a Google cluster analysis shows that 56% of messages are $\leq$32 bytes and 93% are $\leq$512 bytes [52]. (2) Nested messages can be defined via pointer references, analogous to pointer chasing, incurring significant cumulative overhead during (de)serialization [48]; in real-world applications, the nesting of RPC message may exceed ten levels [51], [52].

A recent study RpcNIC [49] proposed a hardware offloading design of the RPC stack to a PCIe-NIC, which we have implemented in SimCXL as shown in Fig. 10. The host pre-runs the Protobuf compiler to store message structure metadata in a schema table, which guides message fields to decode in in-memory C++ objects or encode them into binary sequences. ❶ When an RPC request arrives, the NIC deserializer decodes it field-by-field, accumulating results in a 4 KB on-chip temp buffer. ❷ When the current request's deserialization is finished or the buffer is full, a one-shot DMA transfer to host memory is triggered. ❸ The NIC then increments the head pointer of a ring buffer in main memory through a DMA write to signal the arrival of a new request; the CPU processes the request and increments the tail pointer to indicate completion. ❹ For RPC responses, RpcNIC employs a pre-serialization mechanism; the CPU first invokes the on-chip memory copy engine (i.e., Intel's DSA) to iteratively copy noncontiguous fields into a pre-allocated DMA-safe buffer. ❺ The CPU then updates an NIC-resident ring buffer via MMIO to signal the completion of pre-serialization. ❻ The NIC copies the prepared data from the host memory via a DMA read. ❼ The hardware serializer iteratively encodes the data in the NIC memory and transmits the RPC response over the network.

Despite the one-shot DMA and pre-serialization mechanism in the RpcNIC design mitigate the high latency and low throughput of fine-grained or nested RPC message transfers over PCIe, three inherent limitations remain: (1) high design complexity in multi-device interaction, (2) redundant data copies, and (3) non-trivial CPU control overhead that may negate acceleration benefits.

2) Cohet-driven RPC offloading design in CXL-NIC: Fig. 11 illustrates our proposed CXL-based NIC offloading design, where the NIC operates as a type-2 device that enables both the CPU and the NIC to access each other's memory through load/store instructions. The RPC processing pipeline comprises six phases as follows. ❶ When an RPC request arrives, the CXL-NIC deserializer performs field-by-field decoding. ❷ Each decoded field when ready is pushed by DCOH to a designated location in the CPU's LLC via NC-P (since the CPU will soon access the decoded fields to service the RPC request). ❸ A ring buffer is maintained in host memory and, more importantly, cached in the LLC for task queuing. ❹ For RPC responses, the CPU constructs and populates message objects to device memory using CXL.mem. ❺ Once the objects are ready, the CPU notifies the NIC of pending serialization tasks. ❻ The serializer performs iterative encoding from the NIC memory and transmits the RPC response over the network.

Although CXL.mem-based message construction delivers high performance, requiring application code modifications poses backward compatibility challenges for legacy systems. We also propose an alternative CXL.cache approach: The CPU constructs response messages in the host memory as done in conventional implementations, while the NIC uses CXL.cache to directly access these structures during serialization. To further reduce access latency, we enhance the DCOH module with a device-side hardware prefetcher, thanks to the coherenet memory-semantic interconnect. The RPC prefetcher is a multi-stride prefetcher, which records cache-miss addresses to identify data streams with various stride patterns and issues prefetches accordingly, achieving a balance between performance and design complexity.

VI. Experiments and Evaluation

A. Experimental Setup

1) Hardware testbed: Our hardware testbed is a high-performance dual-socket server, detailed in Table I. Each socket houses an Intel Gen4 Xeon Platinum 8468V processor, which integrates four CPU chiplets with each paired with a memory controller driving two memory channels. Each channel is populated with two 32 GB DDR5 4800 MT/s DIMMs. The four chiplets can be configured as a monolithic CPU or two/four NUMA nodes in sub-NUMA cluster mode

TABLE I: Configurations for hardware testbed and SimCXL.

Config. Parameter	CXL Testbed	SimCXL
Linux kernel version	v6.5.0	Modified v6.12
CPU type	Xeon® Platinum 8468V	X86O3CPU
CPU cores	48	48
Local DRAM type	DDR5 4800	DDR5 4400
#Memory channels/NUMA	2	2
DDR DRAM size	1TB	32GB
LLC size	97.5MB	96MB
CXL&PCIe accelerators	Intel Agilex I-Series FPGA	CXL-&PCIe-NIC models
HMC size	128KB, 4 ways	128KB, 4 ways
CXL memory expander	Samsung memory expander	Memory expander model

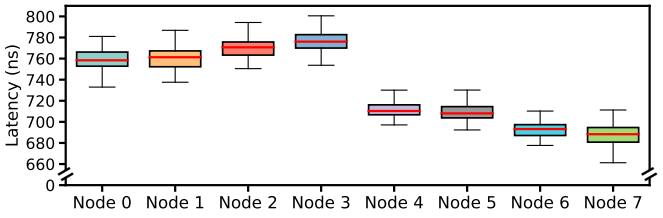


Fig. 12: Latency distribution of cacheline loads by the LSU via CXL.cache from memories on different NUMA nodes.

(SNC) [75]. To ensure accurate and stable D2H memory access measurements, we enabled SNC-4 in the BIOS, dividing each socket into four separate NUMA nodes. Memory accesses from the same physical core to different NUMA nodes incur varying latencies depending on the number of NoC and UPI routing hops, thus manifesting the NUMA effect. The CXL devices on the testbed include an Intel Agilex I-Series FPGA Development Kit [34] and a Samsung 512 GB memory expander [29]. The FPGA operates at 400 MHz and supports both CXL type-1/2 device configurations (denoted as CXL-FPGA) and standard PCIe configurations (denoted as PCIe-FPGA), connected to Socket1 via PCIe5.0 $\times$16. The CXL-FPGA integrates a hard CXL IP [35] and is equipped with a 128 KB 4-way HMC, whereas the PCIe-FPGA integrates a multi-channel DMA IP [76]. The Samsung expander, connected to Socket1 via PCIe5.0 $\times$8, is exposed to the system as a CPU-less NUMA node.

