Understanding and Profiling the Accelerator Chiplet Network Using PingPoint

Junyeol Ryu
University of Wisconsin–Madison
Madison, WI, USA

Ming Liu
University of Wisconsin–Madison
Madison, WI, USA

Matthew D. Sinclair
University of Wisconsin–Madison
Madison, WI, USA

Abstract

Emerging chiplet-based accelerators introduce a new class of intra-host networks—the Accelerator Chiplet Network (ACN)—that links compute chiplets, IO chiplets, and memory modules and increasingly governs application performance. Yet ACN behavior remains largely opaque: existing tools overlook on-package communication and instead attribute overheads to compute or memory subsystems, while ACN-induced latency, bandwidth heterogeneity, and congestion are hard to observe due to proprietary microarchitectures, tight coupling with the execution pipeline, and complex mappings between application activity and hardware.

To overcome this challenge, we build an ACN characterization framework that enables fine-grained, topology-aware probing of paths and links. We then use it to uncover fundamental ACN performance properties on multi-chiplet GPUs. Guided by these insights, we design **`PingPoint`**, a lightweight utility for ACN-native profiling. *Our key insight is that modeling the ACN as a logical, hose-based graph with queueing abstractions, combined with in-situ software probing, makes systematic dissection of the otherwise opaque ACN possible.* It injects latency and bandwidth probes while co-executing target kernels, captures cycle-level link- and path-granular distributions, and applies differential attribution to localize congestion to individual ACN links. Across diverse workloads and hardware, it exposes hidden bottlenecks, guides kernel placement and traffic shaping, quantifies the performance impact of ACN contention, and enables practical optimization with marginal overhead.

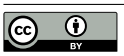
CCS Concepts

• **Computer systems organization → Interconnection architectures**; • **Networks → Network performance analysis**; **Network on chip**.

Keywords

Accelerator Chiplet Network, Performance Profiling

ACM Reference Format:
Junyeol Ryu, Ming Liu, and Matthew D. Sinclair. 2026. Understanding and Profiling the Accelerator Chiplet Network Using PingPoint. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3789240.3829110



SIGCOMM '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/26/08
https://doi.org/10.1145/3789240.3829110

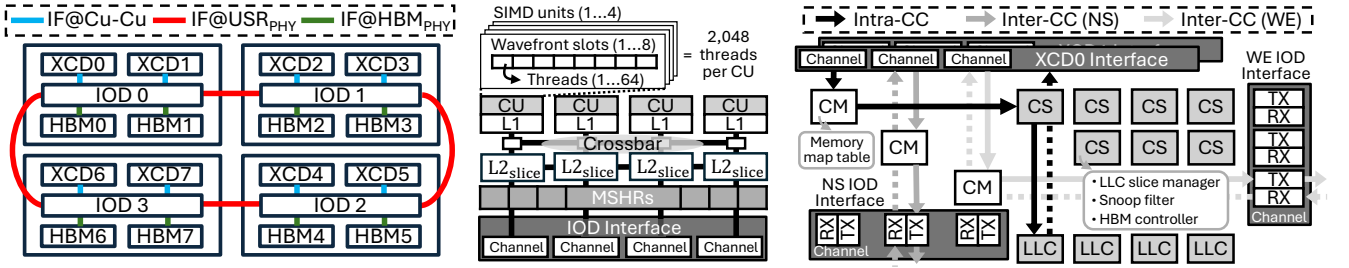
1 Introduction

Chiplets have rapidly become the dominant building block for modern hardware accelerators [11, 73, 74, 87, 160]. Chiplet-based designs decompose a monolithic die into modular, function-specific components that are integrated through advanced packaging technologies [39, 72, 101, 106], improving yield and cost efficiency, enabling heterogeneous integration and silicon reuse, accelerating time-to-market via independent development/verification, and reducing data-movement overheads. Accordingly, a growing number of production accelerators are chiplet-based [7, 8, 75, 88, 126, 164].

As a result, a new class of intra-host networks that interconnects compute chiplets, IO chiplets, and memory modules has emerged; we term this substrate the **Accelerator Chiplet Network** (**ACN**). An ACN governs chiplet-to-chiplet communication for memory requests and responses, cache coherence traffic, data movement, and IO signaling, and sits alongside off-package interconnects [12, 47, 96, 128, 139]. For example, in AMD's MI300X [7], the ACN connects eight compute chiplets (XCDs), four IO chiplets (IODs), and eight HBM stacks within a single SoC. Unlike traditional on-chip networks designed for monolithic dies, ACNs span heterogeneous hardware domains and route traffic across multiple hops using stateful routers with built-in traffic management. They expose a layered stack—physical signaling (L1), reliable framing and hop-by-hop flow control (L2), and transaction-level routing over flits (L3)—and comprise several distinct link types, including COM-IO, IO-IO, and IO-MEM. These properties make ACNs more complex than conventional NoCs and elevate them to a first-class performance determinant inside modern accelerator SoCs.

However, despite being the linchpin, the ACN remains poorly understood in the networking community, making workload profiling, diagnosis, and optimization difficult. Our MI300X/MI350X evaluations show that many kernels are constrained by ACN links rather than wavefront parallelism or HBM bandwidth. For example, in MI300X, north-south and west-east IO-IO links peak at 2.4 TB/s, versus 6.2 TB/s for COM-IO. Yet ACN behavior is tightly coupled with schedulers, prefetchers, and page migration, obscuring root-cause analysis: even when developers follow vendor guidelines [15, 133], performance often falls short due to head-of-line (HoL) blocking, uneven bandwidth partitioning, and transient congestion [78–81, 146]. This opacity is exacerbated by limited interconnect observability [17, 19, 46] and by the scale of modern GPU parallelism, with thousands of in-flight requests generating irregular traffic matrices at sub-μs timescales, where conventional telemetry and traffic engineering techniques [2, 30, 67, 91, 92, 174] are largely ineffective.

Unfortunately, existing tools and frameworks often overlook the ACN, primarily attributing performance overheads to compute units or memory subsystems. For example, prior GPU microbenchmark studies [82, 90, 144, 145, 169, 171], including recent



(a) SoC Architecture and Topology. (b) XCD Architecture. (c) IOD Architecture and Data Flow.

Figure 1: The AMD MI300X accelerator. IF=Infinity Fabric. CM(CS)=Coherent Master(Slave). NS=North-South. WE=West-East.

work on MI300X [68], focus on CU scaling and HBM latency and bandwidth, leaving the chiplet interconnect unexplored. Similarly, AMD's ROCProfiler [20, 21] and NVIDIA's Nsight [127, 131] infer interconnect behavior only indirectly through interacting-module counters or coarse stall metrics, providing only a partial view.

To overcome these challenges, we build a programmable ACN characterization framework (§3) that enables fine-grained control over traffic sources and destinations through XCD pinning, HBM stack localization, and topology-aware traffic generation. Using this framework, we systematically probe and characterize MI300X's latency and bandwidth across directional paths and link types. First, we find that the ACN dominates the end-to-end memory access latency under load and exposes heterogeneous bandwidth domains (§3.3), invalidating the assumption of a flat on-package interconnect. Second, COM-IO links spread traffic across multiple channels and behave like centralized FIFOs (§3.4), introducing bounded but non-negligible HoL blocking under mixed workloads. Third, IO-IO links combine deep buffering with deterministic routing (§3.5) and sustain high throughput when flows are placed on disjoint paths, but incur μs-scale queueing delays when requests concentrate on a single link. Finally, IO-MEM links route requests by address over shallowly buffered channels and apply traffic-oblivious backpressure (§3.6), allowing congestion at one HBM stack to ripple across the entire ACN.

Based on these insights, we propose **`PingPoint`** (§4), a developer utility to uncover ACN communication behaviors. `PingPoint` operates as a lightweight monitoring agent that co-locates with target applications and reports ACN-inherent performance characteristics, bottlenecks, and idiosyncrasies. Our key ideas are to (a) treat the ACN as an abstract logical graph with the hose model [4, 28, 54], (b) represent each compute adapter, IO chiplet, and memory-side adapter with tailored queueing abstractions, and (c) perform in-situ software probing (ping) to infer internal network conditions, inspired by PingMesh [63]. `PingPoint` injects latency- and bandwidth-focused probe traffic across all ⟨XCD, IOD, HBM⟩ tuples while co-executing target kernels through fused GPU kernels, ensuring precise temporal alignment and spatial isolation. Probe results provide cycle-level distributions at link- and path-level granularity. A differentiation-based analyzer then attributes observed delays and throughput deviations to specific ACN links, producing a hierarchical congestion map that explains how application traffic stresses the on-package network.

We evaluate `PingPoint`'s effectiveness in profiling, diagnosing, and guiding optimization with minimal overhead (§5). Our results include: kernel-level decomposition (§5.1), input-driven slowdown variation (§5.2), cross-workload interference diagnosis (§5.3), overhead/tuning tradeoffs (§5.4), application optimization (§5.5), and cross-GPU portability (§5.6). `PingPoint` is available at https://github.com/netlab-wisconsin/pingpoint.

