Past Memory Expansion: The Bandwidth Wall and the Road to Native Cluster Execution on Coherent CXL Fabrics

Peter P. Waskiewicz Jr (PJ)
Jump Trading
Chicago, IL, USA
pwaskiewicz@jumptrading.com · ppwaskie@kernel.org

Abstract

Compute Express Link (CXL) has drawn intense industry focus on memory-expansion devices that use CXL.mem to add capacity, and as large-scale AI models demand massive, distributed memory footprints, the conversation has shifted toward CXL.mem as the substrate for network-attached memory pools. The prevailing mental model treats a CXL memory device as a "far" NUMA node, implying that latency is the primary hurdle. I argue this model is latency-accurate but bandwidth-blind: CXL link bandwidth trails a single multi-channel DDR socket by roughly an order of magnitude, and the High-Bandwidth Memory that feeds AI accelerators by nearly two—a gap that widens once a link is shared across a pool. This disparity, not latency, reshapes the economics of scaling models across distributed memory. With CXL 2.0 only recently delivering hardware memory pooling, the work to build truly memory-coherent clusters remains ahead of us. This paper looks past simple expansion and asks what native cluster execution on a fully coherent CXL fabric requires across switching, firmware, and the Linux kernel's memory and networking subsystems. I close with an opinionated conjecture: coherence at cluster scale should be fabric-resident, distributed, and tiered, with edge home agents on the DPU.

Keywords

CXL, memory, NUMA, bandwidth, kernel, fabric, coherence, disaggregation, AI, DPU

Introduction

The datacenter's memory hierarchy is being pulled apart. As model sizes and context windows grow faster than any single server's memory can hold, the industry has reached for Compute Express Link as the interconnect that lets memory live outside the box that consumes it. CXL is a cache-coherent interconnect layered on the PCIe physical layer, and its CXL.mem sub-protocol lets a host issue ordinary load and store instructions against memory that physically resides on a remote device [6, 34]. A CXL Type-3 memory device surfaces to the operating system as a CPU-less NUMA node—and that is precisely where the trouble starts, because the analogy is taken too literally.

The "far NUMA node" framing carries an implicit claim: that the only thing separating device memory from local memory is latency, and that if the latency is tolerable, the rest is a software placement problem. The first half of that claim is correct. Real, on-silicon CXL memory adds only about two to three times the load-to-use latency of local DRAM [19, 35], comparable to a remote NUMA hop and roughly an order of magnitude better than RDMA. The second half is where the model fails. Latency is not the constraint that breaks at scale—bandwidth is. A single CXL device delivers a small fraction of the bandwidth of the multi-channel DDR socket it is meant to augment, and a tiny fraction of the HBM that feeds a GPU. Treating that device as "just a slower NUMA node" hides the one number that governs throughput for the workloads driving CXL adoption in the first place.

This paper argues that bandwidth, not latency, is the wall standing between CXL memory expansion and native cluster execution on a coherent fabric—and it examines what closing that gap would actually require. This paper will focus on:

- **The bandwidth argument, quantified precisely.** Why the "far NUMA node" model is latency-accurate but bandwidth-blind, with the per-device and per-core gap stated against each baseline—DDR, HBM, and a shared pool—rather than leaning on a single round number.
- **What CXL 2.0 pooling delivers today, and what it does not.** Partitioned (not shared) capacity, the Fabric Manager, and an honest accounting of the skeptic's case against pooling alongside the case for it.
- **The moonshot, layer by layer.** The concrete work required across switching/fabric infrastructure, firmware, and the Linux kernel's memory *and* networking subsystems to move from expansion to coherent cluster execution—each with current state, gaps, and tradeoffs.
- **Where the coherency manager should live.** The paper's centerpiece: separating the control-plane Fabric Manager from the data-plane coherency manager, comparing five candidate placements, and committing to a position—*Fabric-Resident Tiered Coherence with edge home agents*—offered as a conjecture with high-level protocol sketches to provoke discussion.
- **Why neither problem solves the other.** A direct answer to the question this paper invites: does solving coherence fix the bandwidth problem? It does not—and the doubled links and bundled ports of CXL 4.0 do not fix the coherence problem either. The two are separable, they demand

different mechanisms, and the argument for that separation is what justifies a tiered design.

I write this as a position paper, not a measurement study. The fabric I want to build—a multi-host, memory-coherent cluster that uses CXL itself as the coherency protocol—does not yet exist, and the contribution here is to say what it would take to build it and to stake out where I think the hard problems lie.

Background: CXL in Three Acts

CXL defines three sub-protocols multiplexed over one link: **CXL.io**, functionally PCIe, for discovery, configuration, and DMA; **CXL.cache**, which lets a device coherently cache host memory; and **CXL.mem**, which lets the host issue coherent loads and stores against device memory [6, 34]. Memory devices are Type-3 devices: they expose CXL.mem (and CXL.io for management) and present their capacity to the OS as a memoryless NUMA node [34, 35]. The progression the industry is now living through has three acts (Figure 1).

Expansion (CXL 1.1). A single Type-3 device attaches directly behind one host, adding capacity to that host. There is no sharing and no switch; the device is a private, slower tier of memory for its owner.

Pooling (CXL 2.0). A single switch layer lets several hosts draw capacity from a shared physical device. The device partitions into up to sixteen Logical Devices (an MLD, multi-logical device), plus one Fabric-Manager-owned LD; each LD carries its own LD-ID, appears to its host as an independent Type-3 device, and is bound to that host's virtual hierarchy by the Fabric Manager [6, 7]. The critical caveat—too often glossed—is that **2.0 pooling is partitioned, not shared.** Each host owns its LD's capacity as dedicated space. There is no hardware-coherent cross-host sharing of a region; the value proposition is statistical multiplexing of capacity, not collaboration over common data [6, 7]. A 2.0 switch connects hosts to devices but cannot connect to other switches, so the scale of a pool is bounded by one switch's port count [10, 34].

Fabrics (CXL 3.0). This is where coherent sharing becomes architecturally possible. 3.0 replaces the older bias-based coherency with directory/snoop-filter coherency and adds Back-Invalidation: a device can invalidate host caches over new BISnp/BIRsp channels, which is what enables an inclusive snoop filter and, crucially, hardware-enforced coherent shared memory across independent hosts [10, 34]. Switches can now connect to switches; Port-Based Routing (PBR) scales addressing to as many as 4,096 nodes, where a node may be a host, an accelerator, a PCIe device, or a Global Fabric-Attached Memory (GFAM) device—a fully disaggregated pool that up to 4,095 peer nodes reach directly or through switches, with the back-invalidate logic living on the GFAM device itself [10, 34]. Unordered I/O plus BI further allows device-to-device access without routing every transaction through a host, breaking the tree-topology bottleneck [10, 34].

Beyond 3.0. The specification has not stood still. CXL 3.1 added inter-host communication over fabric memory via Global Integrated Memory (GIM) and a Fabric Manager API for PBR switches [8]; 3.2 standardized the CXL Hotness Monitoring Unit (CHMU), device-side hot-page telemetry I return to below [11]; and 4.0 moved to a PCIe 7.0 PHY at 128 GT/s and introduced bundled ports, which spray traffic across aggregated physical ports for higher effective link bandwidth [12]. It is worth noticing what the newest revision's headline features are: bandwidth features.