2) SimCXL simulator: SimCXL is developed based on gem5 v23.1, with X86O3CPU and 32 GB of 4400 MT/s DDR5 dual-channel memory. In terms of CXL devices, the device driver is developed on top of the Linux v6.12 kernel. The modular CXL controller in SimCXL can model CXL type-1/2/3 devices. Device memory can directly leverage gem5's native DDR, NVM, and HBM models; device compute units are customized following the gem5 programming framework. For type-1/2 devices (denoted as CXL-FPGA$_{\text{sim}}$), a 4-way 128 KB HMC is implemented in the Ruby subsystem. Based on the latency breakdowns presented in the CXL specification [18] and related studies [9], [12], [23], [24], the CXL controller and Ruby's heterogeneous MESI protocol set multiple configurable parameters for tuning H2D and D2H access latency and bandwidth. For PCIe devices (denoted as PCIe-FPGA$_{\text{sim}}$), we implemented DMA read and write engines to handle H2D and D2H data transfers, respectively. Both the CXL and PCIe device models can be configured to 400 MHz to match the FPGA configurations. Meanwhile, the frequency can also be set to 1.5 GHz to model real-world ASIC devices (denoted as CXL-ASIC$_{\text{sim}}$ and PCIe-ASIC$_{\text{sim}}$, respectively), by frequency scaling based on the clock cycles measured from CXL-FPGA tests; this allows us to evaluate the realistic offloading performance of RAO and RPC on future production-grade CXL devices.

3) Hardware calibration microbenchmarks: To characterize the performance of CXL.cache, we implemented a load/store unit (LSU) on the CXL-FPGA and in SimCXL to generate host memory requests with configurable access patterns. On the FPGA, a purpose-designed performance monitoring unit collects latency and bandwidth statistics by recording request and response timestamps. In SimCXL, the device model includes detailed statistics tracking for the same measurements. For DMA performance, we utilized the DMA IP on the PCIe-FPGA and the DMA read/write engine in SimCXL to generate workloads with varying message granularities and measured latency and bandwidth.

4) Hardware calibration methodology: In this work, we calibrate the latency and bandwidth of CXL.cache for D2H memory accesses. The calibration of H2D memory access via CXL.mem will be presented in another paper. By tuning the model parameters of SimCXL, we are able to calibrate it to make its performance match that of CXL-FPGA. Note that the DMA performance calibration for SimCXL and PCIe-FPGA follows a similar approach. For D2H accesses via CXL.cache, cachelines may hit in the device's HMC, host LLC, or host memory. HMC hits are tested by repeating address sequences. For LLC hits and memory hits, after initializing the target host memory addresses, we use the *CLDEMOTE* [77] instruction to push cachelines to the LLC or the *CLFLUSH* [78] instruction to flush cachelines to memory, followed by the LSU test sequence. For predictable performance, hyper-threading and hardware prefetchers are disabled, and CPU frequency is fixed at 2.4GHz during test.

B. SimCXL Latency Calibration

1) NUMA effects for CXL.cache: We first evaluated the impact of NUMA effects on CXL.cache accesses by conducting identical tests across nodes 0-7 in sequence: we allocated and initialized huge pages on a target node, followed by LSU issuing 32 cacheline granularity (64 B) load requests to that node's memory region. Results from 1000 repeated trials are shown in Fig. 12. The CXL-FPGA accesses to the nearest node (node 7) exhibit the lowest median latency (688 ns). Within the same socket, the median latency increases with node distance: node 6 (693 ns), node 5 (708 ns), and node 4 (710 ns). In contrast, accesses to remote socket nodes incur substantially higher median latencies: node 0 (758 ns), node 1 (761 ns), node 2 (770 ns), and node 3 (776 ns). The median latency difference between the farthest node 3 and the nearest node 7 is 88 ns, with a maximum latency gap approaching 150 ns. These results demonstrate a significant impact of NUMA effects on CXL.cache access latency, which suggests that the default memory allocation strategy (SNC-disabled) may scatter pages randomly across system memory, leading to unpredictable and

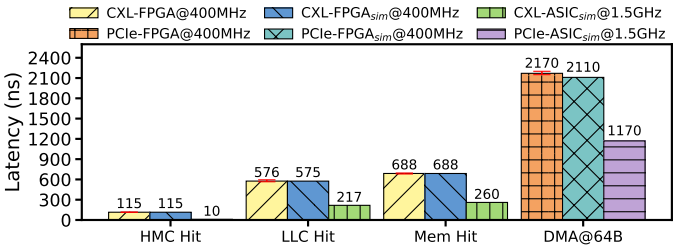


Fig. 13: Median load latency for cacheline HMC hits, LLC hits, and memory hits via CXL.cache, in comparison to DMA read latency at a 64 B message granularity. Error bars represent the measured 25th and 75th percentiles.

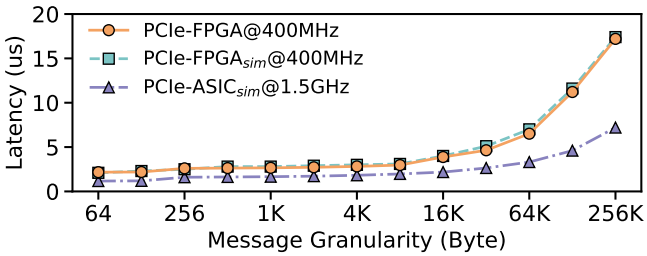


Fig. 14: Median H2D latency for DMA read with varying message granularity.

often suboptimal access performance. Despite these striking latency disparities, the bandwidth measurements across nodes 0-7 show marginal variations.