		Specification
Accelerator Overall	Clock/Power	2100 MHz/750W (TDP)
	Microarch.	CDNA3
	Chiplet Config.	8 XCDs (5nm) and 4 IODs (6nm)
	Throughput	1307 TFLOPS (FP16), 2614 TFLOPS (FP8)
Compute Chiplet (XCD)	CU #	38 (2 for yield management)
	L1D	32KB per-CU, 128B CL, 1 CH, per-CH 64W4S
	L2	4MB per-XCD, 128B CL, 16 CHs, per-CH 16W128S
IO Chiplet (IOD)	XCD-IOD	3D, Copper-to-Copper, 4.3 TB/s per CC
	IOD-IOD	2.5D, USR-PHY, 1.2/1.5 NS/WE TB/s per CC
	LLC	64MB per-IOD, 64B CL, 32 CHs, per-CH 16W2048S
	IOD-HBM	2.5D, HBM-PHY, 1.3 TB/s per CC
HBM Module	Size/Bus/Rate	192GB HBM3, 8192 bit width, 5.2 Gbps
	Paging	2MB page under 4KB interleaved

Table 1: Hardware specification of the AMD MI300X. CL=Cache line. CH=Channel. xWyS=x-way y sets. CC=Chiplet Cluster.

2 Background and Motivation

2.1 Chiplet-based Accelerator

Chiplets are a semiconductor technology that is increasingly used to build modern processors [11, 73, 74, 87, 160]. They divide traditional monolithic chips into modular, function-specific building blocks, and allow different IC (e.g., cores, IO, and memory) compositions via advanced packaging technologies [39, 72, 101, 106]. Chiplets (a) improve manufacturing yield and cost efficiency by reducing the probability of defects within a single module; (b) enable flexible silicon customization via heterogeneous integration of different node processes; (c) expedite time-to-market by allowing independent development, testing, and verification; (d) enhance power efficiency and performance by reducing the physical distance of data movement—a common bottleneck. Recently many chiplet-based accelerators have emerged, including AMD's MI300X/MI350X [7, 8], NVIDIA's B100/B200 [88, 126], Intel's Gaudi 3 [75], and Tenstorrent's Blackhole [164].

We use the AMD MI300X as an exemplar to describe how an accelerator SoC (System-on-Chip) is constructed from chiplets. Table 1 shows its hardware specification. As depicted in Figure 1a, it comprises 8 compute chiplets (Accelerator Complex Dies; XCDs), 4 IO chiplets (Input/Output Dies; IODs), and 8 HBM stacks (192GB total memory). An XCD (Figure 1b) encloses 38 Compute Units (CUs) and a 4MB shared L2 cache, interconnected via a crossbar [40]. A CU has numerous features, including a 32KB private L1D cache and 4 SIMD execution units. Each unit has an 8-slot wavefront

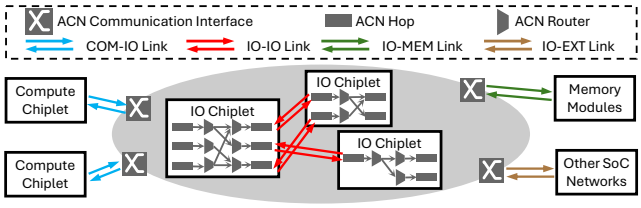


Figure 2: A conceptual overview of ACN architecture.

engine, where each scheduling slot runs a 64-thread group (called a *wavefront* in AMD GPUs, analogous to a CUDA warp), yielding up to 2,048 ($4 \times 8 \times 64$) threads per CU. Appendix §A summarizes the key terms used in this paper.

An IOD (Figure 1c) connects the XCDs and HBMs via its Infinity Fabric (IF) [13], which is built atop different physical layers. The XCD-IOD link employs 3D hybrid bonding Copper-to-Copper (Cu-Cu) [102, 118] and achieves 4.3 TB/s. IOD-IOD and IOD-HBM ones use Ultra-Short-Reach (USR) PHY [52, 157] and HBM PHY [83] atop 2.5D silicon interposers, delivering 1.2-1.5 TB/s and 1.3 TB/s. An IOD contains cache coherence managers (containing coherent master and slave), LLC slices (64MB per-IOD), and upstreaming/downstreaming/peering communication interfaces (INTF) for talking to XCDs, HBMs, and neighboring IODs. An INTF has 16 parallel channels, each of which is provisioned with TX/RX ports for bidirectional traffic. The 8 HBM stacks run at 1.3 GHz, use 2MB pages, and serve data at an 8192-bit width. Each group of physically proximate components—an IOD and its attached 2 XCDs and 2 HBM stacks (outer box in Figure 1a)—forms a tile; we call it a *Chiplet Cluster (CC)* in this paper.

2.2 Accelerator Chiplet Networking (ACN)

The ACN connects compute chiplets, IO chiplets, and memory modules. It defines the chiplet-chiplet (not intra-chiplet) communication behaviors and characteristics of memory requests/responses, cache-coherent messages, data signaling, and IO traffic. More generally, the ACN is part of an accelerator SoC and its interface integrates with other on/off-chip interconnects, like PCIe [139], GMI [12], xGMI [96], NVLink [128], and UALink [47]. An ACN differs from an intra-(compute or IO) chiplet NoC (Network-on-Chip) in that it covers multiple heterogeneous hardware domains with a broad range of data traffic types and requirements, yielding a more complex networking substrate.

An ACN (Figure 2) encompasses (a) chiplet communication interfaces where data originates or drains; (b) intermediate hops that relay/forward traffic to another segment; and (c) stateful routers enclosing ingress/egress traffic management, congestion control, and queueing with certain scheduling disciplines, organized in a graph topology (e.g., Mesh [70], Torus [122], Cube [61], or Dragonfly [94]). The ACN has three hierarchical layers. The physical layer (L1), containing silicon wires and specialized circuits, handles (de)modulation, time synchronization, signal integrity check, and transmission model control. The link layer (L2) ensures reliable and ordered data delivery via frame-level error control and hop-by-hop flow control [98–100]. The transaction layer (L3) routes data from the source to the destination following the channel semantics at the link-stipulated granularity (e.g., FLIT).

The ACN's data path has four link types: Compute-IO chiplet (*COM-IO link*), IO-IO chiplet (*IO-IO link*), IO-Memory (*IO-MEM link*), and IO-other interconnect (*IO-EXT link*). For example, in Figure 1c, an XCD's L2 misses are delivered to the IOD's Coherent Master (CM) via a COM-IO link. The CM contains a memory mapping table and sends the request to a Coherent Slave (CS) based on the requested address [41]. The CS (working as a caching home agent [97]), enclosing 2MB LLC slices, a snoop filter [6], and a memory controller, then forwards the request to neighboring IODs [57, 157] or HBM stacks through IO-IO or IO-MEM links depending on where the data is located.

2.3 Problem, Challenges, and Prior Solutions

Problem. The networking community lacks a sufficient and systematic understanding of the ACN's communication capabilities and limitations, rendering workload profiling, performance diagnostics, and system optimization challenging on chiplet-based accelerators. First, since the ACN bridges the compute, memory, and IO subsystems, it often acts as a data movement choke point. Thus, it cannot be overlooked. As we show in §3 and §5.6, application throughput on MI300X and MI350X is often restricted by ACN links rather than by wavefront parallelism or HBM memory bandwidth. For example, on MI300X, IO-IO links sustain 2.4 TB/s, while COM-IO links provide 6.2 TB/s (§3.3). Second, the tightly coupled ACN imbues many architectural intricacies, complicating root-cause exploration when encountering performance anomalies. However, even after applying recommended best practices [15], developers still struggle to achieve peak performance. We find that many culprits are inherent to the ACN (§5), including: head-of-line blocking slows down synchronization, uneven bandwidth partitioning reduces a wavefront's data feed rate, and ephemeral contention due to uncoordinated scheduling causes significant queue buildup. Third, developers strive to optimize software stacks given emerging needs, like performance, multi-tenancy, reliability, and energy efficiency. The ACN is mostly viewed as a transparent piece and not treated separately today. For example, LithOS's fine-grained GPU resource management via kernel atomization [48] would see diminished returns due to ACN contention.

Challenges. However, there are three non-trivial challenges. First, the ACN lacks clear hardware boundaries. There are also minimal tracking and logging modules: MI300X exposes 259 performance counters [17], only 28 of which are interconnect-related, including sampled IOD stall, idle, and busy events (Table 4 in Appendix §B). Second, ACN characteristics correspond closely to compute chiplet execution and memory access patterns [154–156], making it difficult to isolate them. For example, MI300X's work-conserving wavefront scheduler [158, 159], opportunistic page migration [161, 167], and temporal/spatial-locality driven prefetching [31, 170] can alter traffic profiles, making ACN drill-down analysis challenging. Third, dissecting the ACN requires an accurate and timely top-down mapping between co-located workloads and the underlying hardware. However, GPUs have massive parallelism and thousands of in-flight memory transactions at a sub-microsecond granularity. This creates large, sometimes irregular traffic and intertwined, opaque data paths which conventional network telemetry and traffic management techniques barely tackle [2, 30, 67, 91, 92, 174].

