Octopus: Enhancing CXL Memory Pods via Sparse Topology

Yuhong Zhong Fiodar Kazhamiaka Pantea Zardoshti Shuwei Teng
Rodrigo Fonseca Mark D. Hill Daniel S. Berger

Columbia University *Microsoft Azure* *University of Wisconsin–Madison* *University of Washington*

Abstract

The Compute Express Link (CXL) interconnect enables compute "pods" that pool memory across servers to reduce cost and improve efficiency. These pods also facilitate pairwise communication whose needs conflict with pooling. Importantly, existing pod designs are small or require indirection through expensive switches. These conventional designs implicitly assume that pods must fully connect all servers to all CXL pooling devices.

This paper breaks with this conventional wisdom by introducing *Octopus* pods. Octopus directly connects servers to low-port-count CXL pooling devices (e.g., 4 ports) yet scales to large pods without switches by constructing a *sparse* CXL topology in which each pooling device connects to a carefully chosen subset of servers. Octopus explicitly balances "overlap", where two servers connect to the same pooling device: overlap reduces pooling efficiency but enables low-latency communication. Octopus resolves this tension by grouping servers into "islands" with low-latency intra-island communication and interconnecting islands to favor pooling.

We build a three-server CXL pod prototype and simulate scaled pods with 96 servers under measured device characteristics and physical constraints (1.5 m copper cables). On hardware, Octopus RPCs are 3.2× faster than in-rack RDMA and 2.4× faster than CXL switches. In simulation, Octopus achieves net server cost savings of 3–5.4% whereas CXL switches result in a net cost increase.

1 Introduction

Cloud memory pressure. General-purpose cloud systems face a growing memory shortfall. Compute scales with die area (n^2), but the DRAM a socket can attach over DDR4/5 is "beachfront"-limited ($4n$). As core counts and per-core performance have driven per-socket compute up by 5–10× over the past decade, socket-attached DRAM capacity has grown only 2–3× [43, 73, 98]. Right-sizing cores or pushing more work to scale-out helps, but the most cost-effective lever is: *attach more memory to the same socket*.

CXL memory expansion is the first step. Compute Express Link (CXL) provides a practical path today to add capacity with CPU load/store semantics [40]. CXL-attached DRAM ("expansion") is accessed like local memory (`ld/st`); it is supported by major CPU platforms [16, 20, 36] and shipping servers that double memory capacity [79, 113, 123, 126, 128].

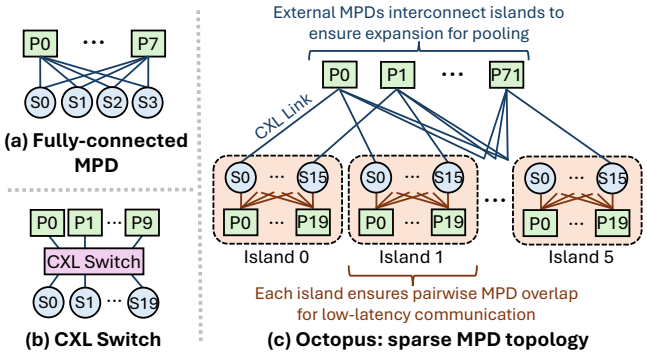


Figure 1: High-level view of **(a)** a fully-connected MPD pod, **(b)** a CXL switch pod, and **(c)** an Octopus pod with six islands (96 servers ("S") and 192 MPDs ("P")). Within each Octopus island, servers use a subset of CXL ports to connect to island-specific MPDs, guaranteeing pairwise MPD overlap for low-latency communication. Remaining server ports connect to external MPDs that interconnect islands and enable effective memory pooling.

While expansion adds latency (≈230 ns vs. 115 ns), many web, key-value, analytics, and database workloads tolerate this NUMA-like penalty [18, 84, 147].

This paper explores CXL "pods" spanning multiple servers with CXL as a scale-up interconnect. Pods enable *memory pooling*, which reduces DRAM spend (often half of server cost) by multiplexing capacity demand across servers [18, 43, 74, 82, 137, 142]. The intra-pod CXL fabric also creates a *low-latency communication and data sharing* fabric that improves performance of several workload classes. Specifically, pod-wide RPCs, coordination, and collective-based data movement can be sped up by sharing memory buffers over the CXL fabric [17, 23, 91, 125, 138, 145, 146, 148, 149]. Realizing these promises, however, largely depends on the CXL fabric's performance and topology.

Existing pod options and their tradeoffs. Two practical ways exist to connect servers to CXL memory modules:

- *Multi-ported devices (MPDs)* collapse a tiny on-device interconnect with the memory controllers so multiple servers can attach directly, *without a CXL switch* [42, 48, 50, 58, 82, 90, 136]. Today's MPDs are small (2–4 ports; ≈270 ns access) [30, 42, 53, 70, 92, 114, 139], and thus thought to limit pods to a handful of servers. Prior work assumes *fully-connected* pods, in which every MPD

connects to every server, to enable hardware interleaving across MPDs for higher bandwidth [42,48,50,82,90,136].

- *CXL switches* expand connectivity (dozens of ports), but add (de)serialization hops, raising latency and server capital expenditure (CapEx) [24,40,81].

However, both options have key drawbacks: fully-connected MPDs limit pooling benefits due to their small pod sizes, whereas CXL switches increase latency and CapEx to the point where switch costs can outweigh the pooling savings.

Our approach: sparse MPD topologies. We address the pod size limitations of MPD topologies by breaking from prior work and considering *sparsely connected* servers and MPDs. In a sparsely-connected topology, each server connects to multiple MPDs, which in turn connect to *different sets of servers*. This allows pods to scale beyond the MPD port count, e.g., dozens of servers with 4-port MPDs, without the expense and latency of CXL switches [81].

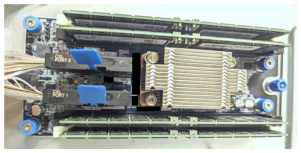
We propose *Octopus*, a deployable, cost-conscious CXL pod design built on sparse MPD topologies that targets both memory pooling and low-latency communication. Octopus accounts for practical CXL constraints, including the length of copper CXL cables, port count limits of commodity MPDs, and the number of CXL ports available on modern servers.

However, designing sparse MPD topologies that simultaneously support both use cases is challenging. Low-latency communication and memory pooling have conflicting requirements in terms of *MPD overlap*, i.e., whether servers connect to a common MPD. Low-latency communication favors pairwise MPD overlap between any two servers, whereas memory pooling disfavors MPD overlap among "hot" servers so that their excess memory demand can be spread across as many MPDs as possible (§5.1).

To overcome this challenge, we observe that low-latency communication is typically needed only at modest scale (e.g., one dozen of servers), whereas effective memory pooling requires much larger scale (dozens to hundreds of servers). Octopus therefore organizes each pod into multiple *islands* (Figure 1). Within each island, servers use a subset of ports to connect to island-specific MPDs arranged using Balanced Incomplete Block Design (BIBD) [38,103], which guarantees pairwise MPD overlap for low-latency communication. The remaining server ports are used for interconnecting islands via additional MPDs, limiting MPD overlap among hot servers and enabling effective memory pooling across the pod (§5.2).

We implement a minimal prototype of Octopus on real hardware spanning three servers and three MPDs. Octopus closely matches the performance of existing CXL memory expansion systems, where a CXL switch would otherwise increase the slowdown of 25% of applications by 3×. Octopus's communication latency is 3.2× lower than in-rack RDMA, 2.4× lower than a CXL switch, and 4.5× lower than other sparse MPD topologies with good memory pooling savings.

We simulate scaled-up Octopus topologies of 25, 64, and 96 servers based on 4-port MPDs. Octopus achieves similarly low intra-island communication latency as fully-connected MPD topologies. Octopus's savings from memory pooling alone can pay for the extra device costs both in cases with existing CXL expansion (5.4% overall cost reduction) and without CXL in place (3.0% overall cost reduction). CXL switches always cost more, even after factoring in their pooling savings. We open source Octopus pods and cost models at https://github.com/yuhong-zhong/Octopus-CXL-Pod.



Device	P50
CXL expansion	230–270 ns
CXL 2/4-port MPD	260–300 ns
CXL switch	490–600 ns
RDMA via ToR	3550 ns

Figure 2: Left: example two-port CXL device. **Right:** Load-to-use read latency (P50) on random 64-byte cachelines accessed via different CXL devices and RDMA on Intel Xeon 6 and AMD Turin.

While CXL can outperform Ethernet and RDMA, it does not replace them. As with GPU pods, we expect inter-pod communication to still use Ethernet or RDMA. Unlike GPU pods, however, CXL use cases are cost sensitive; our work shows how to make the intra-pod CXL fabric cost-effective across multiple use cases.

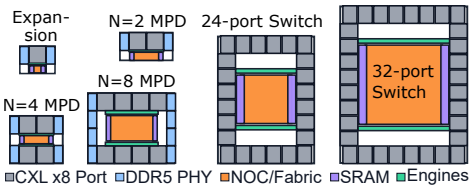
Contributions. We make the following contributions:

- We propose Octopus, the first CXL pod design based on sparse MPD topologies, challenging the assumption that CXL pods must fully connect servers and pooling devices.
- We systematize practical CXL pod constraints and identify key MPD topology properties for CXL use cases.
- We evaluate Octopus using real sparse MPD hardware and simulation, and use a cost model to compare cost savings across CXL pod designs.

2 State of CXL in 2026

The Compute Express Link (CXL) interconnect standard itself is described in detail in a recent tutorial [40]. We discuss real devices and constraints when using CXL.mem today. When a CPU issues a `ld` to an address associated with a CXL device and CPU caches do not have the data, the CPU sends a CXL.mem read *flit* to the device. CXL.mem flits use PCIe physical lanes and electrical signaling but implement custom low-latency protocol layers. The device reads the cache line from its (DDR4 or DDR5) DRAM and sends it back to the CPU, which places it into processor caches and the core.

Latency and bandwidth. On Intel Xeon 6 platforms, a local DDR5 read takes about 115 ns. Reading from a good CXL.mem expansion device takes 200–300 ns. Using a CXL bus analyzer, we measure that 75–170 ns of this latency is due to the CPU (most of the variability), 65 ns is due to CPU port round-trips and flight time, 25 ns is due to device-internal



Type	CXL×8	DDR5	Area [mm^2]	Price [$]
Expansion	1	2	16	200
MPD	2	2	18	240
MPD	4	4	32	510
MPD	8	8	64	2,650
Switch	24	0	120	5,230
Switch	32	0	209	7,400

CXL Cable Length [m]	AWG	Price [$]
0.50	30	23
0.75	28	29
1.00	28	36
1.25	26	55
1.50	26	75

Figure 3: Cost model for CXL devices and cables. **Left:** Die area estimates for devices with different CXL and DDR5 configurations. **Middle:** Prices based on a yield and markup model. Die area figures are simplified but based on a real 8-port MPD layout and modern IO dies. **Right:** Cable prices are based on copper material prices and markup.

processing, and 35–40 ns is due to DRAM access. On AMD Turin platforms, performance is similar. Devices typically offer multiple DDR4 or DDR5 DRAM channels.