2) CXL.cache latency: To measure the latency of CXL.cache, we configured the LSU to issue 32 64 B load requests to sequential addresses, repeating each test 1000 times and reporting the median latency (only load results are shown, since PCIe PHY read/write performance is symmetric). As shown in Fig. 13, on the CXL-FPGA at 400 MHz, HMC hit exhibits the lowest latency (115 ns), followed by LLC hit (575.6 ns), and off-chip memory hit (688.3 ns). The LLC hit latency is roughly 5× higher than the HMC hit, mainly due to the extra PCIe traversal after an HMC miss. Memory hit latency is only 1.19× higher than that of the LLC hit, reflecting a ∼100 ns DRAM access overhead. SimCXL results closely match the CXL-FPGA measurements, and their relatively high latencies come from the lower operating frequency at 400 MHz. When we scale the frequency with the same clock cycle counts to 1.5 GHz in SimCXL, CXL.cache shows the potential for much lower latencies. These results characterize the performance of CXL.cache across different memory access tiers, from HMC, through LLC over PCIe, to the most distant memory, and validate the fidelity and parameterized flexibility of SimCXL in modeling real-world CXL devices.

3) DMA latency: Fig. 14 shows the latency curves for H2D DMA read transfers at different message granularities, for the purpose of comparing with the CXL.cache load. On a PCIe-FPGA operating at 400 MHz, the latency remains roughly constant at ∼2.5 us for message sizes below 8 KB, since the intrinsic DMA setup overhead is dominant at small size. As the message size grows beyond 8 KB, the data transfer time gradually dominates the overall latency. In contrast,

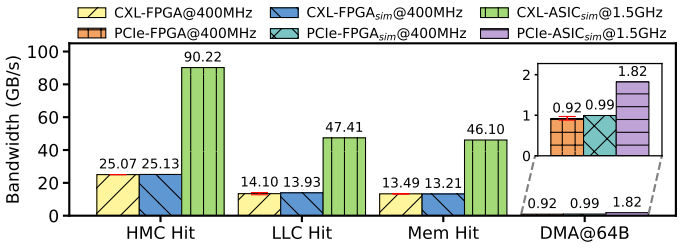


Fig. 15: Average load bandwidth for cacheline HMC hit, LLC hit, and memory hit via CXL.cache, in comparison to DMA read bandwidth at a 64 B message granularity. Error bars represent $\pm 3\sigma$ standard deviation.

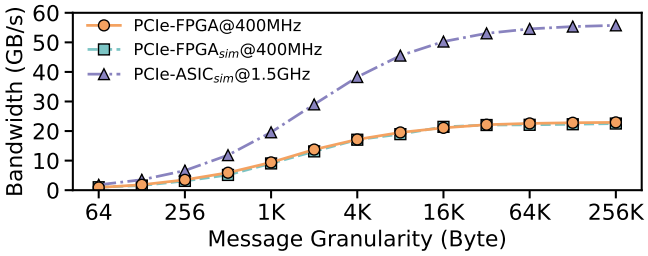


Fig. 16: Average H2D bandwidth for DMA read with varying message granularity.

CXL.cache incurs no setup overhead and performs memory accesses through load/store operations at cacheline granularity, achieving a 68% lower latency for memory hit than DMA at 64B (see Fig. 13). This result highlights the clear performance advantage of CXL.cache for fine-grained and latency-sensitive data accesses. Moreover, the DMA engine implemented in SimCXL at 400 MHz closely matches the measured DMA latency curve on real platform.

C. SimCXL Bandwidth Calibration

1) CXL.cache bandwidth: To measure the bandwidth of CXL.cache, we configured the LSU to issue a large number of memory requests until a stable bandwidth value was achieved. We observed that when the request count exceeds 512 (totaling 32 KB), the bandwidths for HMC hit, LLC hit, and memory hit all converge. We then report the bandwidth results in Fig. 15 at 2,048 requests (totaling 128 KB), which are averaged over 1,000 tests. On the 400 MHz CXL-FPGA, the peak bandwidths for HMC hit, LLC hit, and memory hit stabilize at 25.07 GB/s, 14.10 GB/s, and 13.49 GB/s, respectively. Given the FPGA's theoretical maximum bandwidth of 25.6 GB/s at 400 MHz, LLC and memory hits reach only 55% and 52.7% of the peak bandwidth. This degradation probably stems from pipeline bubbles introduced by cache-coherence checks during requests routed to the host, a behavior also observed in CPU accesses to remote NUMA nodes [23], [24]. In contrast, HMC accesses, eliminating the need for host-side coherence checks, achieve 97.7% of the theoretical bandwidth. One can see that SimCXL accurately reproduces the bandwidth of CXL-FPGA at 400 MHz.

2) DMA bandwidth: Fig. 16 illustrates the average H2D bandwidth achieved by DMA at different message gran-

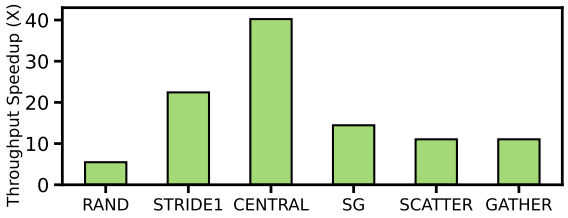


Fig. 17: Throughput speedup of CXL-based RAO versus PCIe-based RAO across six CircusTent workloads.

ularities. On a PCIe-FPGA operating at $400\,\mathrm{MHz}$, DMA throughput is very limited for fine-grained transfers. For example, DMA delivers just $0.92\,\mathrm{GB/s}$ (0.36% of the theoretical peak) at message size of $64\,\mathrm{B}$, whereas CXL.cache achieves $13.25\,\mathrm{GB/s}$, $14.4\times$ that of DMA (see Fig. 15). However, as the message size increases, DMA can pipeline transfers more efficiently, reaching $22.9\,\mathrm{GB/s}$ at $256\,\mathrm{KB}$ (86.7% of the theoretical peak), while CXL.cache achieves only 57.7% of the DMA bandwidth. These results show that CXL.cache provides a clear throughput advantage for small-message exchanges between host and accelerator, whereas DMA remains the preferred mechanism for bulk transfers. Again, the DMA engine in SimCXL reproduces the FPGA bandwidth curve accurately. Note that after all calibration tests, our simulator achieves a mean absolute percentage error of 3%.