Algorithm 1 XCD Pinning.

```
1: kernel fn prelude_xcd (target_xcd, kernel_entrypoint, *args):
2:   asm volatile ("s_getreg_b32 %0,hwreg(HW_REG_XCC_ID)":"=r"(xid));
3:   if xid == target_xcd:
4:     kernel_entrypoint(*args);
```

Algorithm 2 HBM Stack Localizer.

```
1: kernel fn hbm_probe (data, n_chunks, cycles):
2:   asm volatile ("s_getreg_b32 %0,hwreg(HW_REG_XCC_ID)":"=r"(xid));
3:   for i = 0 to n_chunks - 1:
4:     idx = i * blockDim.x + threadIdx.x; // Calculate data index per-thread
5:     synchronize(); // Cross-XCD synchronize
6:     start = clock(); load(data[idx]); end = clock(); // Measure latency
7:     cycles[xid * n_chunks + i] = end - start; // Store result
8: fn hbm_analyzer (home_hbm, n_chunks, cycles):
9:   for i = 0 to n_chunks - 1:
10:    for xcd = 0 to NUM_XCDS - 1:
11:      if cycles[xcd * n_chunks + i] < min_cycles:
12:        min_xcd = xcd; min_cycles = cycles[xcd * n_chunks + i];
13:    home_hbm[i] = (min_xcd->cc)->hbm; // HBM stack within CC of min_xcd
```

Inadequacies of Prior Solutions. Existing hardware characterization frameworks, system utilities, and development toolchains largely overlook the impact of ACN, instead attributing performance overheads to other SoC components. For example, AMD's CDNA3 microbenchmark suite [68] analyzes CUs, CU scaling behavior, and the latency and bandwidth of on-chip memories and off-chip HBM, ignoring the interconnect that connects these components. Likewise, AMD's ROCProfiler [20, 21] decomposes the GPU into functional blocks and applies a Top-Down [173] or VTune-style [76] approach. However, due to Performance Monitoring Unit limitations, interconnect behavior is only indirectly inferred through counters of interacting modules or coarse-grained stall statistics which aggregate cycles stalled by factors often related to interconnect congestion, yielding only partial and indirect information. NVIDIA's Nsight [127, 131] has similar limitations. It provides standalone benchmarking for SMs and memory subsystems but relies on kernel replay techniques [55, 66, 135] to track interconnect data flows rather than directly exposing communication fabric properties (e.g., bandwidth allocation and congestion dynamics). Appendix §B discusses these limitations in detail.

Our work fills this gap through ACN-native characterization and profiling. We first systematically dissect the MI300X's ACN at link and path granularity, uncovering its performance capabilities and bottlenecks and motivating a hose-based graph abstraction (§3). Building on these insights, we develop `PingPoint`, a lightweight probing-based profiler for online, kernel-level ACN visibility (§4); and then demonstrate its effectiveness in identifying, localizing, and attributing ACN-induced performance effects (§5).

3 Understanding ACN

3.1 Experimental Methodology

Hardware Testbeds. Our experiments run on MI300X GPUs (Table 1) in AMD's AI & HPC Cluster, which consists of an AMD EPYC 9684X processor, 2.3TB DDR5, 8× MI300X, and 2TB storage [9]. The Genoa-X CPU has 16 sockets, each with 6 cores, and runs at 2.55GHz [33]. There are a 64KB per-core L1, a 1MB per-core L2, and a 1152MB shared L3. We disable frequency scaling (Precision Boost) to provide more consistent results across runs.

Software Benchmarks. We develop a synthetic benchmarking framework based on prior work [36, 68, 82, 144, 145, 171], and tailor it to chiplet-based GPUs. Specifically, we created two basic kernel primitives: (a) `lat`, performing pointer-chasing over an array using a single thread; (b) `bw`, a reduction of multiple arrays across all threads. The former accesses array elements in random order such that each cacheline is hit exactly once, while the latter touches data in an interleaved manner to maximize memory-level parallelism. Moreover, `bw` stresses different components of the accelerator SoC and measures the maximum achieved bandwidth. We further enhance the framework with software-controlled computation and data allocation to enable flexible traffic generation (§3.2).

Algorithm 3 Traffic Generator.

```
1: let target_xcd, target_hbm;
2: hipMalloc(data, n_chunks * chunk_size);
3: hipMalloc(cycles, NUM_XCDS * n_chunks * sizeof(uint32_t));
4: hbm_probe<<<NUM_XCDS,blockDim,0,0>>>(data, n_chunks, cycles);
5: hbm_analyzer(home_hbm, n_chunks, cycles);
6: args->data = filter(data, i => home_hbm[i] == target_hbm);
7: mask_cu(cu_mask, enabled_cu_ids); // Appendix §C details mask_cu
8: hipExtStreamCreateWithCUMask(stream, cu_mask_size, cu_mask);
9: prelude_xcd<<<gridDim,blockDim,0,stream>>>(target_xcd, benchmark_kernel, args);
```

Performance Metrics. We primarily report average/tail latency and per-CU/XCD bandwidth. Our framework provides fine-grained instrumentation for latency breakdown. We also collect ROCProfiler [20, 21] architectural counters.

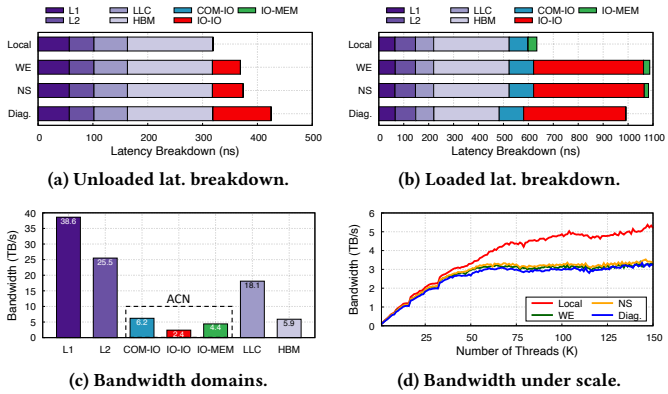
3.2 Benchmarking Programmability

Chiplet accelerators (like MI300X and MI350X) expose limited software control over thread scheduling and data placement, constraining our characterization. Thus, we develop two techniques: (a) *XCD Pinning*, which uses a prelude kernel [43, 48] and, optionally, CU affinity mask [22, 134] to spawn and run threads on specified XCDs, or even specified CUs; (b) *HBM Stack Localizer*, which identifies the mapping between data and HBM stacks via latency differentiation [84] among CCs.

XCD Pinning. Algorithm 1 shows how we send work to a specific XCD. We use `prelude_xcd`, which first identifies the XCD on which a thread is scheduled (`xid`) via inline assembly (L2), and then invokes the actual benchmark kernel (`kernel_entrypoint`), either `lat` or `bw`, only if `xid` matches the `target_xcd` (L3–4). We further reverse-engineer the CU mask for AMD's chiplet GPUs (Appendix §C) to pin a `lat` thread to a specific CU or to modulate the number of active CUs for `bw` to stress test at different levels of bandwidth.

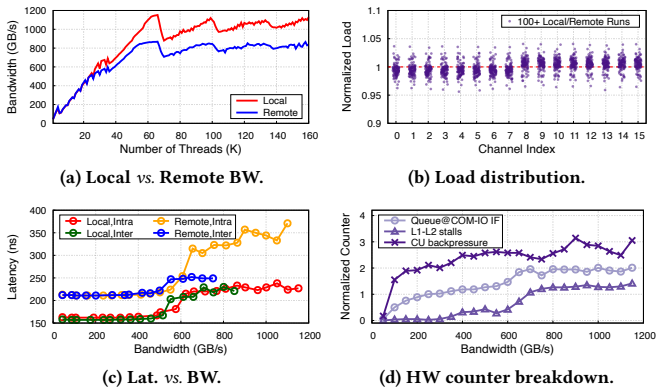
HBM Stack Localizer. Algorithm 2 shows how we localize data to an HBM stack. The MI300X uses 2MB-sized pages, interleaved into 4KB chunks across HBM stacks. `hbm_probe` launches 256 probing threads per XCD to load these 4KB chunks in lockstep (16B per thread × 256 threads = 4KB). The kernel iterates through all target chunks (L3), calculating the global data index for each thread (L4). To ensure accurate timing, it synchronizes all XCDs (L5), measures per-XCD load latency (L6), and stores the results for analysis (L7). `hbm_analyzer` determines the physical location (`home_hbm`) of each chunk. It iterates over the profiled data for each chunk (L9), comparing the load latency observed by each XCD (L10–11) to identify the `min_xcd` with the lowest latency (L12). Based on chiplet distance characteristics [84] and intra-CC spatial affinity, it determines the HBM stack within the CC of the `min_xcd` as the physical location of the chunk (L13).