A single CPU typically has multiple CXL ports. For example, Intel Xeon 6 production platforms use 64 CXL lanes per CPU socket, e.g., [62,66,78,105,127]. A single ×8 CXL port[1] offers 25–30 GiB/s of read-only bandwidth, and twice that for ×16. The CPU socket's CXL lanes are configurable as four ×16 CXL ports or eight ×8 CXL ports. In aggregate, a CPU socket can thus read from CXL devices at 200–240 GiB/s.

Device types. There are three CXL.mem device types:

① An *expansion device* offers a single CXL port and exposes its memory to a single CPU.

② A *multi-ported device (MPD)* integrates N CXL ports, allowing N CPUs to connect to the same controller and access its memory concurrently. Both expansion devices and MPDs are available today, and some products support both modes. $N = 2$ MPDs with two ×8 ports (e.g., Figure 2) are offered by AsteraLabs [70], Marvell [92], Google and Meta [30], and xFusion [53]; Seagate and SKH have $N = 4$ prototypes [42, 114, 121, 139]. On our lab system, our $N = 2$ MPD offers 267 ns read latency and 28 GiB/s read bandwidth per port.

③ A *CXL switch* offers up to 32 ×8 CXL ports and can forward CXL packets between them, e.g., the XC50256 switch offered by XConn [140]. In today's CXL 2.0 standard, switches connect to servers and expansion devices, but cannot connect to other switches. For every flit round-trip, a switch must deserialize and reserialize the flit twice, which adds at least 220 ns of latency [60] (Figure 2). This added latency also reduces effective bandwidth by increasing the bandwidth-delay product, as CPUs often support limited outstanding requests to fully utilize the link [85, 87].

CXL pods. A *CXL pod* [54, 55] is a group of servers connected to shared CXL devices, enabling direct access to device-attached memory and supporting memory pooling and shared memory buffers across the pod. There are two common ways to build a CXL pod: ① One approach uses MPDs, where each MPD connects directly to several servers; prior MPD-based pods assume a fully-connected design, in which every MPD connects to every server within the pod [82]. Using MPDs avoids expensive switches and keeps latency low, but limits pod size by the MPD port count. ② The other approach uses CXL switches [82, 143], which fan out connectivity so many servers can connect to many devices; this supports larger pods but incurs higher latency and cost.

Physical constraints. CXL devices connect to the CPU via PCIe5 pins and cables, whose reach is limited by insertion loss. At 16 GHz, the total budget is 36 dB; after the CPU package, motherboard, and MPD board consume ≈ 26 dB, only ≈ 10 dB remains for the cable and connectors, *constraining cable lengths to* $\leq 1.5\,m$ [109, 129]. Longer runs would require retimers [21] or optical cables [6], both of which add latency, power, or cost.

3 Modeling CXL Device Cost

To understand the cost tradeoffs of different CXL pod designs, we model the cost of CXL devices and cables. As is common with datacenter server components, vendor prices are subject to non-disclosure agreements [22]. We thus rely on a die area model that captures typical IO-pad limitations [122].

We estimate die area and cost using three data points. First, we use a 5 nm design and its die area for an MPD with eight ×8 CXL ports and eight DDR5 channels, as proposed by ARM [50]. Second, we scale die-area estimates from the 6 nm AMD Zen 4 (Genoa) I/O die [97] and infer die cost by interpolating the I/O-die contribution from CPUs with different core counts. Third, we incorporate pricing for today's expansion devices, 2-port devices, and switches based on discussions with supply-chain experts and public numbers [143]. In addition, we model cable costs using copper and assembly prices.

MPDs. Figure 3 shows our die area estimates, CXL device costs, and cable costs. The cheapest device is a single-ported (×8) expansion device with two DDR5 channels, at $200. Compared to expansion devices, we provision MPDs with twice the number of CXL ports per DDR5 channel (i.e., one ×8 CXL port for each DDR5 channel), and assume this CXL-port-to-DDR-channel ratio for all MPDs. This makes an $N =$ 2-port MPD slightly more expensive than an expansion device, which is pessimistic because we know that in practice many expansion devices actually use one ×16 that can be bifurcated into two ×8 ports. An $N = 4$ MPD amortizes some of the fixed area overheads, but we assume that vendors would charge a

[1] A ×8 CXL port has 8 CXL lanes. Similarly, a ×16 port has 16 lanes.

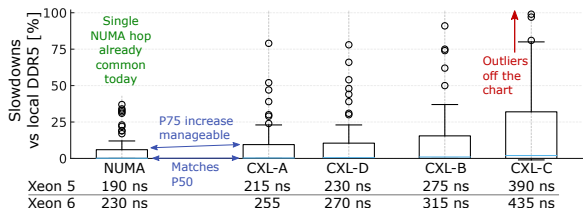


Figure 4: Box plots of workload slowdowns under different CXL latencies show that an increasing fraction of workloads sees slowdown around 390 ns on Xeon 5, which is equivalent to 435 ns on Xeon 6.

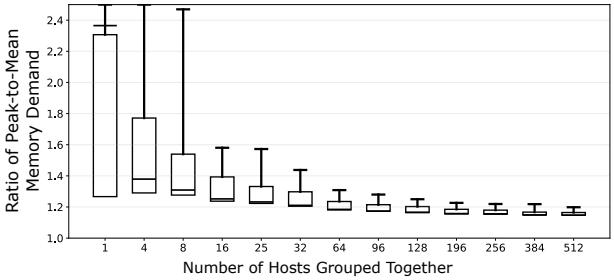


Figure 5: Ratio of peak memory demand to average memory demand across groups of servers of different sizes show that we need to group large numbers of servers to reduce outliers [87]. Even groups of 25-32 servers would still need about 1.5× memory capacity to handle peaks. This data is based on VM traces from Azure, gathered from ten production clusters over two weeks [108].

slightly higher markup. At $N = 8$, MPDs are IO-pad limited and prices increase significantly in our model.

Switches. Our cost model shows that switches with 24 or 32 ports are substantially more expensive than MPDs: a 32-port switch would cost $7,400. A recent paper reports that the XConn XC50256, a 32 ×8-port CXL switch, is priced at approximately $5,800 [143]. This gap is largely explained by process technology: while our model assumes 5 nm or 6 nm nodes for consistency with modern CPUs, CXL switches are often fabricated on mature nodes such as TSMC 16 nm to reduce wafer and NRE costs [5]. Even at 16 nm, switches remain an order of magnitude more expensive than MPDs.

Power. MPD-based CXL pods are also more power efficient than switch-based designs. Using a simple additive model where each CXL port consumes 2 W and assuming $X = 8$ CXL ports per server, MPD pods incur about 72 W per server versus 89.6 W for switch-based pods (24% more), due to the switch silicon and expansion devices. Although the per-server difference is modest (about 3% of total power at 500 W), the overhead compounds at scale.

4 CXL Use Cases

We briefly review CXL memory capacity expansion (§4.1), which is already deployed in production [2, 25], as well as memory pooling (§4.2) and shared memory use cases (§4.3).

4.1 Memory Capacity Expansion

DDR5 memory capacity per unit socket performance has decreased over time, as core counts grow faster than the number of available DDR channels [25]. This is a structural trend: DDR and other parallel memory interfaces (LPDDR, HBM) consume large bundles of pins, routing layers, and on-package PHY area per channel, limiting how many channels fit on a package. Individual DIMMs are also capacity-limited, and high-capacity parts carry steep $/GiB premiums (e.g., 3D stacking with TSV [51, 80]). CXL addresses both constraints by using serial links that deliver 5–10× more capacity per pin than DDR, decoupling the memory controller from the CPU package and enabling fan-out. Practically, CXL can 2–2.5× socket memory capacity today, and CXL-based memory expansion is already deployed in production in Azure [2, 25].

CXL memory expansion is latency sensitive. Figure 4 shows that application slowdowns on CXL significantly increase for devices at around 400 ns of latency, based on data from prior work [84]. Adapting applications [18, 74], OS tiering [41, 75, 86, 116, 135, 141], or hardware tiering [30, 147] can only partially offset these slowdowns, especially at the tail. Thus both datacenters and workloads look for low-latency access to CXL memory.

4.2 Memory Capacity Pooling

Servers are typically provisioned with enough memory capacity to handle peak demands. Unfortunately, capacity demand is highly variable across time and servers which leads to a large gap between average demand and peak [82, 108], causing low memory utilization. Pooling capacity aims to *multiplex these per-server peaks* so that capacity can be provisioned much closer to the average demand, thus provisioning less memory overall and improving memory utilization. CXL enables memory pooling within a pod by allocating CXL-attached memory to servers (e.g., at 1 GiB granularity [82]) and dynamically changing this allocation over time.

Prior work has shown significant cost savings from CXL capacity pooling and that these savings *increase with the number of servers within a pod*. Figure 5 shows that as pod size increases, aggregate peak memory demand within a pod approaches the aggregate average, with diminishing returns beyond about 96 servers. Thus, effectively reducing demand outliers requires pooling across dozens of servers.

Beyond pod size, CXL access latency also affects pooling savings, since higher latency reduces the fraction of workloads that can tolerate CXL memory and thus limits how much memory can be pooled. Using workload sensitivity to CXL latency (Figure 4), the measured latencies of MPDs and switches (Figure 2), and assuming a tolerable slowdown of 10%, we estimate that 65% of memory can be pooled and provisioned from MPDs, compared to 35% when using switches.

Parameter	Description
X	Number of CXL ports per server
N	Number of CXL ports per MPD
S	Number of servers in a pod
M	Number of MPDs in a pod
X_i	Number of server ports to island-specific MPDs

Table 1: MPD topology notation.

4.3 Practical Shared Memory Use Cases

Besides memory pooling, where CXL-attached memory is allocated exclusively to individual servers, CXL today also allows multiple servers to concurrently access *shared memory buffers* on CXL devices[2]. Shared CXL memory enables low-latency communication between servers via load/store instructions, and eliminates the need to serialize large and complex data structures when sharing them between servers, as required by Ethernet and RDMA.

Low-latency communication. Communication between two servers in a CXL pod can be implemented by having one server write to a message queue in a shared CXL buffer while the other busy-polls it [149]. In the best case, communication requires one CXL write and one CXL read, totaling roughly 600 ns, which is orders of magnitude lower than typical RDMA latencies in datacenters (tens of µs [110]).