D. Evaluation of Remote Atomic Operations

We evaluated six RAO access patterns from the CircusTent benchmark [41] on both PCIe-NIC and CXL-NIC in SimCXL, with throughput speedups shown in Fig. 17. The CENTRAL pattern, a many-to-one workload that models a distributed lock service, achieves the highest speedup ($40.2\times$) by caching the hotspot data in CXL-NIC's HMC, avoiding costly PCIe DMA transfers. STRIDE1 follows with a $22.4\times$ speedup, as its sequential atomic updates on $8\,\mathrm{B}$ elements leverage $64\,\mathrm{B}$ cacheline granularity, enabling cache hits for seven subsequent operations after fetching a line. SCATTER and GATHER patterns, which involve randomized updates in the global address space, exhibit moderate speedups due to lower cache hit rates. The SG workload, which combines SCATTER and GATHER, follows a similar trend. Even the fully random pattern RAND, with a near-zero cache hit rate, delivers a $5.5\times$ throughput gain due to the lower access latency of CXL.cache compared to PCIe-based DMA for fine-grained accesses. These results demonstrate that CXL-NIC significantly enhances the distributed RAO throughput across various access patterns by leveraging low-latency memory access and on-device caching of hot data.

E. Evaluation of Remote Procedure Calls

We evaluated de/serialization time for CXL-based RPC (CXL-NIC.*) versus PCIe-based RPC (RpcNIC [49]) using HyperProtoBench [52]. Fig. 18(a) shows deserialization results, with CXL-NIC achieving speedups of $1.33\times$ (Bench5) to $2.05\times$ (Bench1) in six benchmarks. This acceleration comes

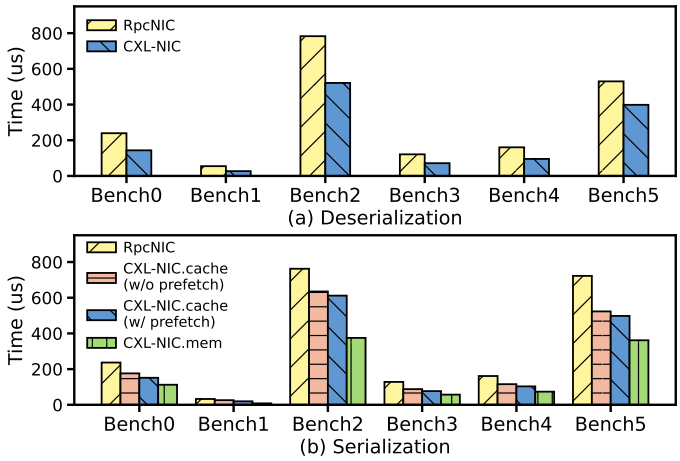


Fig. 18: De/serialization time of CXL-based RPC versus PCIe-based RPC across six benches from HyperProtoBench.

primarily from the CXL-NIC's low-latency fine-grained writes. Although Bench5's large string fields favor DMA, CXL.cache still outperforms RpcNIC by eliminating redundant copies and software overhead. Bench1, with small-field messages, achieves the highest speedup. Moreover, CXL-NIC uses NC-P instructions to push fields directly into the LLC, improving subsequent CPU access performance.

Fig. 18(b) presents serialization results, where all three CXL-NIC solutions outperform RpcNIC. Using CXL.mem, the CPU constructs messages directly in device memory, achieving speedups of $2.0\times$ (Bench5) to $4.06\times$ (Bench1), as devices access local memory for deserialization, eliminating costly PCIe transfer overhead. Our evaluation results using a Samsung memory expander reveal a 8% higher overhead at most for message construction through CXL.mem versus construction in host memory, which is an acceptable cost given CXL-NIC's performance gain. With CXL.cache, the CXL-NIC retrieves individual fields directly from host memory, achieving speedups of $1.34\times$ (Bench2) to $1.65\times$ (Bench1) when the RPC prefetcher is enabled. Compared to the CXL-NIC design without prefetch, our RPC prefetcher improves the serialization performance by 12% on average across the six benches. Note that the minimum gain is 3.6% for Bench2, since it contains a large number of deeply nested messages limiting prefetch effectiveness. Nonetheless, the CXL-NIC without prefetch still benefits from the low latency of CXL.cache in comparison to RpcNIC. In summary, CXL-based RPCs leverage fine-grained low-latency access and unified memory view to significantly streamline operations, accelerating (de)serialization by an average of $1.86\times$ when compared to the PCIe-based solution.

VII. Related Work

CXL simulation and emulation methods. Previous studies often used remote NUMA node (homogeneous in essence) for the emulation of CXL type-1/2 devices [11], [39], [40] or type-3 devices [26], [83]–[85]. Although this approach has greatly facilitated early investigations, the performance

TABLE II: Comparison between SimCXL and prior CXL system simulators/emulators.

Simulator/Emulator	Cohet Support	CXL.cache Support	CXL.mem&io Support	CXL XPU Models	Full System	Hardware Calibration	Configurability	Sim. Error	Sim. Speed
CXLMemSim [79]	No	No	No	No	No	No	Medium	High	Medium
CXL-DMSim [80]	No	No	Yes	No	Yes	Yes	High	Low	Low
Mess+gem5 [81]	No	No	No	No	No	No	High	Medium	Low
QEMU [82]	No	No	Yes	No	Yes	No	High	High	High
Remote NUMA [40], [83]	No	No	No	No	No	N/A	Low	High	High
SimCXL	Yes	Yes	Yes	Yes	Yes	Yes	High	Low	Low

characteristics of a remote NUMA node differ substantially from those of a genuine CXL device [9], [86], which can in some cases lead to misleading conclusions [23]. Existing software simulators focus solely on modeling memory expansion, pooling, and sharing for type-3 devices [79]–[82], without any touch on heterogeneous systems with typ-1/2 devices. Table II compares SimCXL with other existing CXL system simulation and emulation tools across several dimensions. Only SimCXL models CXL XPU devices and all three CXL sub-protocols, and supports full-system simulation of Cohet computing framework. After meticulous hardware calibration, SimCXL achieves a low simulation error of 3%.

CXL memory disaggregation and pooling. A wealth of prior work has evaluated real CXL type-3 devices, providing comprehensive performance characterizations and practical insights [23]–[25], [86]. Many studies have also explored CXL-based tiered memory management at the OS level. [26], [87]–[90]. In addition, some studies exploit CXL memory to expand system memory capacity and bandwidth to accelerate memory-intensive applications [28], [85], [91]–[93].