Traffic Generation. Algorithm 3 integrates these primitives to launch traffic between a source XCD (`target_xcd`) and destination HBM (`target_hbm`). First, we allocate `n_chunks` data chunks (L2),



(a) Unloaded lat. breakdown. (b) Loaded lat. breakdown. (c) Bandwidth domains. (d) Bandwidth under scale.

Figure 3: (a)/(b) present the latency breakdown of a local, WE, NS, and diagonal memory access under unloaded/loaded cases; (c) reports the bandwidth capacity of different path segments; (d) shows the achieved bandwidth when increasing the number of concurrent benchmarking threads.



(a) Local *vs.* Remote BW. (b) Load distribution. (c) Lat. *vs.* BW. (d) HW counter breakdown.

Figure 4: (a) compares the achieved local/remote access bandwidth when increasing the number of threads; (b) depicts the load distribution over 16 channels; (c) presents the latency of local-/remote-CC traffic when increasing the intra-/inter-CC load; (d) shows the hardware counter breakdown when increasing the load of the COM-IO link (y-axis is log scale).

identify each one's position via `hbm_probe` and `hbm_analyzer` (L4–5), and store only the chunks located on `target_hbm` in `args->data` (L6). Next, we construct a CU mask for `enabled_cu_ids` using `mask_cu` (Algorithm 6 in Appendix §C) and create a HIP stream with `hipExtStreamCreateWithCUMask` that is restricted to the specified CUs/SEs with (L7–8). Last, we issue the `prelude_xcd` wrapper on this stream (L9), where the `benchmark_kernel` runs on threads scheduled on the `target_xcd` and accesses `args->data`, whose chunks all reside on the `target_hbm`.

Based on this, we create four traffic patterns: (a) `local` (XCD→local CC); (b) `WE` (XCD→x-axis neighbor CC); (c) `NS` (XCD→y-axis neighbor CC); (d) `diagonal` (XCD→diagonal CC). Taking `lat` with `local` as an example, a thread pinned at XCD0 performs pointer chasing, accessing HBM stacks 0 and 1, thereby siloing ACN traffic within CC0 and excluding inter-CC effects. Taking `bw` with `WE` as another example, every two XCDs on CC 0, 1, 2, 3 reduce over CC 1, 2, 3, 0, ensuring all traffic traverses IO-IO links at least once.

Together, these patterns exercise all ACN paths by varying source XCDs and targeting either HBM stacks or IOD LLCs. We select LLC or HBM destinations by sizing the working set W relative to the per-XCD L2 and per-IOD LLC capacities (Table 1); when L2<W<LLC, a warmup sweep populates the LLC.[1] By comparing paths that differ by a single segment, we isolate the contribution of individual ACN links: e.g., paths from XCD0 to IOD0's LLC, IOD1's LLC, and HBM2 attached to IOD1 traverse progressively one additional COM-IO, IO-IO, and IO-MEM segment, allowing incremental decomposition of per-segment latency. The following §3.3–§3.6 focus on MI300X, but the methodology generalizes to chiplet GPUs like MI350X (§5.6).

3.3 Performance Basis

Latency Breakdown. We first examine end-to-end data access from a CU to an HBM stack from different directions and quantify latency variation across paths. To do this, we gradually increase the working set, launch the `lat` kernel to the designated data location, and measure elapsed time via instrumentation. When the system is unloaded (Figure 3a), the HBM access dominates, consuming 156 ns and contributing 42.3% of the total latency (372 ns) on average across the four cases, followed by the LLC (61 ns), L1 (57 ns), ACN (54 ns), and L2 (45 ns). Routing in the ACN is not free, accounting for 14.4%, where one IO-IO hop takes ≈50 ns. When the accelerator is loaded with `bw` kernels running in the background (Figure 3b), the ACN dominates, consuming 436 ns and contributing 43.1% of the total (949 ns), followed by HBM, L2, LLC, and L1. The IO-IO link is the most costly segment, consuming 324 ns on average due to its deep buffering and associated queueing delay (§3.5).

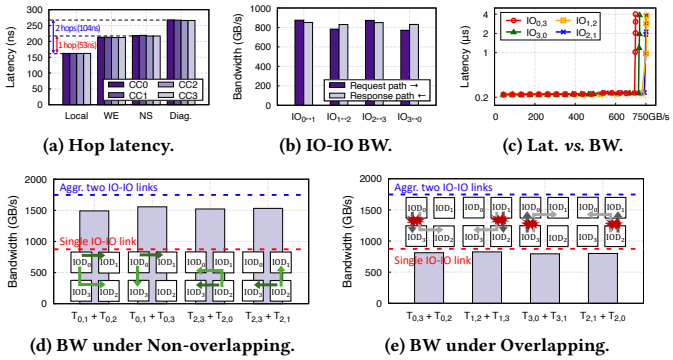
Bandwidth Domain. Figure 3c shows that the ACN introduces several implicit bandwidth domains, whose capacity is lower than the memory hierarchy. Specifically, the COM-IO, IO-IO, and IO-MEM links sustain 6.2 TB/s, 2.4 TB/s, and 4.4 TB/s, whereas L1, L2, LLC, and HBM can deliver 38.6 TB/s, 25.5 TB/s, 18.1 TB/s, and 5.9 TB/s. The issue becomes even worse when concurrent streams compete for the same link, or traffic is partitioned over channels (discussed in §3.5). We validate this by examining the bandwidth scalability of four directional accesses (Figure 3d). The local case, bypassing the IO-IO link, achieves 5.3 TB/s, while WE, NS, and diagonal ones are throttled at 3.3 TB/s.

Implication #1: The ACN contributes a substantial and, under load, often dominant share of end-to-end memory-access latency, especially on multi-hop paths. Its bandwidth is highly non-uniform: different link types form distinct capacity domains, creating "thin pipes" and heterogeneous bandwidth-delay products (BDPs) along a path. Thus, modeling the ACN as a flat, opaque substrate hides first-order performance bottlenecks. Effective diagnosis requires exposing the ACN as a structured, topology-aware network whose links and paths can be explicitly modeled, measured, and attributed.

3.4 COM-IO: Partial-Isolated Multi-Channel Link

Multi-channel with Spraying. A COM-IO link comprises 16 data channels, shared by the local- and remote-CC traffic. As shown in Figure 4a, the COM-IO link from a single XCD sustains 1.1 TB/s (70.6 GB/s per channel) for local access when the XCD is fully activated, i.e., each CU running 2K benchmarking threads, 2K×38=76K in total. For remote traffic, the link only achieves 0.8 TB/s (52.1 GB/s per channel) because each request, traversing the ACN, takes a

[1] On MI300X/MI350X, each per-IOD LLC caches only its two/four attached HBM stacks.



(a) Hop latency. (b) IO-IO BW. (c) Lat. *vs.* BW.

(d) BW under Non-overlapping. (e) BW under Overlapping.

Figure 5: (a) presents the latency of intra-/inter-CC communication; (b) compares the maximum achieved bandwidth of IO-IO links; (c) shows the latency when increasing the link load; (d)/(e) report the aggregate bandwidth of concurrent flows without and with contention on an IO-IO link.

longer time to complete (§3.3). The sawtooth behavior while increasing the number of threads is due to batched wavefront/workgroup scheduling [32, 138, 148]—since the GPU does not have enough resources to launch all wavefronts at the same time, as each group of wavefronts completes, another group is launched. We then examine load distribution using `TCC` (L2) and `TCC_EA0` (L2-Fabric) hardware counters, and find that the COM-IO link performs traffic-agnostic, uniform spraying across its 16 channels (Figure 4b).

Controlled Head-of-Line (HoL). Since each channel is oblivious to the request type, HoL blocking starts to occur when the link becomes 43.7–66.8% loaded, as shown in Figure 4c. Local-CC latencies under intra-/inter-CC load increase by up to 76 ns, and remote-CC latency under inter-CC load by up to 41 ns. In contrast, remote-CC latency under intra-CC load increases much more (up to 160 ns), suggesting that local-CC requests can HoL-block remote-CC requests on shared COM-IO channels. Figure 4d further shows that queueing mainly stems from three reasons: CU backpressure, L1/L2 stalls due to a lack of free Miss Status Holding Registers (MSHRs), and the COM-IO interface (see Appendix §D for details). These hardware-level mechanisms bound the HoL-induced overheads.