This is especially useful for distributed systems that rely on frequent small messages, including consensus protocols such as Viewstamped Replication [83], ZooKeeper [56], Raft [101], proposer-acceptor messaging in Paxos [72], and HotStuff [144]. More broadly, even complex communication patterns are commonly expressed using message-passing interfaces [32, 37, 88, 117].

Many high-availability scale-out systems operate at *modest cluster sizes, typically between 3 and 16 servers*: MySQL InnoDB Cluster (3–7 nodes) [102], MongoDB replica sets (3–7) [95], SAP HANA TDI (up to 16) [111, 112], Amazon Aurora (up to 15 read replicas) [19], Redis Cluster (6 nodes) [107], Bigtable (3+ nodes) [46], and primary-backup replication [33, 94, 99, 130, 146, 150].

Eliminating serialization. Sharing large and complex data structures over Ethernet and RDMA requires expensive serialization and compression. Prior work shows that these overheads account for 6–10% of CPU time in RPC-heavy services and up to 20% for small 1 KB payloads [57, 61, 89, 91, 104, 106, 115, 124, 145]. In CXL pods, *these serialization and compression overheads can be avoided* by allocating pairwise shared CXL buffers between servers [23, 89, 91, 145, 146].

[2]CXL devices available today do *not* support cross-server cache coherence. Although the CXL 3.0 specification introduces Back-Invalidate (BI), an optional cross-server hardware coherence mechanism [3], it requires significant hardware changes on both processors and devices.

MPD Topology	Pooling	Communication Latency
Fully-connected [82] (S=4)	Poor	Low (4)
BIBD [38, 103] (S=25)	Poor	Low (25)
Expander [120, 133] (S=96)	Optimal	High
Octopus (S=96)	**Near Optimal**	**Low (16)**

Table 2: Memory pooling effectiveness and communication latency of MPD topologies with $N = 4$ and $X \leq 8$ constraints. "Low (k)" denotes low-latency communication among k servers, while "High" indicates multi-hop server-level forwarding.

5 Octopus Design

Rather than relying on CXL switches, Octopus builds on MPDs as a low latency and cost-effective building block for CXL pods. We seek to fulfill the following three goals:

1. **Effective memory pooling**, which requires a large number of interconnected servers within a CXL pod (§4.2).
2. **Low-latency communication**, which requires servers to directly share an MPD (§4.3).
3. **Practical deployability**, which limits us to short copper CXL cables ($\leq$ 1.5 m), MPDs with $N = 4$ ports, and at most eight CXL links per server (§2).

Octopus uses *sparse MPD topologies*, where each MPD connects to only a subset of servers, enabling larger pods despite limited MPD port counts. However, designing sparse MPD topologies that simultaneously satisfy goals #1 and #2 is challenging because they favor conflicting properties.

We first describe the conflicting topology requirements for effective memory pooling and low-latency communication (§5.1). We then present Octopus's logical server-to-MPD topologies (§5.2), physical layout in racks (§5.3), and software stack (§5.4).

5.1 Tension & Balancing Use Cases

To show the tension between memory pooling (goal #1) and low-latency communication (goal #2), we model a CXL pod as a bipartite graph with two vertex sets: servers and MPDs. Edges represent CXL links between servers and MPDs. Each server has degree X, corresponding to the number of CXL ports per server, and each MPD has degree N, corresponding to the number of ports per MPD. The notation is summarized in Table 1. For example, a fully-connected topology is a complete bipartite graph where the MPD port count (N) equals the number of servers (S). In contrast, in a sparse topology, S can be much larger than N.

The tension between memory pooling and low-latency communication is because they have conflicting requirements in terms of *MPD overlap*, i.e., whether servers connect to a common MPD. We will show why they favor opposing overlap properties as well as why existing topologies (e.g., datacenter network topologies [31, 38, 47, 103, 120, 133]) fail to balance the tension.

5.1.1 Communication Favors Pairwise Overlap

Low-latency communication over CXL requires pairs of servers to be able to issue load/store instructions to the same MPD directly. Although servers without a common MPD can still communicate through intermediate hops by forwarding messages through other servers and MPDs, we find that this server-level forwarding loses CXL's latency advantages over RDMA (§6.2).

Therefore, the ideal graph property for low-latency communication is *pairwise MPD overlap*, where every pair of servers in a pod connects to at least one common MPD.

The best graphs for this property are Balanced Incomplete Block Design (BIBD) graphs [29, 44, 118], which have also been used for datacenter networks [38, 103]. BIBD is a classical combinatorial design used to construct bipartite graphs in which every pair of vertices on one side (servers) connects to exactly one common vertex on the other side (MPDs)[3].

With $N = 4$-port MPDs and using $X \leq 8$ ports per server, BIBD yields three pod topologies with different sizes: 13 servers ($X = 4$), 16 servers ($X = 5$), and 25 servers ($X = 8$). Note that the 25-server graph is the largest BIBD graph with $N = 4$ and $X \leq 8$, implying that larger pods cannot satisfy the pairwise overlap property. As we show later (Table 2), although the 25-server BIBD graph maximizes the number of servers that can communicate at low latency (goal #2) under practical CXL constraints (goal #3), it performs poorly for memory pooling (goal #1).

5.1.2 Pooling Disfavors Overlap Among Hot Servers

Memory pooling reduces overall memory capacity by multiplexing server demand peaks (§4.2). Pooling savings are determined by the peak memory usage across MPDs because MPDs must be provisioned to handle the worst-case load. Therefore, our goal is to minimize the *peak memory usage across MPDs* in a pod.

Memory demand in datacenters is highly spiky across servers [49, 82, 108]. As a result, peak MPD usage is often dominated by a small subset of "hot" servers with high memory demand in a pod. To maximize pooling savings, the topology should therefore *avoid MPD overlap among worst-case hot server sets*, allowing their excess demand to be spread across as many distinct MPDs as possible, since demand patterns are not known in advance.

MPD overlap among servers can be characterized using graph *expansion* [52]. For a given subset size k, we define the graph's expansion e_k as the minimum number of distinct MPDs connected to any subset of k servers [52, 133]. Expansion directly yields a lower bound on peak MPD usage[4]. *Expander* graphs, including random graphs (e.g., Jellyfish [120]) and Ramanujan graphs (e.g., Xpander [133]), provide asymptotically optimal expansion for fixed X and N [52]. Moreover, larger expander graphs achieve better expansion than smaller ones. Under practical cabling constraints (goal #3), expander graphs with up to 96 servers are feasible (§5.3).

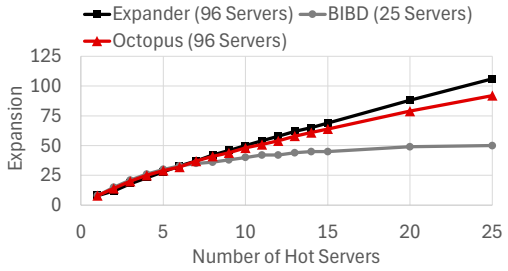


Figure 6: Expansion across topologies as the number of hot servers varies. Expansion denotes the number of distinct MPDs connected to the worst-case hot server set.

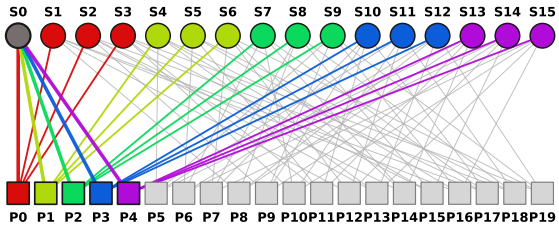


Figure 7: An Octopus island guarantees pairwise MPD overlap, i.e., every pair of servers connects to exactly one common MPD. The MPDs and links connecting server "S0" to other servers are highlighted; the same applies to each server.

Figure 6 shows that a 96-server expander graph achieves stronger expansion than the 25-server BIBD graph. While expander graphs are well suited for memory pooling, they do not satisfy the pairwise overlap property required for low-latency communication. In the 96-server expander graph, worst-case communication between two servers requires forwarding through two intermediate servers (Table 2).

5.2 Logical Network Topology

To balance the tension between memory pooling and low-latency communication, we observe that low-latency communication operates at smaller scales (e.g., 16 servers (§4.3)) than effective memory pooling, which begins to show meaningful gains at around 64 servers (§6.3).

Octopus leverages this observation by organizing each pod into multiple *islands* of highly connected servers that provide low-latency intra-island communication, while maintaining sufficient inter-island connectivity to increase expansion and enable effective memory pooling (Figure 1).

Each island satisfies the pairwise MPD overlap property. A subset of X_i server ports on each server is dedicated to connecting to *island-specific MPDs*, while the remaining $X - X_i$ ports connect to additional MPDs that interconnect islands and provide the expansion required for pooling.

[3] BIBD uses the parameter λ to control the number of vertices on one side of the bipartite graph shared by each pair of vertices on the other side. We set $\lambda = 1$ to enforce the pairwise overlap requirement.

[4] The theorem and its proof appear in Appendix A.1.

# islands	# servers per island	Server count (S)	MPD count (M)
1	25	25	50
4	16	64	128
6	**16**	**96**	**192**

Table 3: Octopus defines MPD pod designs parameterized by island count. The default pod (shown in **bold**) has 6 islands and 96 servers. All designs use X=8 ports per server and N=4-port MPDs.

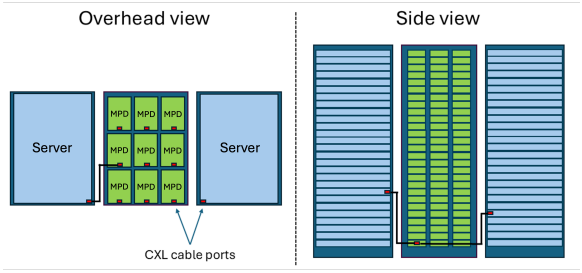


Figure 8: A 3-rack Octopus configuration.

One might expect Octopus's hierarchical design to reduce expansion. However, Figure 6 shows that a 96-server Octopus topology achieves expansion close to that of a 96-server expander graph. Octopus trades off the size of the low-latency communication domain to achieve this expansion. In particular, using $X_i = 8$ yields an island of 25 servers, while we empirically find that $X_i = 5$ provides strong expansion with islands of 16 servers, a 36% reduction in low-latency communication domain. This tradeoff is reasonable given the typical cluster sizes for low-latency communication (§4).

Octopus forms a family of MPD pod designs parameterized by the number of islands (Table 3). Octopus includes a single-island configuration with 25 servers, where all server ports are used for intra-island connectivity. For multi-island pods, we allocate $X_i = 5$ ports per server for intra-island connectivity, yielding feasible pod sizes of 64 (four islands) and 96 (six islands). We use the 96-server pod by default, as it achieves near-optimal memory pooling (goal #1) and low-latency communication within each island (goal #2) while satisfying deployability constraints (goal #3).