CXL-based heterogeneous computing. Recently, a study provides a detailed performance characterization of the CXL type-2 device (FPGA) and offloads two OS kernel memory optimization functions [9]. Some studies leverage the cache coherence and unified memory view provided by CXL.cache to optimize specific application scenarios [39], [40], [94]. Other works explore host-accelerator synchronization mechanisms based on cache coherence [10], [11], [95], [96]. However, these works are limited to: 1) hardware customization exploiting selected CXL.cache features with direct manipulations on physical memory addresses (FPGA-like development), 2) predominately relying on remote NUMA emulation for evaluation. In contrast, our work builds a Cohet computing framework with a global perspective on the software stack and hardware architecture, greatly simplifying programming model and memory management. We also introduce SimCXL, a research tool that fills a critical gap in CXL-enable heterogeneous systems simulation infrastructure.

VIII. Discussion and Future Work

Despite its promise, CXL-driven coherent heterogeneous computing still faces many challenges. The overhead of ATS-based address translation remains unexplored due to the lack of ATS support on current CXL FPGA platforms [97]; prior CCIX studies indicate substantial ATC miss penalties [98]. Furthermore, OS support for CXL devices remains immature [36], and mainstream programming frameworks like OpenCL have not yet integrated CXL workflows.

Our future work will focus on adding new features to SimCXL, currently limited to CXL 1.1. With an industry target toward CXL 3.x and its fabric-based multi-node interconnects [18], we are designing CXL switch models to extend SimCXL and exploring integration with SimBricks [99] for large-scale fabric-based system simulation. As the coherence domain scales up (i.e., with more child nodes in a supernode), the performance loss and design complexity of maintaining hardware coherence increase sharply, and the huge coherence traffic can become a system bottleneck. To mitigate coherence-traffic storms, we plan to explore a hierarchical coherence protocol for small-scale supernodes. Each child node interacts with a local agent for coherence transactions; the local agent consults a global agent only if it lacks the requested replica.

Furthermore, we will continue to explore the broader applications of Cohet. In graph processing, large datasets can be stored in a unified memory pool, and graph algorithms with fine-grained random-access patterns [100], [101] offloaded to CXL accelerators can benefit from the coherent CXL interconnect. Similarly, in-memory key-value store operations (e.g., GET/PUT) offloaded to CXL accelerators will benefit from lower-latency, fine-grained memory accesses. Moreover, the superior attributes of CXL such as memory pooling and sharing, p2p direct memory access, and TEE security offer compelling opportunities to accelerate workloads such as large language model inference and distributed services.

IX. Conclusion

In this work, we introduce Cohet, a CXL-driven coherent heterogeneous computing framework that overcomes the inefficiency of conventional PCIe-based heterogeneous systems in fine-grained host-accelerator interactions and simplifies the heterogeneous programming model. We also present SimCXL, a full-system cycle-level simulator that provides a flexible research platform for Cohet computing systems, and it will be open-sourced soon. We have rigorously calibrated SimCXL against a real CXL-supported testbed to ensure high-fidelity simulations. Finally, we have demonstrated two Cohet-driven killer apps namely RAO and RPC on SimCXL, demonstrating significant performance gains and streamlined design flows compared to PCIe-based solutions.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful and constructive feedback. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4500304, and in part by the National Natural Science Foundation of China under Grants 62332021, 62304257 and 62472058.

REFERENCES

[1] Y. Yuan, R. Wang, N. Ranganathan, N. Rao, S. Kumar, P. Lantz, V. Sanjeepan, J. Cabrera, A. Kwatra, R. Sankaran, I. Jeong, and N. S. Kim, "Intel accelerators ecosystem: An soc-oriented perspective : Industry product," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 848–862. [Online]. Available: https://ieeexplore.ieee.org/document/10609705

[2] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. A. Patterson, "TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, 2023, pp. 1–14. [Online]. Available: https://doi.org/10.1145/3579371.3589350

[3] I. Burstein, "Nvidia data center processing unit (DPU) architecture," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–20. [Online]. Available: https://ieeexplore.ieee.org/document/9567066

[4] B. Burres, D. Daly, M. Debbage, E. Louzoun, C. Severns-Williams, N. Sundar, N. Turbovich, B. Wolford, and Y. Li, "Intel's hyperscale-ready infrastructure processing unit (IPU)," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–16. [Online]. Available: https://ieeexplore.ieee.org/document/9567455

[5] Y. Go, M. Jamshed, Y. Moon, C. Hwang, and K. Park, "Apunet: revitalizing GPU as packet processing accelerator," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 83–96. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/go

[6] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale DNN processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 1–14. [Online]. Available: https://ieeexplore.ieee.org/document/8416814

[7] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: scalable deep learning on distributed GPUs with a GPU-specialized parameter server," 2016, pp. 1–16. [Online]. Available: https://doi.org/10.1145/2901318.2901323

[8] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud," in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2018, p. 51–64. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone

[9] H. Ji, S. Vanavasam, Y. Zhou, Q. Xia, J. Huang, Y. Yuan, R. Wang, P. Gupta, B. Chitlur, I. Jeong, and N. S. Kim, "Demystifying a CXL type-2 device: A heterogeneous cooperative computing perspective," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 1504–1517. [Online]. Available: https://ieeexplore.ieee.org/document/10764537

[10] V. Suresh, B. Mishra, Y. Jing, Z. Zhu, N. Jin, C. Block, P. Mantovani, D. Giri, J. Zuckerman, L. P. Carloni, and S. V. Adve, "Mozart: Taming taxes and composing accelerators with shared-memory," in *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2024, p. 183–200. [Online]. Available: https://doi.org/10.1145/3656019.3676896

[11] H. N. Schuh, A. Krishnamurthy, D. Culler, H. M. Levy, L. Rizzo, S. Khan, and B. E. Stephens, "CC-NIC: a Cache-Coherent Interface to the NIC," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 1*, Apr. 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3617232.3624868