Implication #2: The COM-IO link comprises multiple parallel channels, and traffic-agnostic spraying makes them collectively behave as a centralized FIFO. Because each channel carries both local and remote requests, heterogeneous flows can experience HoL blocking in shared queues. However, we find that interface-level arbitration, L1/L2 caching, and backpressure propagation bound the impact of HoL blocking. Hence, accurate ACN analysis requires explicit modeling of channel-level multiplexing and flow control, rather than assuming independent or idealized links.

3.5 IO-IO: Buffered Link with Deterministic Routing

Link Homogeneity. IO-IO links connect IODs, yielding a scale-out accelerator architecture. We denote the unidirectional IO-IO link from IOD_x to IOD_y as $IO_{x,y}$. Thus, the full set of IO-IO links in a 2×2 IOD mesh of an MI300X is: $IO_{0,1}$, $IO_{0,3}$, $IO_{1,2}$, $IO_{1,0}$, $IO_{2,3}$, $IO_{2,1}$, $IO_{3,0}$, $IO_{3,2}$. We measure the latency and bandwidth of intra-/inter-CC memory accesses across different directions. As shown in Figures 5a/5b, all IO-IO links in an MI300X are equivalent, taking 53 ns for traffic forwarding and sustaining 820.0 GB/s.

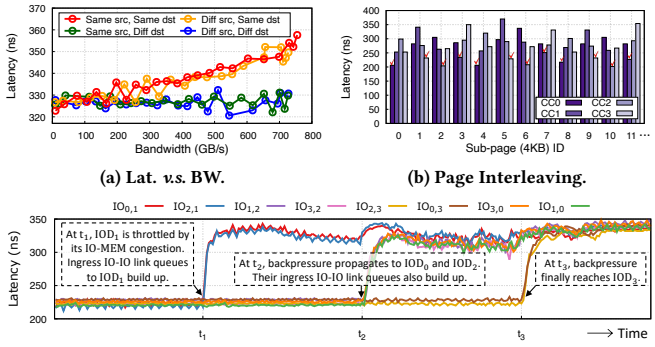
Deep Buffering. An IOD employs a multi-channel switching architecture with a large SRAM buffer for each IO-IO link at the egress to accommodate the temporary burst under traffic incast. This causes the queue build-up and increases the memory access latency dramatically. As shown in Figure 5c, when an IO-IO link is heavily loaded and close to the bandwidth limit, the average request latencies over $IO_{0,3}$, $IO_{3,0}$, $IO_{1,2}$, and $IO_{2,1}$ rise to 4005 ns, 3903 ns, 3818 ns, and 3787 ns, which are 17.1×, 16.7×, 16.6×, and 16.0× increases over each link's unloaded latency, respectively.

Deterministic Stateless Routing. Existing chiplet-based accelerators [7, 8, 75, 88, 126, 164] employ a location-based oblivious routing scheme to streamline the design and reduce the engineering cost. The MI300X uses the canonical YX routing algorithm [27, 95, 137] for traffic forwarding between IODs. This indicates: (a) the request and response take asymmetric routes; (b) physically adjacent IODs always choose the shortest path without detouring. Taking Figure 1a as an example, when XCD_0 writes to HBM_7, data flows through $IO_{0,3}$ and $IO_{3,0}$ regardless of the link loading status or whether there are other longer paths (like $IO_{0,1} \rightarrow IO_{1,2} \rightarrow IO_{2,3}$) offering more available bandwidth. Given such deterministic routing, one could determine traffic congestion based on flow (memory access) characteristics (like source XCD, destination HBM, size, and in-flight request depth). Thus, as shown in Figures 5d/5e, when there is no overlap among concurrent flows, accessing the same target HBM can use all IO-IO links, yielding a bandwidth of 1.5 TB/s. However, when overlapping happens, the aggregated bandwidth sustains 791.1–820.0 GB/s, throttled by the shared link, where bandwidth is partitioned among competing flows opportunistically.

Implication #3: IO-IO links exhibit largely uniform performance characteristics, independent of their physical placement in the topology or the specific IODs they interconnect. To absorb traffic bursts, each IO-IO link is provisioned with deep buffering at the IOD, which stabilizes throughput but induces substantial queueing latency under sustained load. Further, an IOD employs deterministic, stateless routing, making traffic distribution across IO-IO links highly predictable. When flows use disjoint IO-IO links, bandwidth scales with the number of links. In contrast, when they contend on a shared link, bandwidth is dynamically partitioned, and queueing delay dominates. These observations suggest that ACN performance hinges on explicit, topology-aware traffic placement rather than reactive congestion handling.

3.6 IO-MEM: Addr-Directed Traffic-Oblivious Link

Address-Directed Multi-Channel. The IO-MEM link bridges an IOD and an HBM stack, carrying traffic between the LLC and memory controller. Since an LLC is usually divided into several segments, an IO-MEM link has multiple channels, each of which is dedicated to one partition. For example, an IOD of the MI300X has a 64MB LLC (256MB in total) and 32 channels, with the IO-MEM link bandwidth capped at 737.5 GB/s. Unlike the COM-IO link, which employs spraying (§3.4), the IO-MEM link restricts each memory request to a single channel, determined by its destination address. Thus, for flows targeting the same HBM partition, as shown in Figure 6a, channel sharing causes up to a 37 ns latency increase when the link is over-subscribed, indicating that a shallow buffer is provisioned compared with the IO-IO link (§3.5). Flows issuing to different partitions would see little interference. Because of this, to



(a) Lat. *v.s.* BW. (b) Page Interleaving.

(c) Backpressure Propagation.

Figure 6: (a) shows the latency when increasing the load of an IO-MEM link; (b) presents the average latency of four XCDs in different CCs when accessing 16 subpages; (c) depicts how a congested IO-MEM link propagates contention across IODs. $IO_{x,y}$=IO-IO link from IOD_x to IOD_y.

fully drive the memory bandwidth under data locality, the MI300X performs page interleaving at the sub-page (4KB) granularity, where the default page size is 2MB. We also observe that the page interleaving order obeys a pre-defined clockwise direction as follows: $H_0 \rightarrow H_2 \rightarrow H_4 \rightarrow H_6 \rightarrow H_1 \rightarrow H_3 \rightarrow H_5 \rightarrow H_7$, where H_x denotes HBM x. We identify this behavior using the HBM Stack Localizer (§3.2), and Figure 6b shows it at CC granularity.

Backpressure Storm. The IO-MEM link is traffic-oblivious. When it becomes congested (queue builds up at the channel), it would throttle the transmission of the connected IOD blindly instead of figuring out the culprit flows, e.g., IOD_1 in Figure 6c. As such, all traffic targeting the choked chiplet is affected, where the backpressure propagates along the IO-IO links (e.g., $IO_{0,1}$ and $IO_{2,1}$). Consequently, the victim region is expanded to IOD_0 and IOD_2 and eventually to all IODs. This phenomenon resembles the PFC pause frame deadlock issue in RDMA networks [62]. As shown in Figure 6c, we observe that (a) a completely disjointed flow with no link sharing between the source XCD and destination HBM experiences up to a 139 ns latency increase; (b) the performance anomaly would take tens to hundreds of microseconds to resolve.

Implication #4: The IO-MEM link comprises shallow-buffered channels that route memory requests deterministically based on destination address. This traffic-oblivious design is effective under moderate load but becomes problematic under congestion: when a single flow overwhelms an IO-MEM link, backpressure propagates from the attached IOD to all others, indiscriminately throttling unrelated traffic. Such congestion persists for a non-negligible amount of time before dissipating, introducing significant and system-wide performance perturbations. These findings highlight the need for flow-aware congestion control or isolation mechanisms in the ACN, beyond coarse-grained, reactive backpressure.

4 PingPoint: an ACN Profiler

Based on gathered insights (§3), we build the PingPoint utility to profile and analyze the ACN. Our design goals are:

- **Informative.** PingPoint should provide adequate running statistics of the ACN from multiple angles, capturing traffic loads across links/paths, localizing queueing-induced stalls, and identifying congestion-affected victim regions.
- **End-to-end.** PingPoint should take a holistic view and place the ACN in the context of applications running atop an accelerator. It draws the interactions between the ACN and other architectural components for the given workload.
- **Lightweight.** The ACN carries TB/s-scale traffic and delivers massive transactions for thousands of threads. It is pivotal to ensure PingPoint incurs minimal overheads with a marginal impact on the profiled applications.

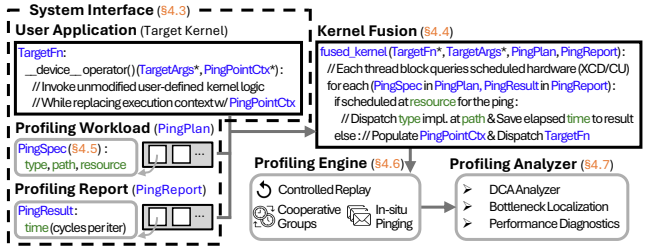


Figure 7: PingPoint System Overview.