We next describe how islands are constructed and interconnected in more detail.

5.2.1 Intra-Island Connectivity

Each island is a subgraph in which every pair of servers connects to exactly one common MPD (§5.1.1), enabling low-latency communication within the island without intermediate hops (Figure 7). In multi-island Octopus pods, each island is a BIBD graph with 16 servers and 20 island-specific MPDs, where each server uses $X_i = 5$ ports to connect to the island-specific MPDs. The larger 25-server BIBD graph (§5.1.1) is used only in the single-island Octopus configuration, as it consumes all eight CXL ports per server for island-specific MPDs, leaving no ports available for inter-island connectivity.

5.2.2 Inter-Island Connectivity

The main purpose of inter-island connectivity is to increase the expansion of hot server sets for memory pooling. We achieve this using dedicated "external" MPDs and the remaining $X - X_i$ server ports. With $X_i = 5$, each island has 48 external links. In the 96-server Octopus pod with six islands, this corresponds to 72 external MPDs, or 37.5% of all MPDs.

Designing inter-island connectivity is a challenging combinatorial problem. A natural starting heuristic is to connect each external MPD to servers from *different* islands. However, achieving good expansion requires enforcing strong symmetry. Specifically, each pair of islands must share the same number of external MPDs, and any two servers from different islands should share at most one external MPD to bound worst-case MPD overlap. In short, the design must maintain uniformity across islands, island pairs, and server pairs.

We address this design problem using a two-level approach. At the first level, we select the set of islands connected by each external MPD using a block design similar to BIBD. When an exact solution is not feasible, we fall back to a round-robin heuristic. At the second level, we assign servers from the selected islands to the specific ports of each MPD. With $X_i = 5$, each server has three external ports, and we perform the assignment in three rounds such that each server is used exactly once per round. We additionally enforce that any two servers from different islands share at most one external MPD.

5.3 Physical Layout

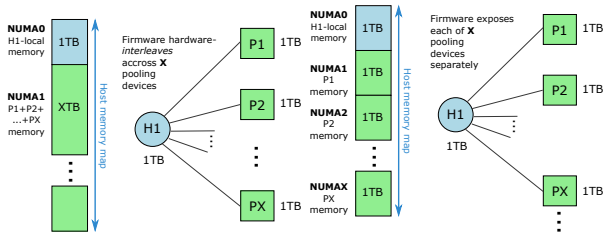
The physical layout of an Octopus pod is largely driven by CXL cable lengths, cable routing, and constraints of typical datacenter rack layouts. Reasonable cable lengths should be less than 1.5 m (§2). To deploy pods with up to 96 servers in a datacenter row, we propose a 3-rack pod designs where MPDs are places in the middle rack and servers placed in adjacent racks. Figure 8 sketches this design. The middle rack can hold a varying number of MPDs in each slot, depending on the MPD size (e.g., $N = 4$ vs $N = 8$). The dimension of each standard rack slot is approximately $100 \times 60 \times 5$ cm.

To further optimize cable lengths, we assume servers where CXL edge connectors are located at the front corner of the server chassis closest to the MPD rack, directly on the motherboard. This is a similar requirement to those imposed by the OCP NIC 3.0 specification [100] which also requires PCIe5 ports at the front of the server. Based on existing designs, we assume that CXL ports on the MPD side are in the front-middle of each MPD.

We validate achievable Octopus pod designs in §6.4.

5.4 Octopus Software Stack

API and exposure. In fully-connected MPD pods, server firmware interleaves MPDs at 256 B granularity [34], yielding



(a) Fully-connected. **(b)** Octopus (sparse).

Figure 9: Host view of CXL memory under interleaving vs. Octopus.

one large pool (Figure 9a). Octopus disables this interleaving in firmware and exposes each CXL port as a distinct NUMA node (Figure 9b). This lets software target a specific MPD for capacity balancing and for sharing with peer servers that connect to the same MPD. A datacenter control plane (e.g., Borg/Protean-like) assigns server IDs and disseminates the MPD pod topology and each server's MPD set [49, 131].

Pooling policy. When allocating CXL memory, each server allocates from the *least-loaded* connected MPD, prioritizing those with the most available capacity. This reduces allocation failures caused by individual MPDs becoming fully utilized, without requiring global defragmentation.

6 Evaluation

Our evaluation of Octopus includes a small-scale hardware prototype evaluation (§6.2), scaled-up simulations (§6.3), physical layout validation (§6.4), and cost modeling (§6.5).

6.1 Evaluation Setup

Hardware. We have access to three pre-production MPDs with two ×8 CXL 2.0 ports (i.e., $N = 2$). To connect them, we soldered custom risers to route CXL control signals to all ports. We use three servers each with an AMD EPYC 9825 Processor, local DDR5-5600 memory, and a 100 Gbit Mellanox CX5 NIC. Each server has 64 CXL lanes, but we connect only 16 lanes (two ×8 lanes) per CPU to form a simplified Octopus island ($X = N = 2$): each pair of servers shares one MPD. All servers additionally connect with a 100 Gbit Arista 7060X series switch and run an updated Ubuntu 24.04.

RPC. We implement a simple CXL-based RPC by passing a message that contains the RPC parameters from one server to another and returning a new message containing the return value back to the initiating server. To pass a message, the sender first writes the message to a MPD, while the receiver busy polls the MPD to retrieve the message. The message can be a copy of parameters/return values or a pointer to them. To measure the RPC latency with RDMA, we use the `send` RDMA verb to send RPC parameters and to send return values between two servers. We also measure RPC latency with a user-space networking stack [45].

Simulations. Our simulations study MPD topologies with between 2 to 8 MPD ports (N), though we focus on $N = 4$. We consider server ports (X) between 4 to 16, though we focus on $X = 8$. Our main configuration is the 96-server Octopus pod (Octopus-96) consisting of six islands of 16 servers each.

Memory pooling simulations play back VM memory demand traces from Azure [108]. We collected trace data over a two-week period in December 2024 from thousands of cloud VMs. For each VM, we record its allocation and deallocation times, resource allocation, and the hosting server.

When a VM launches, it greedily allocates memory from the least utilized connected MPDs (§5.4), potentially spanning multiple MPDs, and releases memory on termination. We randomly group servers into CXL pods, replay the traces, and record the peak usage across all MPDs. This peak determines per-MPD capacity, with lower peaks indicating better pooling.

Physical layout model. We validate the physical realization of Octopus topologies within the 3-rack layout (§5) under CXL cable length constraints by modeling server and MPD placement as a satisfiability (SAT) problem. Each potential server and MPD location within the racks is modeled as a 3-D coordinate that specifies the location of the ports for each server/MPD slot.

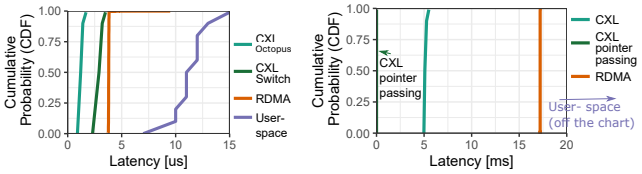
The cable length limit is modeled as a three-dimensional Manhattan distance between server and MPD ports, and is enforced through constraints on the one-to-one mapping of logical MPD and server pairs to physical locations in the racks. If a mapping that satisfies the constraints is found, then the mapping should be realizable physically, modulo cabling complexity. We implement the model in PySAT [59] and use MiniSat 2.2 to sweep cable lengths to find the shortest satisfiable cable constraint.

Cost model. We focus on CapEx. We model CXL costs in a hyperscale setting and normalize CXL costs by the number of servers. This is because although a smaller CXL pod has lower cost per pod, a hyperscaler must deploy more such pods to serve the same number of servers, resulting in similar per-server CXL cost. We assume a server cost of $30K [14, 15] and use the estimated per-MPD cost from Figure 3.

6.2 Hardware Prototype

We briefly describe our current MPD's characteristics and then evaluate our three-server island prototype. We show that we can achieve low-latency RPCs, motivate why islands are necessary, and quantify bandwidth gains for island-wide collective communication. Our prototype server each uses two ×8 CXL ports and a single 100 Gbit NIC (vs. eight ports and 400 Gbit at production scale), scaling both CXL and Ethernet by the same 4× factor so that CXL-vs.-network comparisons remain representative.

MPD hardware characteristics. We measure each ×8 CXL link to deliver 24.7 GiB/s read-only and 22.5 GiB/s write-



(a) 64 Bytes. **(b)** 100 MB.

Figure 10: Distribution of RPC round-trip latency for small and large messages.

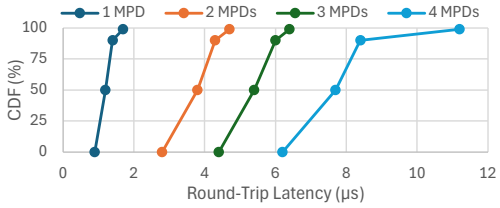


Figure 11: Round-trip RPC latency with messages being transmitted through a varying number of MPDs. Within an island, passing messages between any pair of servers only go through a single MPD.

only bandwidth. For mixed 1:1 read-write workloads, total bandwidth increases to only 28.8 GiB/s. This is lower than expected given that the CXL link is fully bidirectional, and is due to a MPD firmware issue. When both servers are active, per-server bandwidth saturates at 22.1 GiB/s.

Application slowdown. Given that MPDs have higher latency than expansion devices (267 ns vs. 233 ns, measured with our hardware), we evaluate slowdown across a broad set of cloud workloads: web (Ruby YJIT [9]), key-value stores (YCSB [35] on Redis [8] and Memcached [93]), and databases (TPC-C [10] on Silo [132], and TPC-H [11] on PostgreSQL [7]).

Figure 12 shows that about 65% of applications incur less than 10% slowdown with MPDs, consistent with our earlier estimate (§4.2). Note that latency-sensitive workloads that exceed 10% slowdown with expansion devices degrade further on MPDs; however, such workloads would not be provisioned with CXL memory in practice.

Small RPCs. We measure the round-trip latency of RPC where RPC parameters and return values are both 64 B. Figure 10a shows that the median RPC latency within an Octopus island is 1.2 µs. A CXL switch has 2.4× higher latency. RDMA has 3.2× higher latency at 3.8 µs. User-space networking has 9.5× higher latency at over 11 µs.

We validate Octopus's island concept (§5.2) by measuring RPC latency when servers have to forward messages because they do not share a MPD. For example, expander topologies with 96 servers typically require traversing 3 MPDs. We measure the round-trip RPC latency with a varying number of intermediate MPDs. Figure 11 shows that transmitting a message through two MPDs increases the median latency from 1.2 µs to 3.8 µs, comparable to RDMA.