The arc is real and the direction is set. But this coherent fabric exists far more completely on paper than in silicon, and the gap between the two is the subject of the rest of this paper.

The Bandwidth Wall

Latency Is Manageable—the Part the Industry Gets Right

It is worth conceding the strong form of the opposing view, because it is correct as far as it goes. On real CXL-ready systems, the load-to-use latency of a Type-3 device lands at roughly 214–271 ns, against roughly 78–116 ns for local DDR5 and 135–142 ns for a one-hop remote NUMA access [19, 35, 36] (Figure 2). That is a 2–3$\times$ penalty over local memory—the same order as a NUMA hop, and roughly an order of magnitude better than RDMA's 1.5–3 μs [19, 29]. Because CXL exposes native load/store at cache-line granularity, there is no read/write amplification and no page-fault path; the MICRO'23 "Demystifying CXL" study found that even complex microservice workloads see only marginal tail-latency increase with most of their pages on CXL [35]. The "far NUMA node" model is, on latency, accurate.

Two honest caveats belong here. Adding a switch and retimers pushes latency past $\sim$600 ns (each retimer costs roughly 10 ns per direction), and CXL exhibits worse *tail* latency than local or NUMA memory under load [16, 19]. Neither changes the conclusion: latency is a manageable engineering quantity, not a wall.

Bandwidth Is Where the Analogy Collapses

The NUMA analogy breaks on bandwidth, and it breaks hard. CXL 1.1 and 2.0 ride a PCIe 5.0 PHY at 32 GT/s per lane, about 3.94 GB/s usable, so an x16 device tops out near 64 GB/s unidirectional in theory—and real Type-3 devices deliver only 18–52 GB/s measured, 20–70% of their own backing DRAM's theoretical bandwidth [18, 20, 28, 35]. Compare that to the memory it is meant to sit beside (Figure 3): a 12-channel DDR5 socket on EPYC Genoa measures about 375 GB/s (460 GB/s theoretical), an 8-channel Xeon SPR socket about 307 GB/s theoretical, and a single H100's HBM3 about 3,350 GB/s [25, 28]. CXL 3.0 doubles the PHY with PCIe 6.0 (64 GT/s, PAM-4), and CXL 4.0 doubles it again with PCIe 7.0 (128 GT/s), adding bundled ports that aggregate several physical ports into one logical link [10, 12, 34]. The trajectory itself concedes the point of this section—each new revision's headline features are bandwidth mitigations—but shipping Type-3 silicon remains PCIe 5.0-class, bundled ports spend the host's finite lane and port budget, and the structural gap to multi-channel DDR—and the chasm to HBM—remains [10, 12, 34].

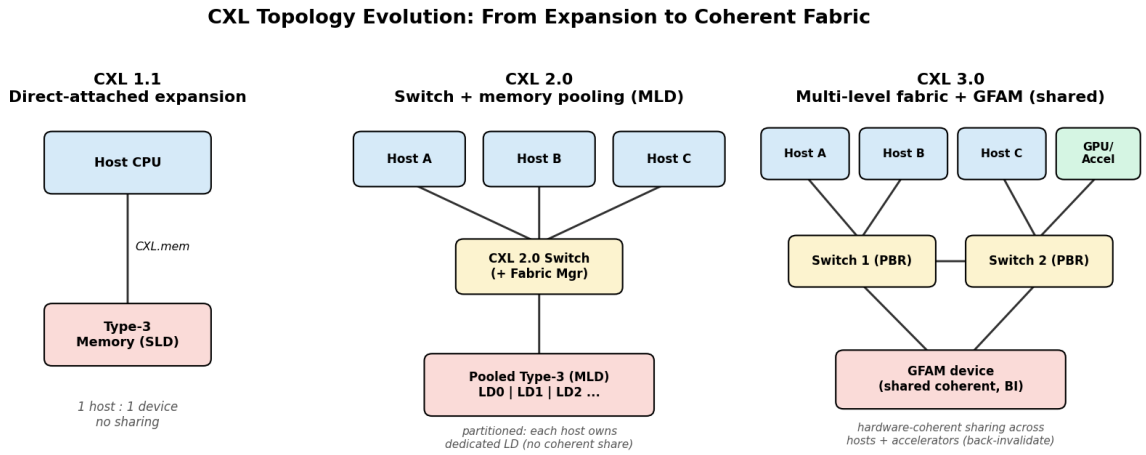

Figure 1: From direct-attached expansion (1.1) to switched pooling (2.0) to a multi-level coherent fabric with GFAM (3.0).

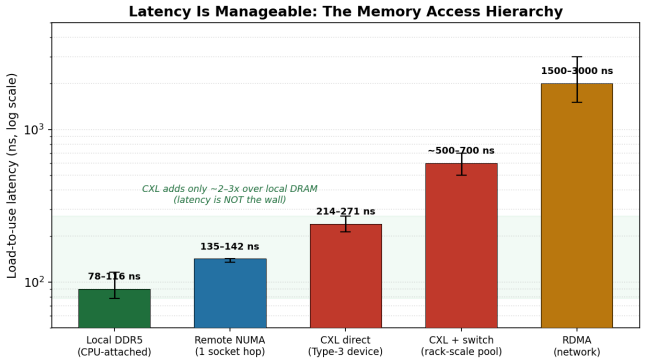

Figure 2: Latency hierarchy on a log scale. CXL sits between remote NUMA and RDMA.

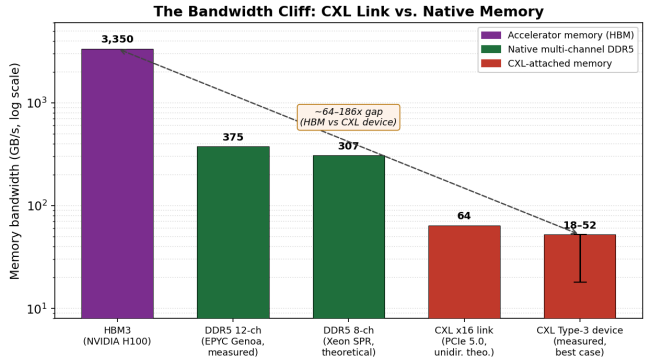

Figure 3: Per-device and per-socket bandwidth on a log scale.

Pooling Multiplies the Problem

A dedicated CXL device already trails a DDR socket. Put it in a shared pool and the situation worsens, because the single link's bandwidth is now divided across every host and core drawing on it. Per-core bandwidth—the metric that actually governs throughput on bandwidth-bound kernels—degrades as the pool fans out (Figure 4). A device that looked like "one slow NUMA node" to a single host becomes a small fraction of that to each of eight hosts sharing it. The obvious rebuttal—add more devices and interleave across them—runs into the same wall from the other side: aggregation is bounded by the host's finite PCIe lane budget, and in a pool every added consumer re-divides whatever aggregate the devices provide.