4.1 Key Idea and Overview

Our key idea is to model the ACN as an abstract logical graph, perform in-situ software pinging within the accelerator SoC, and apply conventional traffic analysis techniques to infer network conditions. This design is inspired by PingMesh [63, 110], an established latency-SLA diagnostic system for data center networks. First, PingPoint tailors the hose model [4, 28, 54] to the ACN and augments each communication entity and link with a specialized device abstraction and queueing model, enabling software-level observability without hardware instrumentation. Second, PingPoint injects designated probe traffic for each ⟨XCD, IOD, HBM⟩ tuple. By collecting end-to-end latency distribution snapshots at both link and path granularities, PingPoint analyzes the load and contention of each modeled ACN component, revealing otherwise opaque architectural behaviors. Third, PingPoint correlates application execution, ACN traffic flows, and memory accesses, and applies delay-based inference [24, 58] to identify performance bottlenecks.

As shown in Figure 7, PingPoint comprises five components. (1) An input parser in the system interface (§4.3) extracts the application description, user-specified inputs, and profiling objectives. (2) A kernel fusion module (§4.4) composes application kernels with profiling kernels under a unified execution interface, enforcing performance isolation while preserving application semantics. (3) A ping constructor (§4.5) synthesizes structured probing plans and associated measurement contexts. (4) A profiling engine (§4.6) executes online measurements through controlled replay and bulk synchronization to ensure high-fidelity observations. (5) A profiling analyzer (§4.7) applies performance-differentiation techniques to attribute observed effects to ACN communication behaviors.

4.2 ACN Hardware Model

The sheer volume and granularity of memory traffic between CUs and HBM stacks make per-flow tracing within the ACN prohibitively expensive. Inspired by pioneering networking abstractions, we apply the hose model [4, 28, 54] and view each ACN link as a

hose link that captures aggregate, rather than per-flow, behavior. Each hose link is parameterized by three quantities: (a) the maximum sustainable aggregate outgoing rate, (b) the maximum sustainable aggregate incoming rate, and (c) a performance envelope conditioned solely on the total traffic observed at the hose interface. This aligns with our findings (§3), where each link is governed by aggregate utilization rather than by per-packet forwarding.

Based on the measured behavior of COM-IO, IO-IO, and IO-MEM links, we model the controlling hardware entities—compute adapter, IO chiplet, and memory controller—using three queueing abstractions that reflect the dominant sources of latency and contention uncovered (§3). We further adopt Kendall's notation [86] for discussion: in a queue `A/S/c/K/N/D`, `A` denotes the arrival process, `S` the service-time distribution, `c` the number of servers, `K` the queue capacity, `N` the number of sources, and `D` the service discipline.

- *Central-FCFS.* We model the **COM adapter** as a centralized first-come-first-served queue ($G/G/1/Q/X/FCFS$) that aggregates requests from multiple producers ($X \geq 1$). This reflects our observation that traffic-agnostic channel spraying and shared per-channel buffering induce bounded HoL blocking at COM-IO links (**Implication #2**), while hardware flow control limits pathological queue growth.
- *Split-FCFS.* We model the **IO Chiplet** as a collection of coupled FCFS servers ($G/G/X/Q/1/FCFS$) arranged along the execution pipeline. This captures both the deep buffering provisioned at IO-IO links and the deterministic, stateless routing logic that makes contention predictable and topology dependent (**Implication #3**). Prior work, such as S3-FIFO [172], has explored similar queueing structures.
- *Central-PS.* We model the **Memory-Side Adapter** as a centralized processor-sharing queue ($G/G/1/Q/X/PS$), reflecting token-ring arbitration [49] and the coarse-grained backpressure mechanisms that throttle multiple IO chiplets when a single IO-MEM link saturates (**Implication #4**). The processor-sharing discipline approximates the opportunistic bandwidth sharing we observe under congestion.

4.3 System Interface

`PingPoint` exposes an interface for developers to (a) onboard target kernels for profiling, (b) specify structured probing workloads over the ACN, and (c) define the profiling report.

User Application. We encapsulate the user's target kernel in a C++ function object, termed `TargetFn` (Figure 7). Its template defines an `operator()` with a `__device__` signature, indicating GPU execution, and two arguments: (a) `TargetArgs`, which carries the kernel's original parameters, and (b) `PingPointCtx`, which conveys runtime context to launch the kernel. The target kernel body is encoded in this operator, and CUDA/HIP built-ins (e.g., `gridDim`, `blockDim`, and `blockIdx`) are replaced with the corresponding fields in `PingPointCtx`. This indirection allows `PingPoint` to invoke *unmodified application logic* through the profiling engine (§4.6) while retaining control over the execution context.

Profiling Workload. The probing workload is defined by a list of pings, called `PingPlan`. A ping acts as an atomic probing unit, executing a specific microbenchmark across a defined hardware datapath, specified by `PingSpec` (§4.5).

Profiling Report. The profiling report, called `PingReport`, mirrors the `PingPlan` structure and maintains a strict one-to-one correspondence between each profiling specification (`PingSpec`) and its output record (`PingResult`). For each ping, `PingResult` allocates a buffer that records raw elapsed clock cycles across profiling iterations. These measurements serve as proxies for ACN congestion induced by the target kernel. Aggregated across the full plan, the results enable reconstructing the logical ACN graph and quantifying link-level contention under application load. Upon completion, `PingReport` is transferred to host memory and becomes the primary dataset consumed by the result analyzer (§4.7).

4.4 Kernel Fusion

`PingPoint` fuses the target kernel (`TargetFn`) with profiling logic into a single GPU kernel, termed `fused_kernel` (Figure 7), enabling fine-grained temporal alignment between application-induced traffic and probing traffic. `fused_kernel`'s launch configuration is determined dynamically to satisfy the combined resource demands of the target computation and profiling pings, including registers, shared memory, and probe-reserved thread blocks. The kernel is parameterized by `TargetFn`, and its corresponding `TargetArgs`, `PingPlan`, and `PingReport`. The fused kernel is launched using the Cooperative Groups API [18, 129], which provides grid-wide synchronization primitives (e.g., `sync`) spanning all thread blocks, independent of their physical placement on CUs or XCDs. `PingPoint` uses these global barriers to precisely align the start and end of each ping with the execution window of the target kernel, ensuring that probe traffic reflects the instantaneous ACN state induced by the application.

Within `fused_kernel`, each thread block queries its hardware locality—specifically, its assigned XCD and CU—and determines its role at runtime. A subset of blocks, specified by each `PingSpec` and placed on the designated source XCD, executes the ping microbenchmark. The remaining blocks execute the application logic by invoking `TargetFn` with a runtime context dynamically populated in `PingPointCtx`, including role assignment and hardware identifiers. This allows unmodified application code to run while exposing placement information to the profiling framework.

To minimize measurement noise, `PingPoint` spatially isolates probing and application execution by assigning them to disjoint CUs. This design eliminates interference from thread-block scheduling and resource contention, ensuring that measured latency and bandwidth variations reflect ACN behavior rather than on-XCD compute effects.

4.5 `PingSpec` Structure

`PingSpec` describes the parameters of a ping, including the microbenchmark type, e.g., latency or bandwidth (§3.1); the probed datapath, expressed as a source XCD and destination HBM pair (§3.2); and the resource configuration, like the number of active CUs, which controls probe intensity. Bandwidth pings use 10–16 CUs, with probe bandwidth scaling approximately linearly (e.g., ≈50 GB/s/CU on MI300X), whereas latency pings use a single CU. Further, it specifies the number of profiling iterations, e.g., pointer-chasing operations for latency pings, and the memory buffer pinned to the destination HBM by the HBM Stack Localizer (§3.2).

Algorithm 4 Differential Congestion Attribution (DCA)

1: DEVIATES(p): compares ping p against its calibrated baseline $\hat{p}$ in mean/tail latency or bandwidth.
2: COMMONSLOWDOWN(G_l): returns minimum slowdown among pings in G_l that traverse link l.
3: **for all** ping $p \in$ `PingReport` **do**
4: **if not** DEVIATES(p) **then continue**
5: **for all** link $l \in$ Path(p) **do**
6: $G_l \leftarrow$ ping $q \mid q$ traverses link l
7: **if** $\forall q \in G_l$, DEVIATES(q) **then**
8: $\lambda_l \leftarrow$ COMMONSLOWDOWN(G_l)
9: Normalize λ_l into attribution shares s_l such that $\sum_{l \in \text{Path}(p)} s_l = 1$
10: **for all** link $l \in$ Path(p) **do**
11: $\Delta L_l(p) \leftarrow s_l \cdot \Delta L(p)$

By default, `PingPoint` synthesizes an exhaustive `PingPlan` containing `PingSpecs` spanning all supported ping types, topological endpoint pairs, and sampled resource allocations. `PingPoint` currently implements two built-in ping types, `lat` and `bw` in §3.1, while allowing users to register new ping implementations, add them to a `PingPlan`, and customize the plan to focus on specific datapaths or reduce profiling overhead. Once constructed, a `PingPlan` can be reused across target kernels within the same application. For example, in §5, we profile four Mixture-of-Experts kernels in a single execution by reusing a common `PingPlan`.