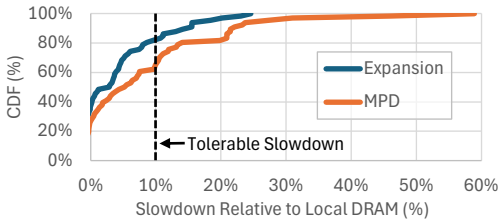


Figure 12: Cumulative distribution of application slowdown for CXL expansion devices and MPDs relative to local DRAM.

Large RPCs. We measure 100 MB parameters and 64 B return values. With Octopus, we can pass RPC parameters by value or by reference. For the latter we assume that all parameters are already in the MPD, so there is no need to serialize and copy the parameter.

Figure 10b shows the round-trip RPC latency. When passing by value, CXL achieves 5.1 ms median latency. The median latency with RDMA is 3.3× higher. When passing by reference, CXL latency matches the 64 B case, orders of magnitude lower than passing by value.

Broadcast collectives. The source server connects directly to the two other servers each via a separate MPD. Thus, the source server writes the data to the two MPDs in parallel and both destination servers read the data from the corresponding MPD in a pipeline while the source server is still writing. We measure the completion time for broadcasting 32 GB to two servers at 1.5 s, confirming the MPD serves at full bandwidth to both servers in parallel, a 2× speedup over RDMA.

All-gather collectives. The CXL links in our three-server island form a cycle, so we use the ring all-gather algorithm. With 32 GiB shards per server the all-gather completes in 2.9 s, achieving 22.1 GiB/s aggregate bidirectional bandwidth (vs. 28.8 GiB/s expected), limited by the same MPD firmware bottleneck noted above.

6.3 Scaled-Up Simulations

We use the hardware measurements to simulate scaled-up versions of Octopus with 96 servers.

6.3.1 Memory Pooling Savings

We show that Octopus achieves almost optimal memory pooling savings among MPD topologies with $X = 8$ and $N = 4$. We also compare Octopus to CXL switches which have better reachability than MPDs. However, CXL switches also have higher latency and thus can pool a smaller fraction of memory. By pooling a larger fraction of memory capacity, Octopus is able to match the pooling savings of CXL switches even under very optimistic assumptions on the CXL switch topology.

Octopus vs. other MPD topologies. To show that Octopus achieves near-optimal pooling savings, we compare it against expander topologies (Jellyfish [120]) with varying pod sizes

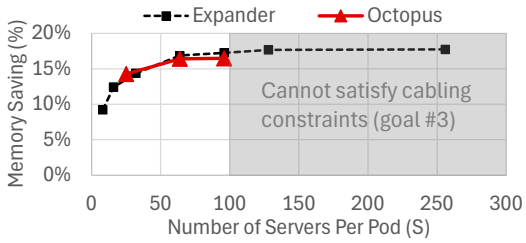


Figure 13: Average memory pooling savings with Octopus and expander topologies with a varying pod size (S).

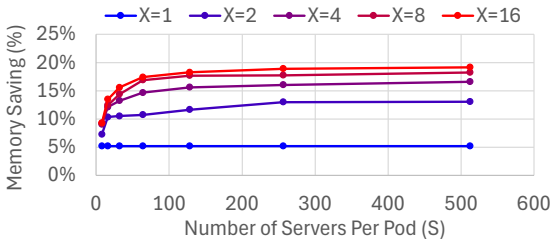


Figure 14: Average pooling savings of expander MPD topologies with a varying pod size (S) and a varying server port count (X).

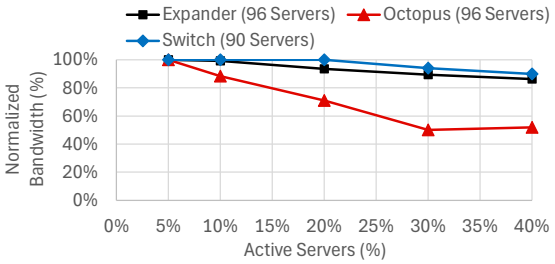


Figure 15: Average normalized bandwidth under random traffic with a varying number of active servers.

under $X = 8$ and $N = 4$. Expander topologies are known to have asymptotically optimal expansion [120, 133]. All topologies have the same MPD cost per server, since the server-to-MPD ratio is identical.

Figure 13 shows pooling savings for expander topologies up to 256 servers. Note that some of these are not physically feasible topologies, as copper cable lengths will not scale beyond two server racks, and roughly 100 severs. These large and unrealistic MPD topologies save up to 18% compared to Octopus-96 which saves 16%. The benefits flatten out around 100 servers, consistent with the peak-to-average demand ratios in Figure 5, validating Octopus's focus on 96 servers with practical copper cabling.

Octopus vs. CXL switches. CXL switches are subject to similar limitations as MPDs under fully-connected wiring assumptions. Specifically, when every CXL switch must connect to every server (as in prior work [82]), we cannot connect more than 20 servers in a pod, because we must at least use 10 ports for CXL devices and 2 ports for management [60]. When pooling across 20 servers, CXL switches can save 12% of memory, compared to 16% for Octopus-96.

We therefore simulate an optimistic CXL switch design to upper-bound its pooling savings. Specifically, we assume switches adopt a sparse topology similar to Octopus, forgo management ports, and connect up to 90 servers. Although such a sparse switch topology would be less efficient than a switch fabric that fully connects servers and devices, we optimistically model it as a global pool with 90 servers. Under this optimistic assumption, the switch topology achieves 16% memory savings, matching Octopus. This is because it pools only 35% of total DRAM and saves 46% of that pooled memory. In contrast, Octopus pools 65% of total DRAM while saving 25% of the pooled memory (§4.2).

Sensitivity analysis. We use expander topologies (Jellyfish [120]) to show how pooling savings vary with pod size (S), server port count (X), and MPD port count (N). For reference, Octopus-96 exhibits behavior similar to a expander topology with 64 servers. Figure 14 shows memory savings for expander topologies with S ranging from 2 to 512 servers and X from 1 to 16. Pooling savings generally increase with X, with diminishing returns beyond $X = 8$.

In terms of MPD port count (N), pooling savings are minimal with $N = 2$. This configuration cannot reach sufficient MPDs even with $X = 8$. Conversely, $N = 8$ is far more effective than $N = 4$, though no $N = 8$ MPDs exist today.

6.3.2 Bandwidth-Bound Communication

Octopus provides both low server-to-server communication latency and high bandwidth within each island. We show that Octopus achieves optimal performance for bandwidth-bound communication within each island.

Single active island. We simulate the optimal completion time of a uniform all-to-all communication within an island. The optimal completion time is obtained by solving a multi-commodity max flow problem [76] using linear programming. We assume that only one island is active, which allows the active island to also route traffic through inactive islands.

Our simulations show that all-to-all communication within the active island achieves optimal bandwidth, with each server fully saturating its 8 CXL links (5 intra-island, 3 inter-island), demonstrating that Octopus can leverage unused inter-island link bandwidth.

Random traffic. However, as Octopus has less inter-island bandwidth, it has lower performance under random communication across the entire pod compared to an expander topology. Figure 15 shows that with 10% active servers, Octopus has 12% lower performance. In contrast, switches achieve higher bandwidth due to their high fanout.

6.3.3 CXL Link Failures

CXL link failures are the dominant failure factor introduced by CXL memory [25]. To study how CXL link failures affect

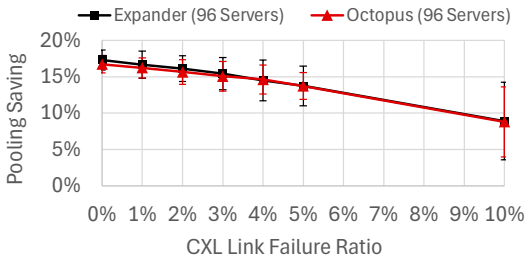


Figure 16: Average memory pooling savings under a varying CXL link failure ratio. The standard deviation is shown as the error bars.

Islands	Pod Size	CXL CapEx Cost	Cable Len.
1	25	$1252 / server	0.7 m
4	64	$1292 / server	0.9 m
6	96	$1548 / server	1.3 m

Table 4: Octopus configurations with $X = 8$ server ports and $N = 4$ MPD ports. The CXL cost includes CXL devices and required cables. The cable length is the known minimum required to realize the Octopus topology within a 3-rack configuration. Increased cable cost is the main reason for increasing costs of Octopus-96.

different CXL use cases, we simulate memory pooling savings and bandwidth-bound communication performance with a varying CXL link failure ratio. For a given failure ratio, we uniformly randomly select CXL links to fail[5].

Pooling savings. CXL link failures reduce pooling savings because affected servers can access fewer MPDs. If such a server becomes "hot," its excess memory demand can only be served by the remaining MPDs it connects to, which lowers pooling savings. We compare the memory pooling savings between Octopus and the expander topology with a varying CXL link failure ratio. Figure 16 shows that the pooling savings of both Octopus and the expander topology degrade gracefully from 17% to 14% with 5% failed CXL links.

Communication performance. We also simulate communication under random traffic with CXL link failures. With 5% link failures, performance degrades by 5%–12%, indicating Octopus's path diversity helps sustain performance.

6.4 Physical Layout Validation

To find physical placements for Octopus topologies within a 3-rack system (§5.3), we solve the physical mapping SAT problem for various cable lengths and determine the length required to realize each topology in Table 3. The cable lengths for which a solution was found within 48 hours of wall-clock time are documented in Table 4. All three Octopus topologies can be realized with cable lengths of 1.3 m or less.

[5]Disconnecting CXL devices from a running server (e.g., caused by CXL link failures) is termed *surprise removal of devices* in CXL specifications [3]. As of CXL 3.0, surprise removal may cause server faults and reboots. We therefore assume affected servers have rebooted and can access remaining MPDs via functional links.

Topology	Pod Size	CXL CapEx	Mem Saving
Expansion	\	$800 / server	\
Octopus	96	$1548 / server	16%
Switch	90	$3460 / server	16%

Table 5: Comparison of CXL device CapEx and memory pooling cost savings of Octopus and CXL switches.

Power factor	1.00	1.25	1.50	2.00
Switch CapEx ($/server)	$2969	$3589	$4613	$9487
Server CapEx	+1.7%	+3.7%	+7.1%	+22.9%

Table 6: Switch cost sensitivity under a power-law die-area model.

6.5 Cost Comparison

CXL CapEx. Table 5 summarizes the CXL device CapEx and memory pooling cost savings of the 96-server Octopus and the 90-server CXL switch topology. Although Octopus and the CXL switch topology achieve the same memory pooling savings, the switch topology's device cost is more than twice that of Octopus. Octopus's cost is 5% of server CapEx vs. 12% for switches.

Server CapEx. Combining device CapEx with memory pooling savings, Octopus reduces overall server CapEx by 3.0% compared to servers without CXL. This reduction may appear small at first, but is significant at hyperscale. Moreover, this estimate excludes additional savings from data sharing and communication use cases, which can further reduce CPU core requirements by lowering RPC overheads.