On "Orders of Magnitude"—Say It Precisely

A position paper is more credible when it refuses to round in its own favor, and the bandwidth gap is strong enough that it does not need rounding. Stated against each baseline: versus a single DDR5 socket (∼375 GB/s), a CXL device (18–52 GB/s) is about **7–20× slower**—roughly one order of magnitude. Versus HBM3 (∼3,350 GB/s), it is **64–186× slower**—nearly two. And against aggregate multi-socket DDR, a multi-HBM GPU node, or a fanned-out pool measured per consuming core, the gap comfortably exceeds two orders of magnitude. The defensible phrasing, which I adopt throughout: *CXL link bandwidth trails a single DDR socket by roughly an order of magnitude and the HBM that feeds AI accelerators by nearly two, and that gap widens once a link is shared.*

Why This Changes the Economics of AI

The reason the gap matters is that the workloads pushing CXL toward network-attached pools are the most bandwidth-hungry in the datacenter (Figure 5). Offloading model weights or KV-cache from HBM to CXL buys capacity by giving up roughly 50× in bandwidth—PCIe 5.0 x16 (∼64 GB/s) against H100 HBM3 (∼3,350 GB/s) means every offloaded byte is dramatically more expensive to fetch [25]. For a bandwidth-bound kernel this reframes scaling as a bandwidth-and-placement problem, not a capacity problem, which is the thesis of this paper. The most counterintuitive empirical anchor comes from real measurement: naïvely

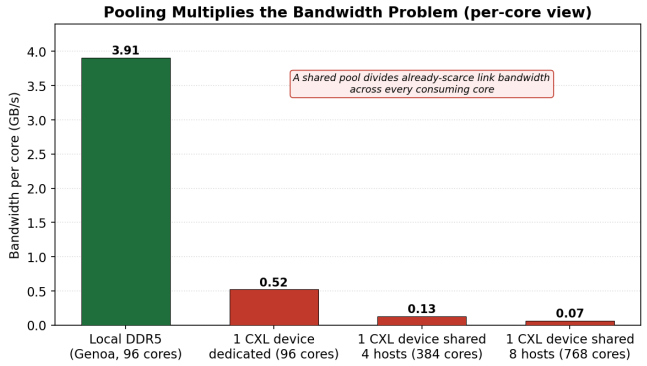

Figure 4: Illustrative per-core bandwidth (aggregate ÷ cores) for a Genoa DDR5 socket versus a CXL device dedicated and shared across 4 and 8 hosts. Illustrative because pool fan-out is a design choice, not a fixed spec.

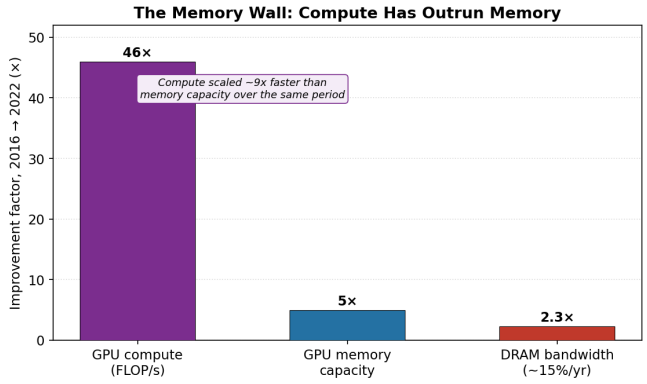

Figure 5: The growth imbalance that motivates disaggregated memory—GPU compute grew $\sim$46$\times$ from 2016–2022 while GPU memory capacity grew $\sim$5$\times$.

placing 50% of a bandwidth-intensive workload's pages on CXL can *reduce* throughput despite the higher aggregate bandwidth, because of how—and how unevenly—that bandwidth is delivered [35]. Capacity you cannot feed is not capacity you can use.

What Pooling Gives Us Today—and What It Doesn't

Before reaching for the moonshot it is worth being precise about what shipping CXL 2.0 actually provides, because the honest answer tempers some of the hype while sharpening the real motivation.

What 2.0 provides is *capacity multiplexing*. The Fabric Manager—standardized in 2.0—composes and dismantles resources by programming each component's Component Command Interface (CCI), and binds Logical Devices to host virtual hierarchies [6, 7]. A host that needs more memory than its packing allotment can be assigned an LD from the pool; a host that needs less releases it. This is genuinely useful in fleets where memory is stranded by VM packing, and it is the basis for the strongest published case in pooling's favor—Pond (ASPLOS'23), which shows that small pools of 8–16 sockets capture most of the achievable benefit at a latency cost of +75–90 ns, cutting datacenter DRAM by roughly 7% at 16 sockets and translating, at hyperscale, into hundreds of millions of dollars [17].

What 2.0 does *not* provide is coherent sharing. Two hosts bound to the same MLD do not share a coherent region; they share a device that has been carved into private slices. Collaboration over common data—the thing that would let a cluster execute against one logical memory image—is not in the 2.0 model at all [6, 7].

The Skeptic's Case, Stated Fairly

The strongest argument against memory pooling deserves to be in this paper, not paraphrased away. "A Case Against CXL Memory Pooling" (Google, HotNets'23) argues that pooling fails on cost, complexity, and utility [16]. On **utility**: on production traces from Google and Azure, modern servers are large relative to the VMs they host, so simple packing already strands little memory—pooling yields $\leq$1% utilization gain even at 16$\times$ VM inflation and $<$5% at 32$\times$. On **complexity**: extracting good performance requires software rewrites to explicitly stage CXL blocks into DRAM (bulk 8 KB copies approach DRAM speed, but fine-grained loads and pointer-chasing suffer, adding $\sim$140 ns), and the staging copies erode the very memory savings that justified the pool. On **cost**: a parallel cabling and switching plane alongside Ethernet can outweigh the RAM it saves [16].

I think this critique is correct *on its own terms*—and that its terms are the wrong ones for the case that matters here. The skeptic evaluates pooling-for-utilization: can we reclaim stranded RAM cheaply? The answer, for general cloud fleets, is often no. But the motivation for coherent CXL fabrics in AI is not utilization; it is **bandwidth and coherence for cluster execution**—feeding bandwidth-bound models too large for one node's memory, and letting many compute elements operate over shared state. That is a different and, I will argue, stronger motivation, and it is not addressed by the utilization analysis at all. The honest synthesis: pooling-for-utilization is contested and the skeptics may well be right; bandwidth-and-coherence-for-cluster-execution is a separate question, and it is the one this paper pursues.

The Moonshot, Layer by Layer

Going from expansion to native cluster execution on a coherent fabric is not one problem but three, spanning the layers the abstract names. I take each in turn: current state, the gaps that remain, and the tradeoffs of closing them.

Switching and Fabric Infrastructure

Current state. Multi-level, switch-to-switch fabrics with PBR exist in the 3.0 specification, but shipping multi-host *coherent* switch silicon is nascent; even 2.0 single-switch pooling is only now appearing in products [5, 7, 10].

Gaps. Physical reach is the hard limit nobody can specify away: PCIe/CXL signal integrity caps practical distance at roughly 2 m even with retimers [29]. This is the structural

reason CXL complements rather than replaces datacenter-wide RDMA—a coherent CXL domain is a rack-scale object, and anything larger must bridge to a different transport. Switch-induced latency (∼600 ns and up) and per-port bandwidth provisioning become first-order design constraints the moment a pool fans out [16, 19].