4.6 Profiling Engine

`PingPoint` orchestrates controlled probe injection and application replay to profile the ACN from multiple angles. It executes `PingPlan` in two phases: a baseline calibration phase and a concurrent profiling phase. During calibration, the engine sequentially executes each ping in isolation, without co-running the target kernel, to establish baseline latency and bandwidth distributions for each probed datapath and resource configuration. These unloaded measurements capture the steady-state characteristics of the ACN and serve as ground truth for the analyzer when attributing subsequent performance deviations to application-induced contention.

In the concurrent profiling phase, the engine executes each probe in `PingPlan` alongside the target kernel using a `fused_kernel` that synchronizes probe and application execution via barriers (§4.4). For each `PingSpec`, the engine dispatches the corresponding microbenchmark to thread blocks pinned to the designated source XCD and passes both the specification and its output record (`PingResult`) to the probe logic. Latency pings perform pointer chasing over destination buffers to expose end-to-end delay, while bandwidth pings stream over the same buffers and apply reductions to sustain traffic at the configured footprint (§3.1). During each profiling iteration, the engine records cycle-level timing samples into the associated `PingResult` structure, preserving fine-grained distributions that enable downstream inference of queueing dynamics and contention inside the ACN.

4.7 Profiling Result Analyzer

`PingPoint` analyzes profiling results using **Differential Congestion Attribution (DCA)** to attribute observed congestion to individual ACN links. Algorithm 4 outlines the workflow of the analyzer, which operates on each ping result and uses two helper functions: DEVIATES checks whether the profiled measurement (p) differs considerably from its calibrated baseline ($\hat{p}$) in mean/tail latency or bandwidth; COMMONSLOWDOWN computes the minimum latency increase or bandwidth drop among pings G_l sharing link l, providing a conservative estimate of congestion on that link.

Algorithm 5 DCA Helper Methods

function SLOWDOWN(p)
$\delta_{\text{mean}} \leftarrow \frac{L_{\text{mean}}(p) - \hat{L}_{\text{mean}}(p)}{\hat{L}_{\text{mean}}(p)}$ $\delta_{\text{p99}} \leftarrow \frac{L_{\text{p99}}(p) - \hat{L}_{\text{p99}}(p)}{\hat{L}_{\text{p99}}(p)}$ $\delta_{\text{bw}} \leftarrow \frac{\hat{T}_{\text{bw}}(p) - T_{\text{bw}}(p)}{\hat{T}_{\text{bw}}(p)}$
return $(\delta_{\text{mean}}, \delta_{\text{p99}}, \delta_{\text{bw}})$
function DEVIATES(p)
$(\delta_{\text{mean}}, \delta_{\text{p99}}, \delta_{\text{bw}}) \leftarrow$ SLOWDOWN(p)
return $(\delta_{\text{mean}} > \epsilon_{\text{mean}}) \vee (\delta_{\text{p99}} > \epsilon_{\text{p99}}) \vee (\delta_{\text{bw}} > \epsilon_{\text{bw}})$
function COMMONSLOWDOWN(G_l)
return $\min_{q \in G_l} \max$ (SLOWDOWN(q))

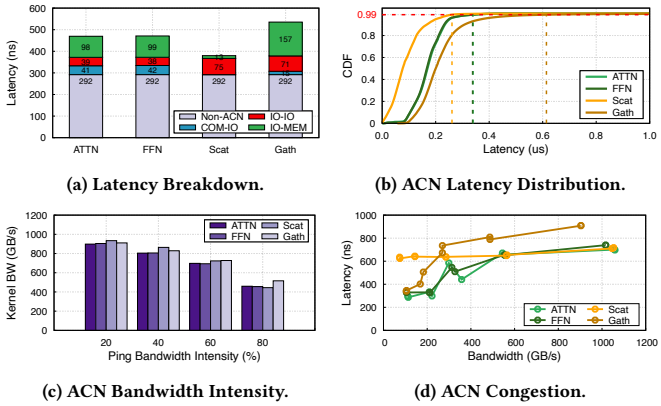
Algorithm 5 details the helper methods. For a latency ping p, the analyzer extracts the mean latency $L_{\text{mean}}(p)$ and tail latency $L_{\text{p99}}(p)$. For a bandwidth ping p, it extracts the achieved throughput $T_{\text{bw}}(p)$. These runtime measurements are compared against their calibrated baselines: $\hat{L}_{\text{mean}}(p)$, $\hat{L}_{\text{p99}}(p)$, and $\hat{T}_{\text{bw}}(p)$. SLOWDOWN(p) converts the runtime measurements into normalized deviations from baseline: δ_{mean}, δ_{p99}, and δ_{bw}. The latency terms quantify the relative latency increase over baseline, while the bandwidth term captures the relative bandwidth drop from baseline. DEVIATES(p) compares these deviations against tolerance thresholds, ϵ_{mean}, ϵ_{p99}, and ϵ_{bw}, and returns true if any metric exceeds its threshold. This condition allows DCA to detect contention that manifests as mean-latency inflation, tail-latency inflation, or bandwidth degradation. Empirically, we set ϵ_{bw} to 0.15, while ϵ_{mean} and ϵ_{p99} are much smaller. Given a group of pings G_l that all traverse link l, COMMONSLOWDOWN(G_l) first reduces each ping q's three-dimensional slowdown tuple to a scalar by taking the maximum across the metrics. It then returns the minimum scalar slowdown among pings in G_l. This value provides a conservative estimate of the slowdown common to all traffic crossing link l and is used as the congestion indicator for link l (i.e., λ_l in Algorithm 4).

In the main DCA (Algorithm 4), it iterates over each record p in `PingReport` (L3). If the observed runtime metrics closely match the baseline, it concludes that the ping's datapath experiences no ACN-induced contention and skips further attribution (L4). Otherwise, the deviation indicates that at least one ACN link along the datapath is congested. To localize the contention source, it inspects each link l on the ping's datapath (L5) and forms G_l, the set of pings whose datapaths also traverse link l (L6). If all pings in G_l exhibit statistically significant slowdowns from their baselines, it infers that the shared link l is a common congestion point (L7). It then records the minimum contention magnitude common to these pings as λ_l (L8), capturing a lower bound on congestion attributable to link l.

The analyzer normalizes the congestion indicators of all links on the ping's datapath into proportional shares s_l (L9). Finally, it attributes the ping's excess end-to-end latency, $\Delta L(p)$, across the traversed links according to these shares (L10–11), ensuring that the per-link attributions sum to the observed latency increase. The resulting decomposition produces a link-level congestion profile that localizes ACN bottlenecks and explains how the application stresses the network, forming the basis of `PingPoint`'s performance reports.

5 Evaluation

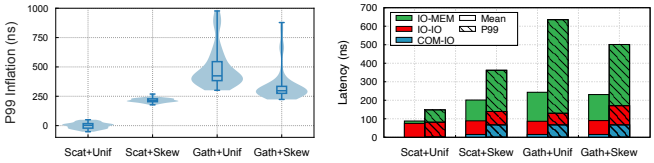
We use the same experimental setup as in §3.1 and evaluate `PingPoint` with MoE kernels [56, 152, 178], widely used in modern LLMs [51, 60, 132]. MoE augments standard Transformer layers with a routing



(a) Latency Breakdown. (b) ACN Latency Distribution.

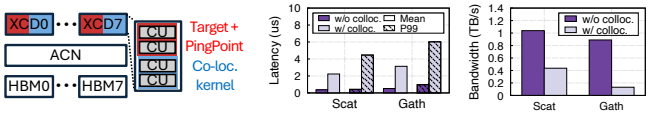
(c) ACN Bandwidth Intensity. (d) ACN Congestion.

Figure 8: ACN-centric performance report of four applications.



(a) Tail Latency. (b) Latency Breakdown.

Figure 9: ACN performance comparison between uniform and skewed traffic patterns of `Scat` and `Gath` kernels.



(a) Co-located Deployment. (b) Latency. (c) Bandwidth.

Figure 10: ACN-level performance interference under collocation.

mechanism and relies on four core kernels: Attention (`ATTN`), Scatter (`Scat`), Feedforward Network (`FFN`), and Gather (`Gath`). In the MoE execution pipeline, tokens are first processed by `ATTN`, distributed to experts via `Scat` according to a gating function, transformed by `FFN`, and finally merged by `Gath`. Our experiments instantiate eight experts, each mapped to a single XCD, and evaluate both uniform and highly skewed (90%) routing distributions [56, 178] to stress different ACN traffic patterns.