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

[13] NVIDIA, "Unified Memory for CUDA Beginners," Accessed: 2025-03-30. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

[14] G. Schieffer, J. Wahlgren, J. Ren, J. Faj, and I. Peng, "Harnessing integrated CPU-GPU system memory for HPC: a first look into Grace Hopper," in *Proceedings of the 53rd International Conference on Parallel Processing (ICPP)*, 2024, p. 199–209. [Online]. Available: https://doi.org/10.1145/3673038.3673110

[15] NVIDIA, "NVLink-C2C," Accessed: 2025-03-30. [Online]. Available: https://www.nvidia.cn/data-center/nvlink-c2c/

[16] AMD, "amd-cdna-3-white-paper," Accessed: 2025-03-30. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf

[17] P. Gupta, "Accelerating datacenter workloads," Accessed: 2025-03-30. [Online]. Available: https://fpl2016.epfl.ch/slides/Gupta%20--%20Accelerating%20Datacenter%20Workloads.pdf

[18] CXL Consortium, "Compute Express Link Specification," https://www.computeexpresslink.org/download-the-specification, Accessed: 2025-04-1.

[19] OpenCAPI Specification Archive - Compute Express Link. Accessed: 2025-03-30. [Online]. Available: https://computeexpresslink.org/resource/opencapi-specification-archive/

[20] Gen-Z Specification Archive - Compute Express Link. Accessed: 2025-03-30. [Online]. Available: https://computeexpresslink.org/resource/gen-z-specification-archive/

[21] CCIX Specification Archive - Compute Express Link. Accessed: 2025-03-30. [Online]. Available: https://computeexpresslink.org/resource/ccix-specification-archive/

[22] Integrators list - Compute Express Link. Accessed: 2025-05-30. [Online]. Available: https://computeexpresslink.org/integrators-list/

[23] 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. [Online]. Available: http://dx.doi.org/10.1145/3613424.3614256

[24] Y. Tang, P. Zhou, W. Zhang, H. Hu, Q. Yang, H. Xiang, T. Liu, J. Shan, R. Huang, C. Zhao, C. Chen, H. Zhang, F. Liu, S. Zhang, X. Ding, and J. Chen, "Exploring performance and cost optimization with ASIC-based CXL memory," in *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*, 2024, p. 818–833. [Online]. Available: https://doi.org/10.1145/3627703.3650061

[25] J. Zeng, S. Pei, D. Zhang, Y. Zhou, A. Beygi, X. Yao, R. Kachare, T. Zhang, Z. Li, M. Nguyen, R. Pitchumani, Y. S. Ki, and C. Jung, "Performance characterizations and usage guidelines of Samsung CXL memory module hybrid prototype," 2025. [Online]. Available: https://arxiv.org/abs/2503.22017

[26] 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. [Online]. Available: http://dx.doi.org/10.1145/3582016.3582063

[27] 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*, vol. 2, 2023, pp. 574–587. [Online]. Available: https://dl.acm.org/doi/10.5555/3154630.3154683

[28] H. Kim, N. Wang, Q. Xia, J. Huang, A. Yazdanbakhsh, and N. S. Kim, "Lia: A single-GPU LLM inference acceleration with cooperative AMX-enabled CPU-GPU computation and CXL offloading," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, 2025, p. 544–558. [Online]. Available: https://doi.org/10.1145/3695053.3731092

[29] CMM-D CXL memory Samsung semiconductor global. Accessed: 2025-07-21. [Online]. Available: https://semiconductor.samsung.com/cxl-memory/cmm-d/

[30] CXL memory initiative. Accessed: 2025-07-21. [Online]. Available: https://www.rambus.com/cxl-memory-initiative/

[31] World's First CXL 2.0 and PCIe Gen 5.0 Switch Chip. [Online]. Available: https://www.xconn-tech.com/products

[32] SK hynix secures CXL 2.0-based CMM-DDR5 customer validation. Accessed: 2025-07-21. [Online]. Available: https://news.skhynix.com/sk-hynix-completes-customer-validation-of-cxl-based-ddr5/

[33] Montage, "Montage MXC," https://www.montage-tech.com/MXC, Accessed: 2025-07-21.

[34] Agilex™ 7 FPGA i-series development kit (2x r-tile and 1x f-tile). Accessed: 2025-07-21. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/agi027.html

[35] Compute express link (CXL) 2.0/1.1 FPGA IP. Accessed: 2025-07-26. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/cxl-ip.html

[36] Compute express link subsystem maturity map — the linux kernel documentation. Accessed: 2025-07-23. [Online]. Available: https://docs.kernel.org/driver-api/cxl/maturity-map.html

[37] OpenCL - the open standard for parallel programming of heterogeneous systems. Accessed: 2025-07-26. [Online]. Available: https://www.khronos.org/opencl/

[38] oneAPI programming model. Accessed: 2025-07-21. [Online]. Available: https://oneapi.io/

[39] J. Huang, J. Lou, S. Vanavasam, X. Kong, H. Ji, I. Jeong, D. Zhuo, E. K. Lee, and N. S. Kim, "HAL: Hardware-assisted Load Balancing for Energy-efficient SNIC-Host Cooperative Computing," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2024, pp. 613–627. [Online]. Available: https://ieeexplore.ieee.org/document/10609665

[40] C. Song, M. J. Kim, T. Wang, H. Ji, J. Huang, I. Jeong, J. Park, H. Nam, M. Wi, J. H. Ahn, and N. S. Kim, "Tarot: A CXL SmartNIC-based defense against multi-bit errors by row-hammer attacks," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 3*, 2024, p. 981–998. [Online]. Available: https://doi.org/10.1145/3620666.3651325

[41] B. Williams, J. Leidel, X. Wang, D. Donofrio, and Y. Chen, "Circustent: A benchmark suite for atomic memory operations," in *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, 2021, p. 144–157. [Online]. Available: https://doi.org/10.1145/3422575.3422789

[42] X. Wang, B. Williams, J. D. Leidel, A. Ehret, M. Kinsy, and Y. Chen, "Remote atomic extension (rae) for scalable high performance computing," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/9218589