The overall server CapEx of the switch topology is 3.3% *higher* than the baseline without CXL. Due to the expensive price of switch and the many cables required, the CXL switch topology costs more than it saves in terms of DRAM CapEx.

So far, we have assumed a server without CXL as a baseline. Since CXL memory expansion is seeing increasing adoption [2], we consider a baseline where the server includes CXL memory expansion. In this case, Octopus reduces the server CapEx by 5.4%. The key is that Octopus's MPDs cost little more than existing expansion devices, whereas the switch topology still increases CapEx by 0.6%.

Sensitivity analysis. We also evaluate an alternative switch cost model in which die cost scales as a power law with die area, reflecting non-linear yield effects in semiconductor manufacturing. We vary the power factor starting from 1.0 (linear scaling). Table 6 shows that even under the optimistic linear model, server CapEx still increases by 1.7%.

7 Discussion

Limitations. In terms of communication, Octopus is optimized for low-latency and high-bandwidth communication *within* an island. Communication between servers in *different* islands may traverse two MPD hops, incurring higher latency

and lower bandwidth compared to fully-connected MPD or switched topologies. In addition, broadcast-style collectives (e.g., all-gather over shared CXL memory [17, 134, 138]) also favor full connectivity, as a sender can write once and all receivers read from the same MPD; in Octopus, such patterns require multi-hop reads or replication across MPDs.

For memory pooling, Octopus may have suboptimal savings when server demands are extremely skewed. For example, if a single server needs to allocate nearly all CXL memory, only a fully-connected (MPD or switch) topology can serve this extreme skew; Octopus's per-server MPD reachability is bounded by the number of MPDs a server connects to.

Finally, Octopus's irregular cabling may be harder to manage than a single-switch or fully-connected MPD pod [120, 133], though the comparison is less clear at sub-rack scale.

Port count changes. Octopus assumes $X = 8$ CXL ports per server and $N = 4$ ports per MPD, based on existing Intel Xeon 6 and AMD 5th Gen EPYC deployments [1, 4]. CXL 4.0 over PCIe 6.0 will double per-lane bandwidth, making narrower links ($\times 4$ or even $\times 2$) viable, so $X = 8$ independent $\times 4$ links on 32 total lanes is realistic and MPDs can scale to $N \geq 4$ within today's form factors. Octopus's topologies are specific to X and N; the split between island-specific ports (X_i) and cross-island ports ($X - X_i$) must be re-optimized for each configuration, which we leave to future work.

CXL switch topologies and future interconnects. CXL 3.0 enables non-tree switch topologies and direct switch-to-switch links [3], but each switch hop adds $\approx$220 ns of latency, making multi-level switching unattractive for latency-sensitive workloads. A promising middle ground is to combine MPD-based Octopus islands with a small switch fabric for global reachability. A transition from copper to optical CXL links could extend reach but does not eliminate the need for topology design as long as switches remain electrical.

Security. Within a single server, CXL memory isolation relies on the OS or hypervisor page tables, exactly as with local DRAM. Across servers sharing a CXL device, CXL 2.x provides no inter-server access control; pooling and sharing must also rely on hypervisor page tables. CXL 3.x Dynamic Capacity Devices (DCD) [3, 48] adds hardware-enforced, per-server access control for shared regions, enabling fine-grained multi-tenant isolation. Octopus works with both: static MPD partitioning under CXL 2.x, and on-demand secure sharing via DCD under CXL 3.x.

Memory allocation. Memory allocation for pooling in Octopus is challenging: greedily allocating from MPDs shared with neighbors may cause contention when those servers later become hot. Whether per-server demand prediction or limited memory migration can improve pooling efficiency remains an open problem.

In addition, bandwidth-sensitive workloads may require software interleaving across multiple MPDs, making pooled memory allocation more challenging when bandwidth constraints are considered.

Multi-socket servers. Octopus supports multi-socket servers in two ways: treating each server as a single node, or treating each socket as a separate node. This choice again trades off pod size and latency. Treating an entire server as one node can enable larger pods, but communication may traverse inter-socket links (e.g., UPI), increasing latency.

8 Related Work

CXL pod topologies. Octopus is, to our knowledge, the first work to compare sparsely connected CXL pod *topologies* and layouts rather than assuming one wiring pattern. Most prior work and industry decks adopt fully connected MPD pods or switch fabrics and study device properties, not topology tradeoffs across pooling and communication [42, 48, 50, 58, 82, 90, 136]. Typical designs connect every CPU to the same X MPDs so that N-ported MPDs yield pods of size N; this scales poorly in port count [42, 48, 50, 58, 82, 90, 136].

Scale-up "islands." Large shared-memory systems (HPE Superdome Flex, SGI UV) compose high-bandwidth islands with longer inter-island paths; Octopus islands mirror this: dense local sharing on a common MPD with sparser inter-island links, aligning with MDC/OpenFAM-style locality-aware programming [12, 13, 64, 65, 119].

Interconnection evolution. Point-to-point device fabrics and switches have traded places across decades: trees to high-radix, low-diameter designs (flattened butterfly, Slim/X-pander), and Clos-like CXL 3.0 fabrics [26–28, 39, 63, 67–69, 71, 77, 96]. Octopus targets near-term copper-first deployments with sparse MPD bipartite graphs with one-hop sharing while expanding pooling reach. Server-forwarded schemes like BCube [47] are less attractive under CXL latency.

9 Conclusion

Octopus challenges the assumption that effective CXL pods require fully-connected topologies with large pooling devices, showing that sparsely-connected topologies can be cost-efficient for pooling and low-latency communication. Octopus offers a practical blueprint for CXL memory pooling that balances cost, scalability, and performance.

Acknowledgments

We thank our shepherd, Alex C. Snoeren, and the anonymous reviewers for their helpful comments. We also thank Midhul Vuppalapati for his feedback and contributions to the memory pooling analysis.

References

[1] 5th Gen AMD EPYC™ Processor Architecture. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/5th-gen-amd-epyc-processor-architecture-white-paper.pdf. Accessed: 2025-08-02.

[2] Azure delivers the first cloud VM with Intel Xeon 6 and CXL memory. https://techcommunity.microsoft.com/blog/sapapplications/azure-delivers-the-first-cloud-vm-with-intel-xeon-6-and-cxl-memory---now-in-priv/4470067.

[3] Compute Express Link (CXL) 3.0 Specification. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.0-Specification.pdf.

[4] Intel® Xeon® 6 Processors. https://www.intel.com/content/www/us/en/products/details/processors/xeon/xeon6-e-cores.html. Accessed: 2025-08-02.

[5] Meeting Petabyte-scale MemorySystems Challenges with CXLMemory Pooling. https://memverge.com/wp-content/uploads/2022/10/CXL-Forum-OCP_Samsung-Xconn.pdf.

[6] Pcuo – PCIe®-over-fiber, firefly™ optical cable assembly. https://www.samtec.com/products/pcuo. Accessed 2025-09-11.

[7] PostgreSQL Database Management System. https://www.postgresql.org/.

[8] Redis, an in-memory data structure store. http://redis.io.

[9] Shopify/yjit-bench: Set of benchmarks for the YJIT CRuby JIT compiler and other Ruby implementations. https://github.com/Shopify/yjit-bench.

[10] TPC Benchmark C (TPC-C). http://www.tpc.org/tpcc/.

[11] TPC Benchmark H (TPC-H). https://www.tpc.org/tpch/.

[12] SGI UV 300 system reliability, availability and serviceability. White paper, Silicon Graphics International Corp., 2016. Accessed 2025-09-16.

[13] Hpe superdome flex server architecture and ras. Technical white paper, Hewlett Packard Enterprise, 2021. Accessed 2025-09-16.

[14] Supermicro A+ Server 2115S-NE332R, 2025. Accessed: 2025-04-02, available at https://www.thinkmate.com/system/storage-a+-server-2115s-ne332r.

[15] Thinksystem sr650 v3, 2025. Accessed: 2025-04-02, available at https://www.lenovo.com/us/en/configurator/dcg/index.html?lfo=7D76A03XNA.

[16] Advanced Micro Devices, Inc. AMD EPYC™ 9005 Processor Architecture Overview. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/user-guides/58462_amd-epyc-9005-tg-architecture-overview.pdf, Apr. 2025. Specifies up to 64 lanes of CXL 2.0 and examples of CXL-backed memory capacity.

[17] H. Ahn, S. Kim, Y. Park, W. Han, S. Ahn, T. Tran, B. Ramesh, H. Subramoni, and D. K. Panda. Mpi allgather utilizing cxl shared memory pool in multi-node computing systems. In *2024 IEEE International Conference on Big Data (BigData)*, pages 332–337. IEEE, 2024.

[18] 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. *Proceedings of the VLDB Endowment*, 17(12):3827–3840, 2024.

[19] Amazon Web Services. Adding aurora replicas to a db cluster, 2025. Aurora DB cluster supports up to 15 read replicas plus one writer.

[20] Arm Limited. Overview of Neoverse V2 Reference Design. https://developer.arm.com/documentation/102759/latest/Overview-of-Neoverse-V2-reference-design, 2024. States eight CML links support CML_SMP and CXL 2.0 for connections to accelerators/memory devices.

[21] I. Astera Labs. Aries pcie®/cxl® smart cable modules™. https://www.asteralabs.com/products/aries-smart-cable-modules/, 2024. Product Brief.

[22] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Morgan & Claypool, 3rd edition, 2018. Discusses how hyperscale server component procurement and pricing are typically confidential and governed by non-disclosure agreements.

[23] A. Baumstark, M. Paradies, K.-U. Sattler, S. Kläbe, and S. Baumann. So far and yet so near-accelerating distributed joins with cxl. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*, pages 1–9, 2024.

[24] D. S. Berger, D. Ernst, H. Li, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, L. Hsu, I. Agarwal, M. D. Hill,

et al. Design tradeoffs in cxl-based memory pools for public cloud platforms. *IEEE Micro*, 43(2):30–38, 2023.

[25] D. S. Berger, K. Kumar, M. Vuppalapati, C. Douglas, J. Sathre, I. Robinson, P. Tandon, and M. D. Hill. CXL in Cloud Practice: Practical Lessons for Incrementally Scaling Deployment. *Computer*, 2026. Special Issue on Compute Express Link (CXL).

[26] M. Besta and T. Hoefler. Slim fly: A cost effective low-diameter network topology. In *SC'14: proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 348–359. IEEE, 2014.

[27] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1–es, 2006.

[28] N. Blach, M. Besta, D. De Sensi, J. Domke, H. Harake, S. Li, P. Iff, M. Konieczny, K. Lakhotia, A. Kubicek, et al. A {High-Performance} design, implementation, deployment, and evaluation of the slim fly network. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1025–1044, 2024.