Tradeoff. A richer fabric buys coherent reach across more nodes at the cost of latency, silicon complexity, and a new failure-and-scaling domain centered on the switch—a tension that recurs sharply when I ask where coherence state should live.

Firmware—the Fabric Manager

Current state. The FM is standardized for inventory and binding. It reaches devices through the CCI, in-band over MMIO or out-of-band over MCTP (carried on I2C/I3C, PCIe, or USB), and the spec deliberately allows it to run almost anywhere: on a host, on a BMC, in switch firmware, or as a state machine on a device [6, 9]. CXL 3.1 added a Fabric Manager API for PBR switches [8], and above the spec, OCP's Composable Fabric Management defines a Redfish-over-network control plane for the same role [27].

Gaps. Two stand out. First, *dynamic, coherent, multi-tenant orchestration*—live rebind, QoS and bandwidth allocation, failure isolation—is immature; the FM today is closer to a provisioning tool than a runtime resource scheduler [6, 7]. Second, and most relevant to the kernel, **the host↔FM control path is out-of-band and explicitly out of CXL's scope** [6, 7]. A production fabric needs the operating system to participate in composition decisions, and there is no standardized in-band path for it to do so. Address-space management compounds this: re-mapping a device's address space to a different host can today require a system reset or a traffic quiesce, with the alternative being to map the whole physical space into every host and rely on the FM/VMM for isolation [14]. Either way the kernel is implicated.

Tradeoff. Pushing the FM toward a dynamic, in-band, kernel-integrated controller buys live reconfiguration and tighter isolation at the cost of a far larger trusted, security-sensitive control surface that now spans firmware and OS.

The Linux Kernel—Memory Subsystem

Current state—and it is further along than the fabric. The memory-tiering plumbing largely exists upstream. device-dax `system-ram` surfaces CXL as a NUMA node; reclaim-driven demotion has been present since 5.15; NUMA nodes are organized into explicit memory tiers [23, 31]. TPP (Transparent Page Placement, Meta/UMich) merged in **5.18**, doing transparent hot/cold promotion and demotion—roughly 18% better than stock Linux and within ∼1% of an all-local ideal in its authors' fleet tests [24]. **Weighted interleave** merged in **6.9**, spreading pages across nodes by bandwidth ratio (for example 5:1 DRAM:CXL)—notably a *bandwidth* mitigation rather than a capacity one [22, 23]. DAMON-based self-tuning tiering and DAMON-driven dynamic weighted interleave have followed, with ∼25% gains reported [23].

Gaps. Promotion is still too slow and too reactive; NUMA-balancing-based promotion lets performance decay while hot pages wait for migration [23]. Help is coming from the device side—CXL 3.2's Hotness Monitoring Unit (CHMU) gives the device itself a hot-page telemetry interface, and RFC patches wiring it into the kernel are on-list—but there is not yet a unified interface that drives promotion from device telemetry [11, 21]. Bandwidth-*aware* placement—treating link bandwidth, not just capacity or latency, as the scheduled resource—is early: research prototypes such as SupMario and Alto beat TPP by up to 177% with best-shot interleaving and amortized-latency tiering, but these are not upstream [18]. And the largest gap for this paper: coherent *shared*, multi-host CXL memory has **no first-class kernel model**—no standard semantics for cross-host shared regions, for ownership, or for fault handling.

Tradeoff. The capacity-tiering story is a success worth building on; the danger is assuming it generalizes. Bandwidth-aware, multi-host-coherent placement is a genuinely new problem, and bolting it onto machinery designed for single-host capacity tiering will not be sufficient.

The Linux Kernel—Networking Subsystem and Cluster Execution

This is the bridge from "tiered memory" to "cluster fabric," and the part that ties the moonshot back to the netdev core. When two hosts can share a coherent region, a class of network copies disappears: communication becomes a memory operation.

Current state. The evidence that this pays off is already concrete. cMPI (SC'25) reports CXL memory sharing beating TCP by up to **49× in latency and 72× in bandwidth**, and beating a SmartNIC by up to 48× in latency; against high-end RDMA, CXL is competitive for small and medium clusters though not a wholesale replacement [2, 26]. MPI-over-CXL maps shared memory into each rank's address space for pointer-based communication, leaving the MPI API unchanged while swapping the transport underneath [2].

Gaps—and these are squarely kernel and fabric work. Cross-node **atomicity** is often unenforceable on pooled memory [2]. The **software-vs-hardware coherence** tradeoff is stark: cache-flush-based software coherence adds about 2.8× latency, while hardware coherence does not scale to huge address ranges [2]—a tension I return to as the core of the conjecture below. **Shared-memory APIs** are a poor fit today: POSIX SHM and tmpfs were not built for this, and cMPI had to construct its own "CXL SHM Arena" to get usable semantics [2]. And **address-space remapping without resets** remains unsolved [14].

Tradeoff. Replacing the network stack with shared memory buys enormous latency and bandwidth wins within a coherent domain, but only inside the ∼2 m reach limit and only where atomicity and coherence semantics can be made safe—which is exactly the boundary the rest of this paper tries to define.

Where Does the Coherency Manager Live?

This is the crux of the moonshot, and the centerpiece of the paper. A multi-host coherent memory fabric that uses CXL itself as the coherency protocol does not exist yet, and the literature confirms it is an open problem rather than a solved

one. Before arguing for a position, I have to clear away a confusion the industry routinely creates by using one word—"manager"—for two entirely different things.

Two Managers—Do Not Conflate Them

The CXL Fabric Manager is a control-plane entity. It performs inventory, port bind and unbind, composition, QoS and telemetry configuration, security (LD erasure on unbind), and event handling. The spec explicitly lets it run as "software running on a host, embedded software running on a BMC, embedded firmware running on another CXL device or CXL switch, or a state machine running within the CXL device itself" [9]. It reaches devices via the CCI, in-band or out-of-band over MCTP, and the host↔FM path is out-of-band and outside CXL's scope [6, 9]. The essential point: **the FM is not in the coherence datapath.** It composes the fabric; it does not keep cache lines coherent.

The coherency manager is a data-plane entity—the home agent, directory, and snoop filter that actually enforce a consistent view of shared cache lines. This is the piece with no good cluster-scale answer today. CXL uses an *asymmetric* coherence model: the host CPU's home agent administers CXL.mem for HDM-H and HDM-D regions, while for HDM-DB the *device* implements the snoop filter and issues Back-Invalidate to host caches [33, 34]. So in CXL 3.0's shared-memory model, the coherence directory lives on the device—the GFAM or MLD—by default [10, 34]. The central design question of this paper is whether that default is the right place once no single host owns the fabric.

Five Places It Could Live

The question—where should the coherency manager live in a fabric no single host owns?—has at least five candidate answers, each a real position with published precedent or a clear open gap. Table 1 summarizes them.

On the device (the spec default). A device-resident snoop filter plus Back-Invalidate, exactly as CXL 3.0 specifies [10, 34]. It is spec-conformant, BI must live on the device regardless, and it adds no extra hop. It breaks at scale because a precise 64-byte snoop filter is impractical for a terabyte-scale FAM, while an imprecise (say 4 KB) filter generates spurious back-invalidation storms [15].