[43] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, "Fast and general distributed transactions using RDMA and HTM," in *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016, pp. 1–17. [Online]. Available: https://doi.org/10.1145/2901318.2901349

[44] T. Ma, K. Chen, S. Ma, Z. Song, and Y. Wu, "Thinking more about RDMA memory semantics," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 456–467. [Online]. Available: https://ieeexplore.ieee.org/document/9556062

[45] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2016, p. 437–450. [Online]. Available: https://dl.acm.org/doi/10.5555/3026959.3027000

[46] S. Narravula, A. Marnidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda, "High performance distributed lock management services using network-based remote atomic operations," in *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, 2007, pp. 583–590. [Online]. Available: https://ieeexplore.ieee.org/document/4215426

[47] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, p. 445–456. [Online]. Available: https://doi.org/10.1109/PACT.2015.24

[48] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the RPC tax in datacenters," in *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, p. 407–420. [Online]. Available: https://doi.org/10.1145/3466752.3480055

[49] J. Zhang, H. Huang, X. Chen, X. Li, J. Zhao, M. Liu, and Z. Wang, "Rpcnic: Enabling efficient datacenter RPC offloading on PCIe-attached SmartNICs," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025, pp. 1379–1394. [Online]. Available: https://ieeexplore.ieee.org/document/10946709

[50] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, p. 36–51. [Online]. Available: https://doi.org/10.1145/3445814.3446696

[51] K. Seemakhupt, B. E. Stephens, S. Khan, S. Liu, H. Wassel, S. H. Yeganeh, A. C. Snoeren, A. Krishnamurthy, D. E. Culler, and H. M. Levy, "A cloud-scale characterization of remote procedure calls," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023, p. 498–514. [Online]. Available: https://doi.org/10.1145/3600006.3613156

[52] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, "A hardware accelerator for Protocol Buffers," in *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, p. 462–478. [Online]. Available: https://doi.org/10.1145/3466752.3480051

[53] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, p. 733–750. [Online]. Available: https://doi.org/10.1145/3373376.3378450

[54] W. Hou, J. Zhang, Z. Wang, and M. Liu, "Understanding routable PCIe performance for composable infrastructures," in *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024, pp. 1–16.

[55] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: the next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020, pp. 1–14. [Online]. Available: https://doi.org/10.1145/3342195.3387517

[56] Linux, "Heterogeneous Memory Management (HMM) - The Linux Kernel documentation," Accessed: 2025-03-30. [Online]. Available: https://docs.kernel.org/mm/hmm.html

[57] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-aware unified memory management in GPUs for irregular workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, p. 1357–1370. [Online]. Available: https://doi.org/10.1145/3373376.3378529

[58] 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. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[59] 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. [Online]. Available: https://doi.org/10.48550/arXiv.2007.03152

[60] gem5 community, "Ruby," Accessed: 2025-04-1. [Online]. Available: https://www.gem5.org/documentation/general_docs/ruby/

[61] V. Soria-Pardos, A. Armejach, T. Mück, D. Suárez-Gracia, J. Joao, A. Rico, and M. Moretó, "Dynamo: Improving parallelism through dynamic placement of atomic memory operations," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3579371.3589065

[62] T. Ma, M. Zhang, K. Chen, Z. Song, Y. Wu, and X. Qian, "Asymnvm: An efficient framework for implementing persistent data structures on asymmetric NVM architecture," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming*

Languages and Operating Systems (ASPLOS), 2020, p. 757–773. [Online]. Available: https://doi.org/10.1145/3373376.3378511

[63] Y. Zhang, M. Wang, W. Wang, Y. Mai, H. Huang, and Z. Yu, "Atomic cache: Enabling efficient fine-grained synchronization with relaxed memory consistency on GPGPUs through in-cache atomic operations," in *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, p. 671–685. [Online]. Available: https://doi.org/10.1109/MICRO61859.2024.00056

[64] B. Zhu, Y. Chen, Q. Wang, Y. Lu, and J. Shu, "Octopus+: An RDMA-enabled distributed persistent memory file system," *ACM Trans. Storage*, vol. 17, no. 3, Aug. 2021. [Online]. Available: https://doi.org/10.1145/3448418

[65] "Introduction to infiniband," Accessed: 2025-07-26. [Online]. Available: https://network.nvidia.com/pdf/whitepapers/IB_Intro_WP_190.pdf

[66] A. Asgharzadeh, J. M. Cebrian, A. Perais, S. Kaxiras, and A. Ros, "Free atomics: hardware atomic operations without fences," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022, p. 14–26. [Online]. Available: https://doi.org/10.1145/3470496.3527385

[67] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level PIM offloading in graph computing frameworks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 457–468. [Online]. Available: https://ieeexplore.ieee.org/document/7920847

[68] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, F. Feng, Y. Zhuang, F. Liu, P. Liu, X. Liu, Z. Wu, J. Wu, Z. Cao, C. Tian, J. Wu, J. Zhu, H. Wang, D. Cai, and J. Wu, "When cloud storage meets RDMA," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, Apr. 2021, pp. 519–533. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/gao

[69] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: a distributed framework for emerging ai applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2018, p. 561–577. [Online]. Available: https://dl.acm.org/doi/10.5555/3291168.3291210

[70] gRPC. Accessed: 2025-07-21. [Online]. Available: https://grpc.io/

[71] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," Accessed: 2025-07-21.

[72] Protocol Buffers. Accessed: 2025-07-21. [Online]. Available: https://protobuf.dev/

[73] Cap'n proto: Introduction. Accessed: 2025-07-21. [Online]. Available: https://capnproto.org/

[74] FlatBuffers docs. Accessed: 2025-07-21. [Online]. Available: https://flatbuffers.dev/

[75] 4th gen intel xeon processor scalable family, sapphire rapids. Accessed: 2025-07-21. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html

[76] Multichannel DMA FPGA IP for PCI express altera. Accessed: 2025-07-21. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/multichannel-dma-mcdma.html

[77] CLDEMOTE — cache line demote. Accessed: 2025-07-21. [Online]. Available: https://www.felixcloutier.com/x86/cldemote

[78] CLFLUSH — flush cache line. Accessed: 2025-07-21. [Online]. Available: https://www.felixcloutier.com/x86/clflush

[79] Y. Yang, P. Safayenikoo, J. Ma, T. A. Khan, and A. Quinn, "CXLMemSim," Accessed: Jan. 8, 2025. [Online]. Available: https://github.com/SlugLab/CXLMemSim.