[29] R. C. Bose. On balanced incomplete block designs. *Annals of Human Genetics*, 9:353–399, 1939.

[30] P. Chauhan, C. Petersen, B. Morris, and J. Glisse. Hyperscale tiered memory expander specification for compute express link. Available at https://www.opencompute.org/documents/hyperscale-tiered-memory-expander-specification-for-compute-express-link-cxl-1-pdf, 2023. Open Compute Project, Revision 1, Effective October 27, 2023.

[31] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 37–48, San Jose, CA, June 2013. USENIX Association.

[32] L. Clarke, I. Glendinning, and R. Hempel. The mpi message passing interface standard. In *Programming Environments for Massively Parallel Distributed Systems: Working Conference of the IFIP WG 10.3, April 25–29, 1994*, pages 213–218. Springer, 1994.

[33] D. Cohen, S. Cohen, D. Naor, D. Waddington, and M. Hershcovitch. Dictionary based cache line compression. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, pages 8–14, 2024.

[34] Compute Express Link Consortium. Compute Express Link (CXL) Specification, Revision 2.0, nov 2020. Accessed: 2025-03-11.

[35] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[36] I. Corporation. Intel Flat Memory Mode: Expanding System Memory with CXL. https://www.intel.com/content/www/us/en/products/docs/xeon-6-product-brief.html, 2024.

[37] M. Cowan, S. Maleki, M. Musuvathi, O. Saarikivi, and Y. Xiong. Mscclang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 502–514, 2023.

[38] R. Cypher and R. A. Armistead III. Configuring networks using balanced incomplete block designs, Sept. 29 2015. US Patent 9,148,371.

[39] W. J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE transactions on Computers*, 39(6):775–785, 1990.

[40] D. Das Sharma, R. Blankenship, and D. Berger. An introduction to the compute express link (cxl) interconnect. *ACM Computing Surveys*, 56(11):1–37, 2024.

[41] P. Duraisamy, W. Xu, S. Hare, R. Rajwar, D. Culler, Z. Xu, J. Fan, C. Kennelly, B. McCloskey, D. Mijailovic, B. Morris, C. Mukherjee, J. Ren, G. Thelen, P. Turner, C. Villavieja, P. Ranganathan, and A. Vahdat. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *ASPLOS*, 2023.

[42] M. El-Batal. Seagate Composable Memory Appliance (CMA) Architecture. https://www.youtube.com/watch?v=KCgE0WejXl0, June 2024.

[43] M. Ewais and P. Chow. Disaggregated memory in the datacenter: A survey. *IEEE Access*, 11:20688–20712, 2023.

[44] R. A. Fisher and F. Yates. The construction of balanced incomplete block designs. *Annals of Human Genetics*, 9:30–43, 1938.

[45] J. Fried, G. I. Chaudhry, E. Saurez, E. Choukse, Í. Goiri, S. Elnikety, R. Fonseca, and A. Belay. Making kernel bypass practical for the cloud with junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 55–73, 2024.

[46] Google Cloud. Create a production instance (cloud bigtable sample), 2025. Sample shows minimum of 3 serve nodes for production instances.

[47] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *SIGCOMM Comput. Commun. Rev.*, 39(4):63–74, aug 2009.

[48] M. Ha, J. Ryu, J. Choi, K. Ko, S. Kim, S. Hyun, D. Moon, B. Koh, H. Lee, M. Kim, et al. Dynamic capacity service for improving cxl pooled memory efficiency. *IEEE Micro*, 43(2):39–47, 2023.

[49] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, and T. Moscibroda. Protean:VM allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861, 2020.

[50] D. Hawkins and M. Bromage. Design Considerations for CXL Device Hardware Coherency (HDM-DB). https://www.youtube.com/watch?v=2ktM7dPcmqI, 2024.

[51] Hewlett Packard Enterprise. Hpe 256 gb (1×256 gb) quad rank ×4 ddr5-6400 ec8 registered 3ds smart memory kit. Technical report, Hewlett Packard Enterprise, 2025. Product data sheet for 3DS DDR5-6400 256 GB Smart Memory module.

[52] S. Hoory, N. Linial, and A. Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

[53] T. Huang, Y. Liang, S. Yu, and K. Chen. Txcocket: an innovative solution for efficient cross-node data transmission enabled by cxl-based shared memory. *CCF Transactions on High Performance Computing*, January 2025. Regular Paper, Published: 22 January 2025.

[54] Y. Huang, H. Chen, N. Ni, Y. Sun, V. Chidambaram, D. Tang, and E. Witchel. Tigon: a distributed database for a cxl pod. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[55] Y. Huang, N. Ni, V. Chidambaram, E. Witchel, and D. Tang. Pasha: An efficient, scalable database architecture for cxl pods. 2025. Published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license.

[56] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 145–158. USENIX Association, 2010.

[57] D. Huye, Y. Shkuro, and R. R. Sambasivan. Lifting the veil on meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC '23)*, 2023.

[58] R. Hyatt. The quest for bandwidth and capacity: Memory edition, 2023. https://www.hpcuserforum.com/wp-content/uploads/2023/09/Ronen-Hyatt_UnifabriX_The-Quest-for-Bandwidth-and-Capacity-Memory-Edition_Sept-2023-HPC-UF.pdf.

[59] A. Ignatiev, Z. L. Tan, and C. Karamanos. Towards universally accessible SAT technology. In *SAT*, pages 4:1–4:11, 2024.

[60] J. Jiang. CXL Switch for Scalable & Composable Memory Pooling/Sharing. FMS presentation available at https://www.xconn-tech.com/products, 2024.

[61] A. Kalia, M. Kaminsky, and D. G. Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[62] M. Kalodrich. New supermicro x14 systems, 2024. Accessed: 2024-12-18.

[63] J. Kao. CXL 2.0 Switch for a Composable Memory System. https://computeexpresslink.org/wp-content/uploads/2024/09/Xconn_CXL-2.0-Switch-for-a-Composable-Memory-System_FMS-2024_FINAL.pdf, October 2024.

[64] K. Keeton. Memory-driven computing. In *Proceedings of FAST '17: 15th USENIX Conference on File and Storage Technologies*, Santa Clara, CA, Feb. 2017. USENIX Association.

[65] K. Keeton, S. Singhal, and M. A. Raymond. The openfam api: A programming model for disaggregated persistent memory. In S. Pophale, N. S. V. Imam, F. Aderholdt, and M. G. Venkata, editors, *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*, volume 11283 of *Lecture Notes in Computer Science*, pages 70–89. Springer, 2019.

[66] P. Kennedy. Lenovo has a cxl memory monster with 128x 128gb ddr5 dimms, 2024. Accessed: 2024-12-18.

[67] J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 126–137, 2007.

[68] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Computer Architecture News*, 36(3):77–88, 2008.

[69] J. Kim, W. J. Dally, B. Towles, and A. K. Gupta. Microarchitecture of a high radix router. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 420–431. IEEE, 2005.

[70] A. Labs. Leo cxl smart memory controllers. Available at https://www.asteralabs.com/products/leo-cxl-smart-memory-controllers/, December 2023. Product Brief.

[71] K. Lakhotia, M. Besta, L. Monroe, K. Isham, P. Iff, T. Hoefler, and F. Petrini. Polarfly: a cost-effective and flexible low-diameter topology. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[72] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[73] D. Lee and O. Mutlu. Dram scaling challenges and solutions. *IEEE Micro*, 42(2):14–25, 2022.

[74] D. Lee, T. Willhalm, M. Ahn, S. Mutalik Desai, D. Booss, N. Singh, D. Ritter, J. Kim, and O. Rebholz. Elastic use of far memory for in-memory database management systems. In *Proceedings of the 19th International Workshop on Data Management on New Hardware*, pages 35–43, 2023.

[75] T. Lee, S. K. Monga, C. Min, and Y. I. Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 17–34, 2023.

[76] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, Nov. 1999.

[77] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

[78] Lenovo. Lenovo thinksystem sr860 v3 server, 2024. Accessed: 2024-12-18.

[79] Lenovo. Implementing cxl memory on linux on thinksystem v4 servers. White paper, 2025.

[80] Lenovo. Thinksystem memory summary: 256 gb truddr5 3ds registered dimm (ddr5-6400). Technical report, Lenovo Press, July 2025. Memory summary listing 256 GB 3DS TruDDR5 RDIMM for ThinkSystem servers.

[81] P. Levis, K. Lin, and A. Tai. A case against cxl memory pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 18–24, 2023.

[82] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *ASPLOS*, 2023.

[83] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[84] 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 2025 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)*, 2025.

[85] 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*, page 1203–1217. Association for Computing Machinery, New York, NY, USA, 2025.

[86] J. Liu, H. Hadian, H. Xu, and H. Li. Tiered memory management beyond hotness. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[87] J. Liu, H. Xu, D. S. Berger, M. K. Aguilera, and H. Li. Performance predictability in heterogeneous memory. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2026.

[88] N. Luehr. Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl, 2016. Accessed 1/1/2025.

[89] T. Ma, Z. Liu, C. Wei, J. Huang, Y. Zhuo, H. Li, N. Zhang, Y. Guan, D. Niu, M. Zhang, et al. HydraRPC:RPC in the CXL era. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 387–395, 2024.

[90] G. Mackey. You don't know 'jack': Cxl fabric orchestration and management. Available at https://files.futurememorystorage.com/proceedings/2024/20240806_CXLT-102-1_Mackey.pdf, 2024. Presented by Jackrabbit Labs.

[91] S. Mahar, E. Hajyjasini, S. Lee, Z. Zhang, M. Shen, and S. Swanson. Telepathic datacenters: Fast rpcs using shared cxl memory, 2024.

[92] I. Marvell Technology. Structera a 2504 memory-expansion controller. Available at https://www.marvell.com/content/dam/marvell/en/public-collateral/assets/marvell-structera-a-2504-near-memory-accelerator-product-brief.pdf, 2024. Product Brief, P/N MV-SLA25041-A0-HF350AA-C000.

[93] Memcached. Memcached. https://memcached.org/, 2024.

[94] MemVerge. Memverge unveils world's first cxl-based multi-server shared memory at isc. Press release, May 2023. International Supercomputing Conference (ISC), Hamburg, Germany.

[95] MongoDB, Inc. Replica set members, 2025. Up to 50 members, 7 voting; common 3–7 voting members.

[96] D. Moore and D. D. Sharma. CXL 3.0: Enabling composable systems with expanded fabric capabilities. https://computeexpresslink.org/wp-content/uploads/2023/12/CXL_3.0-Webinar_FINAL.pdf, October 2022.