In the switch. A directory or snoop-forwarding engine in the leaf/spine switch ASIC, control plane on the switch CPU, data plane in silicon—the in-network approach of MIND (SOSP'21) and Concordia (FAST'21), which CXL 3.0 permits [32, 34, 37]. It centralizes coherence at the natural fan-out point with low added latency and one place to reason about ordering. It breaks because switch SRAM is tiny relative to the state needed, forcing ownership migration or compact copyset encoding, and it makes the switch a concentrated failure and scaling domain [32, 37].

On a host CPU. One CPU acts as home agent for a coherence domain, with others forming separate domains—essentially today's asymmetric model [34]. It works with shipping silicon and is familiar. It breaks because disaggregated access pays multi-hop latency, the CPU's Caching/Home Agent is already a contention point on real parts, and it contradicts the premise that no host owns the fabric [3, 32].

On a DPU or SmartNIC. A DPU at each host acts as a local home-agent and coherence proxy into the fabric, naturally pairing rack-local CXL coherence with cross-rack RDMA transport [29, 34]. This is the netdev-native angle: the coherence agent sits with the NIC, where the network already lives. Little has been published implementing it and cross-fabric coherence semantics are undefined—which is precisely the open space this paper steps into.

Hierarchical and distributed. Per-rack top-of-rack directories plus a global agent, with hardware-precise coherence for a small hot region and software or coarse coherence for the rest—Concordia's per-rack ToR directories [37], Cohet's local/global agents [3], and Jain's hybrid HW/SW split [15]. It is the only family of approaches that has actually been argued to scale out, and it matches datacenter topology. Its cost is coordination complexity: copyset encoding and inter-rack consistency are unsolved [37].

The Scaling Wall That Forces a Hybrid

One result cuts across all five options and constrains any honest answer: CXL 3.0's hardware coherence does not scale cleanly to terabyte shared pools. A precise per-64-byte-line snoop filter is impractical at FAM scale, and an imprecise one wastes bandwidth on spurious back-invalidations [15]. Every serious proposal therefore *splits* coherence—hardware for a small, hot, atomic-bearing region (locks, semaphores, queue-pairs, synchronization metadata) and software or coarse tracking for the bulk [15]. I take this as a position in its own right: **full hardware coherence over an entire cluster-scale fabric is the wrong goal. The right goal is a tiered coherency manager that is hardware-precise where it must be and software-managed where it can be.**

The Conjecture: Fabric-Resident Tiered Coherence with Edge Home Agents

I now stake out the opinionated centerpiece. It is deliberately a high-level conjecture with protocol sketches, not a wire-level specification—the point of a moonshot paper is to define a target worth arguing about, and to be honest about what remains open.

The core claim. A single, full-hardware coherence manager for a cluster-scale CXL fabric is the wrong target: it is a massive single point of failure and is almost certainly unbuildable at the scale and reliability a datacenter demands. The coherency function should instead be **fabric-resident, distributed, and tiered**—no one box owns coherence; the work is spread across the switch fabric, per-host edge agents, and the memory devices, with hardware precision spent only where it must be.

Four architectural pillars (Figure 6):

1. **Edge home agents on the DPU/SmartNIC.** Each host reaches the fabric through its DPU, which runs that host's *Edge Coherence Agent* (ECA): a home-agent proxy that tracks what its host has cached, issues and absorbs back-invalidates on the host's behalf, and co-locates the host's Fabric-Manager agent. This lifts the coherence burden

Table 1: Five candidate placements for the coherency manager.

Placement	What it looks like	Precedent	Strengths	Why it breaks at scale
On the device (GFAM/ MLD)	Device-resident snoop filter + BI (spec default)	CXL 3.0 spec [10, 34]	Spec-conformant; BI must live on device; no extra hop	Precise 64 B snoop filter impractical at TB FAM scale; imprecise (4 KB) → spurious BI storms [15]
In the switch	Directory / snoop forwarding in switch ASIC; control plane on switch CPU	MIND [32]; Concordia [37]	Centralizes coherence at fan-out point; low added latency; one ordering point	Tiny switch SRAM → ownership migration / copyset encoding; switch becomes failure + scaling domain [32, 37]
On a host CPU	One CPU is home agent for a domain; others separate domains	CXL asymmetric model today [34]	Works with shipping silicon; familiar	Multi-hop latency; CHA already a contention point; contradicts "no host owns the fabric" [3, 32]
On a DPU / SmartNIC	DPU per host = local home-agent/coherence proxy; CXL-in-rack + RDMA-cross-rack	Emerging/niche [29, 34]	Coherence agent sits with the NIC; natural cross-rack bridge (the netdev angle)	Little published implementation; cross-fabric coherence semantics undefined
Hierarchical / distributed	Per-rack ToR directory + global agent; hybrid HW (hot) + SW (bulk) coherence	Concordia [37]; Cohet [3]; Jain [15]	Only family argued to scale out; matches datacenter topology	Coordination complexity; copyset encoding + inter-rack consistency unsolved [37]

off the CPU's monolithic Caching/Home Agent—already a contention point on shipping silicon [3]—and places it exactly where the network already lives, the netdev-native angle. The ECA is also the natural cross-rack bridge: rack-local traffic stays CXL-coherent, while cross-rack traffic is tunneled over RDMA or a coherence-carrying transport, respecting CXL's ∼2 m reach limit [29].

2. **A distributed directory sharded across the switch fabric.** Coherence state—sharer/owner tracking and BISnp forwarding—lives in the leaf/spine switch ASICs, sharded by address, following the in-network-directory precedent of MIND and Concordia [32, 37] and permitted by CXL 3.0's switch-directory option [34]. The control plane (directory management, recovery) runs on a switch-local processor or BMC; the data plane is in the ASIC. Sharding is what removes the single point of failure: a lost leaf isolates its subtree, not the cluster.

3. **Tiered coherence on the GFAM devices.** The memory device exposes two region classes: a small, hardware-coherent **HOT region** (precise 64-byte snoop filter plus device-side Back-Invalidate), sized so the snoop filter is actually feasible, for locks, atomics, queue-pairs, and synchronization metadata; and a large, software- or coarse-managed **BULK region** (page-grained ownership, kernel-placed) for model weights, KV-cache, and datasets. This is the hybrid HW/SW split that the only scale-oriented proposals already endorse [15], made explicit as a first-class fabric feature rather than an implementation detail.

4. **Tier movement under kernel control.** A bulk line that becomes hot or contended can be *promoted* into the bounded hardware-coherent region, and demoted back when it cools—reusing the kernel's existing hotness-tracking machinery (DAMON/TPP-style), but now driving a *coherence* tier rather than only a memory tier.