[80] Y. Wang, L. Wu, W. Hong, Y. Ou, Z. Wang, S. Gao, J. Zhang, S. Ma, D. Dong, X. Qi, M. Lai, and N. Xiao, "CXL-DMSim: A full-system CXL disaggregated memory simulator with comprehensive silicon validation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–14, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/11153390

[81] 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. [Online]. Available: http://export.arxiv.org/abs/2405.10170

[82] T. Q. P. Developers, "QEMU," https://www.qemu.org/docs/master/system/devices/cxl.html, Accessed: 2025-04-1.

[83] J. Zhang, X. Chen, Y. Zhang, and Z. Wang, "Dmrpc: Disaggregated memory-aware datacenter RPC for data-intensive applications," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, 2024, pp. 3796–3809.

[84] 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. [Online]. Available: http://dx.doi.org/10.1145/3624062.3624175

[85] J. Wahlgren, M. Gokhale, and I. B. Peng, "Evaluating emerging CXL-enabled memory pooling for HPC systems," in *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 2022, pp. 11–20. [Online]. Available: https://ieeexplore.ieee.org/document/10024061

[86] J. Liu, H. Hadian, Y. Wang, D. S. Berger, M. Nguyen, X. Jian, S. H. Noh, and H. Li, "Systematic CXL memory characterization and performance analysis at scale," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*, 2025, p. 1203–1217. [Online]. Available: https://doi.org/10.1145/3676641.3715987

[87] Y. Sun, J. Kim, Z. Yu, J. Zhang, S. Chai, M. J. Kim, H. Nam, J. Park, E. Na, Y. Yuan, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim, "M5: Mastering page migration and memory management for CXL-based tiered memory systems," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*, 2025, p. 604–621. [Online]. Available: https://doi.org/10.1145/3676641.3711999

[88] M. Vuppalapati and R. Agarwal, "Tiered memory management: Access latency is the key!" in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP)*, 2024, p. 79–94. [Online]. Available: https://doi.org/10.1145/3694715.3695968

[89] L. Xiang, Z. Lin, W. Deng, H. Lu, J. Rao, Y. Yuan, and R. Wang, "Nomad: non-exclusive memory tiering via transactional page migration," in *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2024, pp. 1–17. [Online]. Available: https://www.usenix.org/system/files/osdi24-xiang.pdf

[90] Z. Zhou, Y. Chen, T. Zhang, Y. Wang, R. Shu, S. Xu, P. Cheng, L. Qu, Y. Xiong, J. Zhang, and G. Sun, "Neomem: Hardware/software co-design for CXL-native memory tiering," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 1518–1531. [Online]. Available: https://ieeexplore.ieee.org/document/10764702

[91] 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. [Online]. Available: https://doi.org/10.14778/3685800.3685809

[92] H. Liu, L. Zheng, Y. Huang, J. Zhou, C. Liu, R. Wang, X. Liao, H. Jin, and J. Xue, "Enabling efficient large recommendation model training with near CXL memory processing," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 382–395. [Online]. Available: https://ieeexplore.ieee.org/document/10609725

[93] S. Sano, Y. Bando, K. Hiwada, H. Kajihara, T. Suzuki, Y. Nakanishi, D. Taki, A. Kaneko, and T. Shiozawa, "GPU graph processing on CXL-based microsecond-latency external memory," in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, p. 962–972. [Online]. Available: https://doi.org/10.1145/3624062.3624173

[94] D. Xu, D. Kim, Y. Feng, H. Jeon, K. Shin, and D. Li, "Efficient Tensor Offloading for Large Deep-Learning Model Training based on Compute Express Link," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024, pp. 1–18. [Online]. Available: https://ieeexplore.ieee.org/document/10793192

[95] Y. Yuan, J. Huang, Y. Sun, T. Wang, J. Nelson, D. R. K. Ports, Y. Wang, R. Wang, C. Tai, and N. S. Kim, "Rambda: RDMA-driven acceleration framework for memory-intensive μs-scale datacenter

applications," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 499–515. [Online]. Available: https://ieeexplore.ieee.org/document/10071127

[96] D. Quinn, M. Nouri, N. Patel, J. Salihu, A. Salemi, S. Lee, H. Zamani, and M. Alian, "Accelerating retrieval-augmented generation," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 1*, 2025, p. 15–32. [Online]. Available: https://doi.org/10.1145/3669940.3707264

[97] Why does Intel Agilex® 7 R-Tile Compute Express Link* (CXL) 1.1/2.0 FPGA IP not include the ATS capability? Accessed: 2025-07-23. [Online]. Available: https://www.intel.com/content/www/us/en/support/programmable/articles/000097527.html

[98] S. Tamimi, F. Stock, A. Koch, A. Bernhardt, and I. Petrov, "An evaluation of using CCIX for cache-coherent Host-FPGA interfacing," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022, pp. 1–9. [Online]. Available: https://ieeexplore.ieee.org/document/9786103

[99] H. Li, J. Li, and A. Kaufmann, "Simbricks: end-to-end network system evaluation with modular simulation," in *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM)*, 2022, p. 380–396. [Online]. Available: https://doi.org/10.1145/3544216.3544253

[100] C. Shin, J. Song, H. Jang, D. Kim, J. Sung, T. Kwon, J. H. Ju, F. Liu, Y. Choi, and J. Lee, "Piccolo: Large-scale graph processing with fine-grained in-memory scatter-gather," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025, pp. 641–656.

[101] P. Yao, L. Zheng, Y. Huang, Q. Wang, C. Gui, Z. Zeng, X. Liao, H. Jin, and J. Xue, "Scalagraph: A scalable accelerator for massively parallel graph processing," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 199–212.