[97] B. Munger, K. Wilcox, J. Sniderman, C. Tung, B. Johnson, R. Schreiber, C. Henrion, K. Gillespie, T. Burd, H. Fair, D. Johnson, J. White, S. McLelland, S. Bakke, J. Olson, R. McCracken, M. Pickett, A. Horiuchi, H. Nguyen, and T. H. Jackson. "zen 4": The amd 5nm 5.7ghz x86-64 microprocessor core. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 38–39, 2023.

[98] O. Mutlu. Memory scaling: A systems architecture perspective. *International Memory Workshop (IMW)*, 2013.

[99] A. Name. *Investigating States and Logged Contents in CXL Memory*. PhD thesis, University Name, 2025.

[100] O. N. s. OCP Server Workgroup. *OCP NIC 3.0 Design Specification Version 1.5.0 - Release*, September 2024. Available at https://drive.google.com/file/d/1z7MU6Lpu8xQ19rprxHrL1bjaxtx106iV/view.

[101] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[102] Oracle. Mysql innodb cluster—introduction, 2025. Production clusters require at least three instances.

[103] Y.-A. Pignolet, S. Schmid, and G. Tredan. Load-optimal local fast rerouting for resilient networks. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 345–356, 2017.

[104] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi. Cerebros: Evading the rpc tax in datacenters. In *Proceedings of MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, Virtual Event, Greece, 2021. ACM.

[105] A. Rack. Gnrd8-2l2t preliminary ceb specifications, 2024. Accessed: 2024-12-18.

[106] D. Raghavan, P. Levis, M. Zaharia, and I. Zhang. Breakfast of champions: Towards zero-copy serialization with nic scatter-gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. ACM, 2021.

[107] Redis Ltd. Scale with redis cluster—recommended topologies, 2025. Recommends 6-node production clusters (3 masters + 3 replicas).

[108] B. Reidys, P. Zardoshti, I. n. Goiri, C. Irvene, D. S. Berger, H. Ma, K. Arya, E. Cortez, T. Stark, E. Bak, M. Iyigun, S. Novakovic, L. Hsu, K. Trueba, A. Pan, C. Bansal, S. Rajmohan, J. Huang, and R. Bianchini. Coach: Exploiting temporal patterns for all-resource oversubscription in cloud platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 164–181, New York, NY, USA, 2025. Association for Computing Machinery.

[109] I. Samtec. Successful pcie interconnect guidelines, 2021. Available at ttps://blog.samtec.com/wp-content/uploads/2021/04/04_15_2021_successful_PCIe_interconnect_guidelines.pdf. Accessed March 11, 2025.

[110] A. Sanaee, V. Jabrayilov, I. Marinos, A. Kalia, D. Saxena, P. Goyal, K. Kaffes, and G. Antichi. Fast userspace networking for the rest of us. *arXiv preprint arXiv:2502.09281*, 2025.

[111] SAP SE. Sap hana tailored data center integration—frequently asked questions, 2018. FAQ repeating the 16 worker-node limit for TDI.

[112] SAP SE. Understanding the maximum number of nodes in sap hana scale-out (kba 3557729), 2024. States HANA TDI scale-out is limited to 16 worker nodes.

[113] S. Scargall. Cxl server buyer's guide: A complete list of ga platforms (updated 2025). Blog post, June 2025.

[114] Seagate Technology Team Member(s). Composable memory appliance (cma) base specification. Specification, Open Compute Project, Nov. 2024. Seagate contribution to OCP CMS; the CFM spec references this Base Spec.

[115] 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 ACM Symposium on Operating Systems Principles (SOSP '23)*, pages 1–17, Koblenz, Germany, 2023. ACM.

[116] S. Sha, C. Li, Y. Luo, X. Wang, and Z. Wang. vTMM: Tiered Memory Management for Virtual Machines. In *EuroSys*, 2023.

[117] A. Shah, V. Chidambaram, M. Cowan, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, O. Saarikivi, and R. Singh. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 593–612, 2023.

[118] S. S. Shrikhande. A survey of balanced incomplete block designs. *Journal of Combinatorial Theory, Series A*, 15(3):271–291, 1973.

[119] S. Singhal, C. R. Crasta, M. Abdulla, F. Barmawer, D. Emberson, R. Ahobala, et al. Openfam: A library for programming disaggregated memory. In S. Poole, O. Hernandez, M. Baker, and T. Curtis, editors, *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Exascale and Smart Networks*, volume 13159 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2022.

[120] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, pages 225–238, San Jose, CA, apr 2012. USENIX Association.

[121] SK hynix. Fms 2025: Sk hynix presents future of ai and storage through innovative tech showcase. https://news.skhynix.com/future-of-memory-and-storage-2025/, Aug. 2025. Press Release.

[122] A. B. Smith and J. L. Walrand. Integrated circuit yield and cost analysis: A critical area approach. In *Proceedings of the IEEE International Conference on VLSI Design*, pages 115–120. IEEE, 2005.

[123] P. Solutions. Cxl memory expansion servers from penguin solutions. Product page / datasheet, 2025.

[124] A. Sriraman and A. Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. ACM, 2020.

[125] J. Suetterlein, J. Manzano, and A. Marquez. Synchronization for cxl based memory. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '24, page 178–185, New York, NY, USA, 2024. Association for Computing Machinery.

[126] Super Micro Computer, Inc. Storage superserver ssg-121e-ne3x12r datasheet, 2025. Datasheet PDF.

[127] K. Systems. Kr2280v3 platform intel amd, 2024. Accessed: 2024-12-18.

[128] D. Technologies. Cxl-supported dell poweredge platforms. Dell InfoHub → "CXL Memory Expanded and Optimized for Dell PowerEdge Servers", 2025.

[129] S. Telian. Pcie gen5 signal integrity implementation. In *DesignCon 2023*, 2023. Available at https://siguys.com/wp-content/uploads/2023/02/DesCon23_PAPER_Track7_PCIeGen5SignalIntegrityImplemenation_Telian.pdf. Accessed March 11, 2025.

[130] Y. Tian. Project gismo: Global i/o-free shared memory objects. Presented at Flash Memory Summit (FMS) 2023, https://memverge.com/wp-content/uploads/MemVerge-Gismo-FMS2023.pdf, 2023.

[131] 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*, pages 1–14, 2020.

[132] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[133] A. Valadarsky, G. Shahaf, M. Dinitz, and M. Schapira. Xpander: Towards optimal-performance datacenters. In *Proceedings of the 12th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '16)*, pages 205–219, Heidelberg, Germany, dec 2016. ACM.

[134] S. Vanecek, M. Turner, M. Gajbe, M. Wolf, and M. Schulz. Modeling the potential of message-free

communication via cxl.mem. In *Proceedings of the Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region*, SCA/HPCAsia '26, page 258–270, New York, NY, USA, 2026. Association for Computing Machinery.

[135] 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*, pages 79–94, 2024.

[136] M. Wagh and R. Sodke. Compute express link 2.0 specification: Memory pooling. Available at https://computeexpresslink.org/wp-content/uploads/2023/12/CXL-2.0-Memory-Pooling.pdf, 2021. Presented at the CXL Consortium.

[137] 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)*, pages 11–20. IEEE, 2022.

[138] X. Wang, B. Ma, J. Kim, B. Koh, H. Kim, and D. Li. cmpi: Using cxl memory sharing for mpi one-sided and two-sided inter-node communications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '25, page 2216–2232, New York, NY, USA, 2025. Association for Computing Machinery.

[139] M. Weisgut, D. Ritter, P. Tözün, L. Benson, and T. Rabl. Cxl memory performance for in-memory data processing. *Proceedings of the VLDB Endowment*, 18(9):3119–3133, 2025.

[140] XConn. XC50256 CXL 2.0 Switch Chip. https://www.xconn-tech.com/products, 2024.

[141] 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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 19–35, 2024.

[142] C. Xu, Y. Liu, Z. Li, Q. Chen, H. Zhao, D. Zeng, Q. Peng, X. Wu, H. Zhao, S. Fu, and M. Guo. FaaSMem: Improving Memory Efficiency of Serverless Computing with Memory Pool Architecture. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 331–348, 2024.

[143] X. Yang, Q. Hu, J. Li, F. Li, Y. Zhu, Y. Zhou, Q. Lin, J. Dai, Y. Kong, J. Zhang, et al. Beluga: A CXL-Based Memory Architecture for Scalable and Efficient LLM KVCache Management. *arXiv preprint arXiv:2511.20172*, 2025.

[144] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[145] 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)*, pages 3796–3809. IEEE, 2024.

[146] M. Zhang, T. Ma, J. Hua, Z. Liu, K. Chen, N. Ding, F. Du, J. Jiang, T. Ma, and Y. Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 658–674, New York, NY, USA, 2023. Association for Computing Machinery.

[147] Y. Zhong, D. S. Berger, C. Waldspurger, R. Wee, I. Agarwal, R. Agarwal, F. Hady, K. Kumar, M. D. Hill, M. Chowdhury, et al. Managing memory tiers with CXL in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 37–56, 2024.

[148] Y. Zhong, D. S. Berger, P. Zardoshti, E. Saurez, J. Nelson, A. Psistakis, J. Fried, and A. Cidon. Beware, PCIe Switches! CXL Pools Are Out to Get You. In *Proceedings of the 20th Workshop on Hot Topics in Operating Systems (HotOS XX)*, Banff, Alberta, Canada, May 2025. ACM SIGOPS.

[149] Y. Zhong, E. Saurez, A. Psistakis, D. S. Berger, J. Nelson, J. Fried, P. Zardoshti, D. R. K. Ports, and A. Cidon. Oasis: Pooling PCIe Devices Over CXL to Boost Utilization. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP '25)*, 2025.

[150] Z. Zhu, N. Ni, Y. Huang, Y. Sun, Z. Jia, N. S. Kim, and E. Witchel. Lupin: Tolerating partial failures in a cxl pod. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems*, pages 41–50, 2024.

A Appendix

A.1 Expansion and Pooling Benefits

The expansion e_k of an MPD topology directly yields a lower bound on peak MPD usage within a pod:

Theorem A.1. *Let D_k denote the maximum aggregate memory demand of any subset of k servers in a pod. Assume a worst-case scenario in which, for each k, a subset of k servers $U \subseteq S$ that attains D_k also connects to only e_k distinct MPDs. Then the peak memory usage across all MPDs, denoted $L^\star$, satisfies*

$$L^\star \geq \max_{1 \leq k \leq |S|} \frac{D_k}{e_k},$$

where $|S|$ is the number of servers in the pod.

Proof. Fix k and let U be a subset of k servers that attains D_k and, by the assumption, connects to e_k distinct MPDs. All demand from servers in U must be served by these e_k MPDs, so at least one of them must carry load at least D_k/e_k. Therefore $L^\star \geq D_k/e_k$ for any k. Taking the maximum over k completes the proof. □