High-level protocol sketches (conceptual; Figure 7 shows the write path):

- **Shared read, HOT region.** A host load miss reaches its ECA; if the line is absent, the ECA requests it from the owning leaf directory, which records the host as a sharer (I→S) and returns the line. Multiple readers yield multiple S entries.
- **Coherent write / ownership, HOT region.** A host store prompts its ECA to request ownership from the leaf directory (I→E). The directory issues Back-Invalidate to current sharers' ECAs; they acknowledge (writing back if dirty); the directory marks the writer as owner, syncs the device, and grants ownership (E/M); the store completes locally.
- **Bulk region, coarse coherence.** No per-line tracking. Ownership moves at page or region granularity, coordinated by the ECAs and the kernel; a producer flushes, and a consumer's ECA invalidates its coarse range before reading. This is where bandwidth-aware placement does its work.
- **Failure handling.** Because state is distributed, failures degrade gracefully rather than globally. ECAs log outstanding ownership; directory and device logging in the style of ReCXL [30] reconstructs state on recovery; a switch withholds responses for a failed node rather than serving stale data [30]. FM high availability uses primary/secondary instances with data-plane multipathing, as shipping vendor fabrics already do [13]. Who arbitrates errors that span tenants—BMC, OS, or FM—is genuinely unsettled in

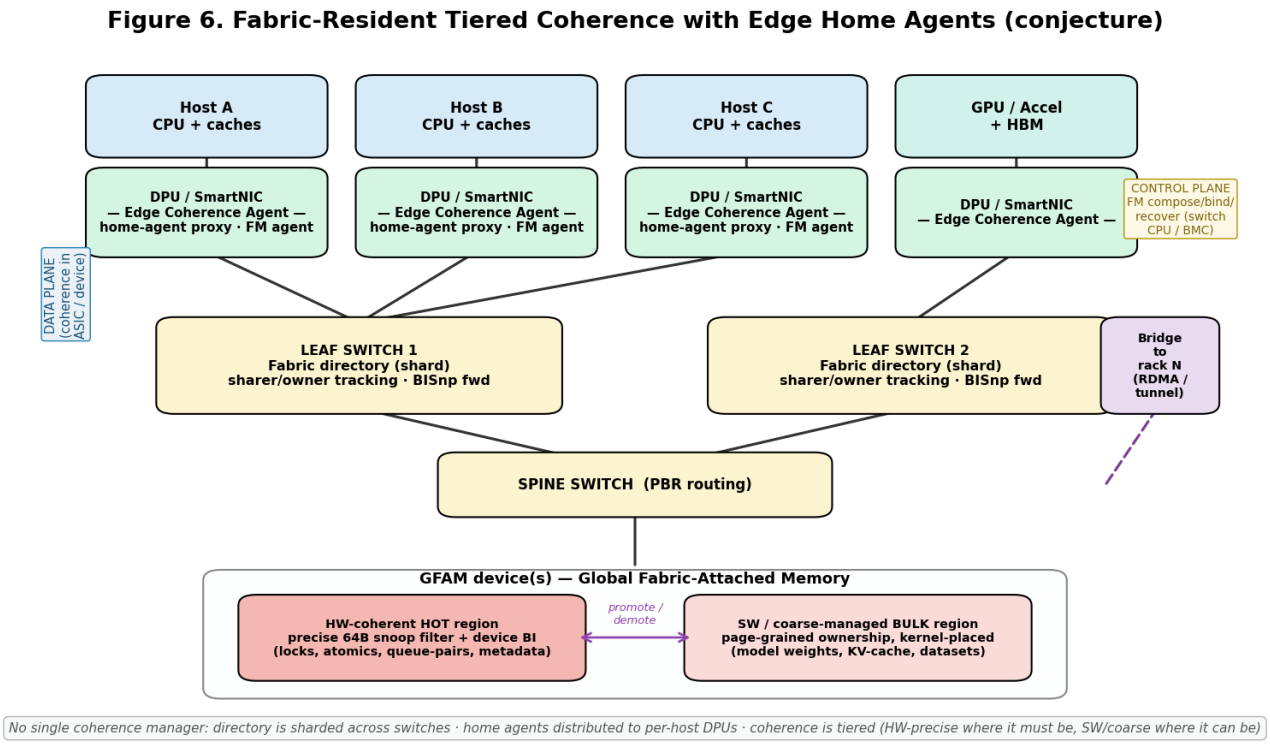

Figure 6: Fabric-Resident Tiered Coherence. Edge Coherence Agents on per-host DPUs, a coherence directory sharded across the switch fabric, and GFAM devices exposing a small hardware-coherent HOT region alongside a large software-managed BULK region.

multi-host CXL today [4]; in this design the FM remains the long-term arbiter while ECAs and host kernels monitor and report.

What this asks of the Linux kernel—and it lands on both subsystems the abstract names. In the **networking subsystem**: a driver and device model for the DPU/SmartNIC acting as a coherence agent, a role distinct from today's NIC offloads; the ECA needs kernel plumbing for ownership state, back-invalidate handling, and the rack-local-versus-cross-rack transport decision. In the **memory subsystem**: a way to declare and manage a region's *coherence class*—a HOT-versus-BULK attribute on a shared mapping, plausibly an extended `mmap` or region API—together with cgroup accounting for the scarce hardware-coherent region and promotion/demotion policy that now spans coherence tiers, not just memory tiers.

Open questions I will leave on the table—directory sharding and address-to-home mapping under multi-path routing; compact sharer/copyset encoding at thousands of nodes [37]; how large the HOT region can realistically be; consistency and atomicity guarantees across the HW/SW tier boundary; and a correctness model for the whole construction, for which CXL0 and CXLMC offer a starting formalism [1]. None of these is solved. That is what makes it a moonshot.

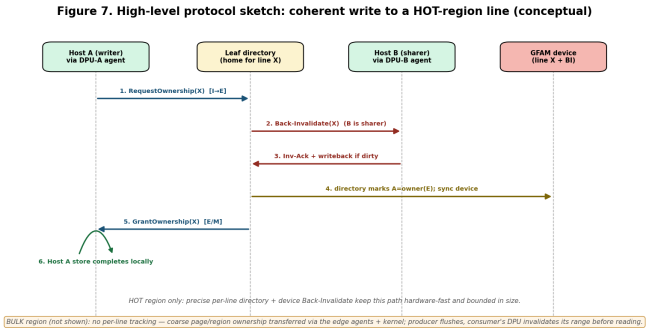

Figure 7: Coherent write in the HOT region—ownership request, directory-issued back-invalidate to sharers, writeback, ownership grant. Conceptual sketch, not a wire-level protocol.

Does Coherence Solve the Bandwidth Problem? (And Does Bandwidth Solve the Coherence Problem?)

Having argued that bandwidth is the wall and that coherence should be fabric-resident and tiered, I owe the reader an answer to the question those two claims invite: are they the same

problem? If we solve coherence, have we addressed the bandwidth disparity—or vice versa? The answer to both is no, and the reasoning is worth making explicit, because it is the strongest justification for the tiered design and the sharpest objection an audience should raise.

Coherence Cannot Widen the Pipe

Two observations settle the first direction. First, **no coherence protocol can increase link capacity.** Link bandwidth is fixed at the physical layer—lanes times signaling rate times encoding efficiency. An x16 PCIe 5.0 link is $\sim$64 GB/s unidirectional no matter what protocol runs above it [20]. Coherence is a protocol-layer construct; delivered bandwidth is bounded by the PHY regardless of how intelligently the protocol behaves. Second, **conservation of bytes**: any byte that must be consumed from far memory must cross the link at least once. If a core streams a gigabyte of model weights that live only in the CXL pool, a gigabyte crosses the link—coherent or not. The only mechanisms that beat that bound are placement (moving or replicating data near compute before it is needed, which spends the same link bandwidth, merely off the critical path) and demand reduction (touching fewer bytes). Neither is a coherence mechanism.

What coherence actually governs is not the size of the pipe but **how many times bytes cross it**—and that cuts both ways. Where data is *reused* or *shared*, coherence reduces crossings: hardware coherence is what makes caching far memory safe, so a line crosses once and then serves subsequent accesses from the local hierarchy; and coherent shared memory eliminates copy traffic outright, moving a pointer where message passing would move—and sometimes re-move—the payload. That, not any widening of the link, is the real mechanism behind cMPI's up-to-72$\times$ bandwidth advantage over TCP [2]: zero-copy sharing stops re-sending bytes, and cache-line granularity avoids fetching a 4 KB page to use 64 bytes of it. But where data is *not* reused, coherence only adds traffic: directory lookups and back-invalidation round trips ride the same starved link, imprecise snoop filters generate spurious invalidations by construction [15], and false sharing turns two hosts updating disjoint fields of one cache line into a bandwidth amplifier.

The decisive variable is therefore the **reuse factor**—and the workloads that motivated the bandwidth argument of this paper have almost none. Weight streaming and KV-cache scans touch every byte approximately once per pass; for them, coherence provides zero traffic reduction while still charging its protocol overhead. The traffic that *does* reward coherence is the opposite profile: locks, queue-pairs, allocator and index metadata—small, hot, and intensely shared. The reader will recognize this dichotomy: it is precisely the HOT/BULK split of the conjecture above. The tiered design is not an implementation convenience; it is the architectural expression of the fact that coherence pays for itself only where reuse lives, and actively burns scarce bandwidth where it does not.

Bandwidth Cannot Order the Bytes

The inverse question has become newly concrete. CXL 4.0's bundled ports aggregate multiple physical ports into one logical attachment, and the consortium's own material shows a bundled x16 at 128 GT/s reaching roughly 768 GB/s per direction—about 1.5 TB/s full-duplex [12]. That is twice the measured bandwidth of a 12-channel DDR5 socket [28] and within an order of magnitude of HBM3 [25]. The raw-capacity wall this paper opened with is, finally, being attacked where it must be attacked: at the PHY, with lanes and signaling—which is itself a quiet confirmation of the argument above, because the industry is solving bandwidth with wires, not with protocol.

But fat pipes without a coherence architecture are just faster private straws. Three caveats keep the trajectory honest. First, bundled ports as specified target host-to-accelerator (Type 1 and Type 2) attach; the pooled Type-3 path is not the headline beneficiary [12]. Second, the same revision extends reach toward multi-rack configurations—up to four retimers, optical support—without any accompanying multi-rack coherence story: the specification's own roadmap is racing ahead of its coherence architecture [12]. Third, and most importantly, **bandwidth scaling amplifies the coherence-plane load rather than relieving it.** At 768 GB/s a shared region can sustain an order of magnitude more line-ownership churn than at 64 GB/s; a snoop filter that was already impractical at terabyte scale [15] must now track it at ten times the rate, and a centralized directory that could hide behind a slow link becomes the visible serialization point. Every increment of PHY bandwidth raises the price of getting the coherence architecture wrong. This is the strongest form of the case for the conjecture: a monolithic coherence manager does not merely fail to scale—it fails *faster* as the links improve, while a sharded, tiered, edge-anchored design is the shape that lets the coherence plane scale alongside the PHY roadmap instead of becoming its new wall.

There is a satisfying symmetry in where this lands. CXL 4.0 holds latency flat while doubling bandwidth [12]—the specification is preserving the property everyone worried about and fixing the one this paper argues actually matters. Of the three axes, latency is managed and bandwidth is on a credible trajectory. The remaining unsolved axis is coherence at scale. Neither problem solves the other; they are separable, both are real, and they require different mechanisms—which is exactly why this paper proposes a coherence architecture and bandwidth-aware kernel placement as distinct, cooperating pieces rather than one mechanism pretending to do both jobs.

Toward Native Cluster Execution

With a coherence architecture sketched, the payoff comes into view: cluster execution where communication is a memory operation rather than a network operation. Within a coherent domain, the wins are large and already measured—cMPI's 49$\times$ latency and 72$\times$ bandwidth advantage over TCP is not a projection [2]. The specification is moving the same direction: CXL 3.1's Global Integrated Memory exists precisely to carry host-to-host communication over fabric memory [8]. The architecture above is what would let those wins extend past a single shared device to a rack-scale fabric: the HOT region carries the synchronization and queue-pair state that message passing needs to be safe, while the BULK region car-

ries the weights and KV-cache that bandwidth-aware placement must stage.

The boundary of the approach is as important as its center. CXL does not replace the network; it displaces it within roughly 2 m. The right mental model is a coherent rack-scale island bridged to the rest of the datacenter over RDMA or a coherence-carrying transport at the DPU—which is exactly why I place the edge home agent on the DPU, where that bridge already terminates. A principled placement model for AI weights and KV-cache across HBM, CXL, and RDMA, governed by the bandwidth constraints quantified earlier rather than by capacity alone, is the natural next research target.

Open Research Questions

The work this paper points to, stated as questions for the community:

- A first-class **kernel abstraction for coherent, multi-host shared CXL regions**—ownership, fault model, `mmap` semantics, and cgroup accounting.
- **Bandwidth-aware placement and migration** that schedules link bandwidth, not just capacity or latency, extending weighted interleave and DAMON with device-side bandwidth telemetry.
- Closing the **host↔Fabric-Manager gap** with an in-band, kernel-integrated control path for dynamic rebind, QoS, and isolation.
- **Reset-free address-space remapping** for live pool reconfiguration.
- **Convergence of the networking and memory subsystems**: when is shared-memory transport the right substitute for the network stack, and what kernel primitives—atomics, coherence control, SHM arenas—make it safe?
- A principled **CXL-vs-RDMA-vs-HBM placement model** for AI weights and KV-cache under the bandwidth constraints quantified above.

Conclusion

The industry is right that a CXL memory device behaves like a far NUMA node on latency, and wrong to stop the analysis there. The constraint that breaks at scale is bandwidth: a CXL link trails a single DDR socket by roughly an order of magnitude and the HBM that feeds AI accelerators by nearly two, and that gap widens once the link is shared. That single fact reframes scaling large models across distributed memory as a bandwidth-and-placement problem, not a capacity problem—and it means the road from memory expansion to native cluster execution runs through coherence, not just capacity. The two problems do not collapse into one: coherence cannot widen the pipe, and the widening pipe of CXL 4.0 only raises the stakes for getting coherence right.

Building that road is genuinely a moonshot, and it spans the three layers a netdev audience cares about: switching silicon that can be coherent at rack scale, firmware that can orchestrate it dynamically and in-band, and a Linux kernel that can place memory by bandwidth and manage coherence as a tiered, multi-host resource across both its memory and networking subsystems. My position on the hardest piece—where coherence lives—is that it should be fabric-resident, distributed, and tiered, with edge home agents on the DPU, a directory sharded across the switch fabric, and hardware precision reserved for a small hot region while the bulk is managed in software. I offer it as a conjecture, not a finished design, precisely so the community can argue with it. As with my prior work, my intent is to bring this toward the upstream conversation: the kernel abstractions this fabric needs do not exist yet, and the right place to design them is in the open.

Acknowledgments

I would like to thank the Netdev program committee for the opportunity to present this work, and the colleagues whose discussions on CXL, coherence, and disaggregated memory shaped these positions.

References

[1] Assa, R.; Feldman, Y. M. Y.; et al. 2024. A programming model for disaggregated memory over CXL (CXL0); CXLMC: Model checking CXL shared memory programs. arXiv:2407.16300; ASPLOS '26.

[2] cMPI authors. 2025. cMPI: Using CXL memory sharing for MPI one-sided and two-sided inter-node communications. In *SC '25*. arXiv:2510.05476.

[3] Cohet authors. 2025. Cohet: A CXL-driven coherent heterogeneous computing architecture. *arXiv:2511.23011*.

[4] CXL Consortium and industry. 2024. CXL fabric management standards and multi-host RAS arbitration; CXL DCD multi-host demo. computeexpresslink.org.

[5] CXL Consortium and Kao. 2024. CXL 2.0 switch for a composable memory system. In *FMS 2024*.

[6] CXL Consortium. 2020. Compute express link 2.0 specification: Memory pooling. computeexpresslink.org.

[7] CXL Consortium. 2021. CXL fabric management; memory pooling webinar Q&A (parts 1–2). computeexpresslink.org.

[8] CXL Consortium. 2023a. Compute express link 3.1 specification and white paper: Global integrated memory (GIM), FM API for PBR switches, trusted security protocol. computeexpresslink.org.

[9] CXL Consortium. 2023b. CXL 3.0 fabric management: CCI/MCTP and FM placement options. computeexpresslink.org; cxl-fabric-manager (GitHub).

[10] CXL Consortium. 2023c. CXL 3.0 white paper; introducing the CXL 3.x specification. computeexpresslink.org.

[11] CXL Consortium. 2024. Compute express link 3.2 specification: CXL hotness monitoring unit and memory-device monitoring and management. computeexpresslink.org.

[12] CXL Consortium. 2025. Compute express link 4.0 specification and white paper: PCIe 7.0 PHY at 128 GT/s, bundled ports, RAS enhancements. computeexpresslink.org.

[13] IntelliProp. 2023. Composable memory requires a fabric; omega memory fabric manager. intelliprop.com.

[14] Jain, R.; Yeleswarapu, N.; Maruf, H. A.; and Gupta, R. 2024a. Memory sharing with CXL: Hardware and software design approaches. In *HCDS '24*. arXiv:2404.03245.

[15] Jain, S.; Yeleswarapu, N.; Maruf, H. A.; and Gupta, R. 2024b. Memory sharing with CXL: Hardware and software design approaches. In *HCDS '24*. arXiv:2404.03245.

[16] Levis, P.; Lin, K.; and Tai, A. 2023. A case against CXL memory pooling. In *HotNets '23*. DOI:10.1145/3626111.3628195.

[17] Li, H.; Berger, D. S.; Hsu, L.; Ernst, D.; Zardoshti, P.; et al. 2023. Pond: CXL-based memory pooling systems for cloud platforms. In *ASPLOS '23*. arXiv:2203.00241.

[18] Liu, J., et al. 2025. Systematic CXL memory characterization (SupMario, Alto). In *ASPLOS '25*. arXiv:2409.14317.

[19] Liu, J.; Hadian, H.; Xu, H.; Berger, D. S.; and Li, H. 2024. Dissecting CXL memory performance at scale: Analysis, modeling, and optimization. *arXiv:2409.14317*.

[20] Logic Fruit and PCIe primers. 2022. CXL vs PCIe gen 5: Per-lane PHY bandwidth. Logic Fruit; AEWIN/OnLogic.

[21] LWN.net / LSFMM; Jonathan Cameron and others. 2025. CXL device-side hot-page detection: LSFMM coverage and CXL hotness monitoring unit (CHMU) perf driver RFC patches. lwn.net; linux-cxl mailing list.

[22] LWN.net / MemVerge. 2024. Weighted interleave merged in linux 6.9. lwn.net; memverge.com.

[23] LWN.net. 2024. Weighted interleaving for memory tiering; better support for locally-attached-memory tiering. lwn.net; DAMON project.

[24] Maruf, H. A.; Wang, H.; Dhanotia, A.; Weiner, J.; Agarwal, N.; et al. 2023. TPP: Transparent page placement for CXL-enabled tiered-memory. In *ASPLOS '23*. arXiv:2206.02878; merged in Linux 5.18.

[25] Micron and industry reports. 2024. LLM memory wall and HBM: Inference reports; H100 HBM3 $\approx$3.35 tb/s. Micron; Google Cloud; Oracle Cloud.

[26] MPI-over-CXL authors. 2025. MPI-over-CXL: Enhancing communication efficiency in distributed HPC systems. *arXiv:2510.14622*.

[27] Open Compute Project. 2023. Composable fabric management (CFM) architecture specification, rev 1.1. opencompute.org.

[28] PCI-SIG and platform vendor reports. 2023. PCIe 5.0 / DDR5 platform bandwidth: Genoa 12-channel ($\sim$375 gb/s measured), SPR 8-channel. chipsandcheese; ServeTheHome; TechPowerUp.

[29] Rcmp authors. 2024. Rcmp: Reconstructing RDMA-based memory disaggregation via CXL. *ACM TACO*.

[30] ReCXL authors. 2026. ReCXL: Towards CXL resilience to CPU failures. *arXiv:2602.08271*.

[31] Scargall, S. 2024. Using linux kernel tiering with CXL memory. Technical report.

[32] seob Lee, S.; Yu, Y.; Tang, Y.; Khandelwal, A.; Zhong, L.; and Bhattacharjee, A. 2021. MIND: In-network memory management for disaggregated data centers. In *SOSP '21*. arXiv:2107.00164.

[33] Sharma, D. D., et al. 2024. An introduction to the compute express link (CXL) interconnect. *ACM Computing Surveys*. DOI:10.1145/3669900.

[34] Sharma, D. D.; Blankenship, R.; et al. 2023. An introduction to the compute express link (CXL) interconnect. *arXiv:2306.11227*.

[35] Sun, Y.; Yuan, Y.; Yu, Z.; Kuper, R.; Song, C.; Huang, J.; et al. 2023. Demystifying CXL memory with genuine CXL-ready systems and devices. In *MICRO '23*. arXiv:2303.15375.

[36] The Next Platform. 2022. Just how bad is CXL memory latency? nextplatform.com.

[37] Wang, Q.; Lu, Y.; Xu, E.; Li, J.; Chen, Y.; and Shu, J. 2021. Concordia: Distributed shared memory with in-network cache coherence. In *USENIX FAST '21*.

Author Biography

PJ Waskiewicz is a Systems Architect and Linux kernel engineer at Jump Trading. He previously worked as a kernel and networking architect at NetApp's SolidFire division, and for many years as a network kernel engineer and device-driver developer in Intel's Networking Division, where he maintained and helped create the igb, ixgbe, and i40e wired Ethernet drivers, added the initial Tx multiqueue support to the Linux network stack, and brought Data Center Bridging to the kernel. He also worked in Intel's Open Source Technology Center on the x86 kernel tree, enabling kernel features in the Broadwell and Skylake microarchitectures.