5.1 Case #1: ACN-Centric Performance Profiling

`PingPoint` exposes ACN behaviors that prior profiling tools cannot observe by decomposing end-to-end slowdowns into per-link contributions. In Figure 8a, the `PingPoint` analyzer shows that IO-MEM links dominate `Gath`, consistent with many-to-one incast from multiple IO chiplets into a single HBM stack. By contrasting calibrated baselines with co-running measurements, it further reveals that ACN traversal accounts for 53.3–67.4% of memory-request tail latency across kernels (Figure 8b), demonstrating that on-package communication frequently lies on the critical path. Additionally, `PingPoint` infers link-level bandwidth intensity via bandwidth pings (Figure 8c), enabling us to distinguish ACN-saturated phases from compute- or memory-limited ones. For example, at the highest bandwidth ping intensity, the ACN-induced latencies of `ATTN`, `FFN`, `Scat`, and `Gath` rise to 701 ns, 739 ns, 718 ns, and 910 ns, respectively, which are 2.4×, 2.2×, 1.2×, and 2.7× higher than in the low-intensity case (Figure 8d). `PingPoint` attributes these increases to transient queue buildup inside COM-IO, IO-IO, and IO-MEM links rather than to reduced wavefront occupancy or HBM throttling, illustrating how ACN-centric profiling diagnoses otherwise opaque performance bottlenecks.

5.2 Case #2: Input-Dependent Perf. Exploration

`PingPoint` allows developers to connect application-level inputs to ACN stress patterns by correlating kernel execution with per-link latency and contention. We compare two MoE routing scenarios: a uniform distribution, where tokens are evenly spread across experts, and a skewed distribution, where seven experts target a single destination. As shown in Figure 9, `PingPoint` reports markedly higher tail latency under skew, reflecting concentrated incast into a subset of HBM stacks. For `Scat`, it attributes the slowdown primarily to IO-MEM links, whose average and tail latencies rise to 112 ns and 223 ns, consistent with the workload's many-to-one traffic pattern. For `Gath`, it shows that beyond a 2.4× increase on IO-MEM, IO-IO links also experience a 1.4× latency increase. These results demonstrate the effectiveness of `PingPoint`'s in-situ probing and link-level attribution, enabling developers to reason about mitigation strategies such as expert placement and traffic shaping.

5.3 Case #3: Performance Interference Diagnostic

`PingPoint` enables reasoning about cross-application interference and collocation even when kernels are spatially isolated on disjoint CUs. We co-run `Scat` or `Gath` with a background `ATTN` kernel placed on a separate stream and pinned to non-overlapping CUs (Figure 10a), ensuring that any slowdown arises from shared ACN links rather than compute contention. Despite this spatial separation, `PingPoint` reports severe interference (Figures 10b/10c): for `Scat`, mean and P99 latencies rise by 6.2× and 10.6×, respectively, while achieved bandwidth drops by 58%; for `Gath`, the corresponding slowdowns are 6.1× and 6.2×, with bandwidth reduced by 85%, indicating that collocation is substantially more harmful for incast-heavy gathers. These degradations are primarily attributed to queue buildup inside shared ACN components. Under collocation, `Scat`'s mean latency is decomposed into 108 ns (COM-IO), 1826 ns (IO-IO), and 111 ns (IO-MEM), while `Gath`'s breaks down into 70 ns, 2747 ns, and 254 ns, respectively. Tail-latency attribution further exposes severe IO-IO congestion: for P99, `Scat` reaches 4038 ns on IO-IO, and `Gath` climbs to 5564 ns, far exceeding COM-IO and IO-MEM contributions. Hence, `PingPoint` can provide concrete guidance for collocation policies that must account for on-package network pressure rather than CU utilization alone.

5.4 Case #4: System Overhead Under Deployment

`PingPoint` Overhead. `PingPoint`'s primary overhead is bandwidth (BW) pings. A BW ping measures the residual bandwidth P of a shared datapath and estimates the target application's bandwidth as $\hat{T} = (C - P)/(1 - q_b)$, where C is the path's calibrated peak bandwidth and q_b is the calibrated slowdown induced by a BW ping of intensity b.[2] As shown in Table 2, `PingPoint` achieves high estimation fidelity, with 7.2% average MAPE across staircase

[2] b is determined by the number of CUs launching BW probes. On MI300X, BW pings using 10–16 CUs yield q_b values of 0.174, 0.204, 0.234, 0.257, 0.285, 0.304, and 0.329.

	Staircase			Random		
	MAPE	Eff. ratio	BW Cost	MAPE	Eff. ratio	BW Cost
1 Ping	5.4–7.0	60.9	409.1/460.5	7.7–8.5	78.1	383.7/426.9
Ping set		91.0	454.8/536.5		96.9	418.6/478.0

Table 2: Comparison of BW ping error estimation (%), probe effectiveness ratio (%), and mean/peak bandwidth cost (GB/s).

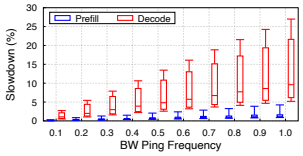


Figure 11: Application slowdown varying BW ping frequency.

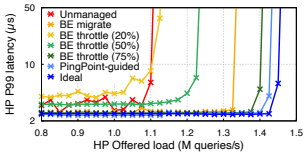


Figure 12: Tail latency of HP application under BE contention.

and random traffic patterns, but causes mean and peak bandwidth costs of 409.1 and 460.5 GB/s for staircase, and 383.7 and 426.9 GB/s for random. Further, increasing the number of BW pings improves profiling coverage by exposing a broader range of traffic conditions. To quantify this, we define the *Probe Effectiveness Ratio* (*PER*) as the fraction of measurements that observe sufficient signal for reliable estimation. Compared to a single-ping, a seven-ping set increases PER by 24.4% across the two traffic patterns while incurring at most 14.2% additional bandwidth cost. Thus, `PingPoint` exposes a tunable tradeoff between profiling fidelity and bandwidth cost.

`PingPoint` Application Impact. To evaluate `PingPoint`'s impact on target applications, we use *Prefill* and *Decode*, two LLM inference phases with distinct resource demands. *Prefill* is compute-bound, whereas *Decode* is memory-bound [45, 165]. Both are instrumented from the MoE kernels under a uniform expert-token distribution (§5.2). We vary the BW-ping frequency from 0.1 to 1.0, controlling how often pings are injected during continuous inference. Figure 11 shows *Prefill* experiences marginal slowdown (1.6%/4.2% mean/peak), since its kernel consumes only a small fraction of ACN bandwidth. In contrast, *Decode* sees higher overhead (13.2%/27.9% mean/peak) because its token-by-token execution exposes limited parallelism and is highly sensitive to ACN contention due to weight and KV-cache accesses. Note that this overhead hinges on the ping frequency. For example, injecting BW pings once every three iterations reduces *Decode*'s slowdown to 7.9%. Thus, `PingPoint`'s overhead is controllable and can be tuned to balance profiling fidelity against application slowdown.

`PingPoint` Provisioning Overhead. We also quantify `PingPoint`'s setup/calibration cost and runtime resource usage. End-to-end system initialization takes 8.98s. During calibration (§4.6), executing a full ping set, comprising 64 latency probes (8 XCDs × 8 HBMs) and 320 bandwidth probes (8 XCDs × 8 HBMs × 7 CU allocations), requires 6.0s with 10K iterations per ping and increases linearly to 62.3s with 100K iterations. This calibration incurs a one-time cost. Runtime memory peaks at 6GB (≈3% of MI300X capacity), with buffers streamed to the host to minimize footprint.

`PingPoint` Deployment Tax. We develop and evaluate an end-to-end, adaptive overhead-control scheme. In the recommended configuration, latency pings remain always-on, while BW pings are triggered only after a latency anomaly is detected. We evaluate this strategy on MoE *Decode* under intermittent skew: one hot expert

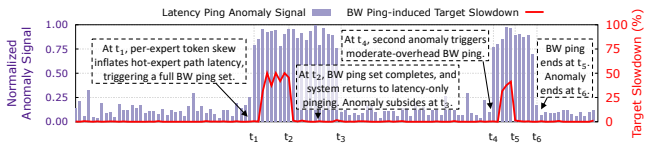


Figure 13: E2E deployment with two-tier online monitoring.

receives 12× more tokens for 25% of wall-clock time, matching observations from real MoE traces [103]. As shown in Figure 13, skew appears at t_1, where latency pings detect a surge on the hot datapath and trigger BW pings. `PingPoint` runs the full BW-ping set until t_2, then falls back to latency-only monitoring; the anomaly subsides by t_3. When another anomaly occurs at t_4, `PingPoint` reduces overhead by using only the 10-CU BW ping, lowering generated-tokens/s degradation by 15.9% relative to the full BW-ping set. Overall, adaptive two-tier monitoring incurs only 3.5% mean target slowdown, 28.1× lower than always-on BW pinging (98.2%), demonstrating that `PingPoint` can be deployed with low runtime overhead.

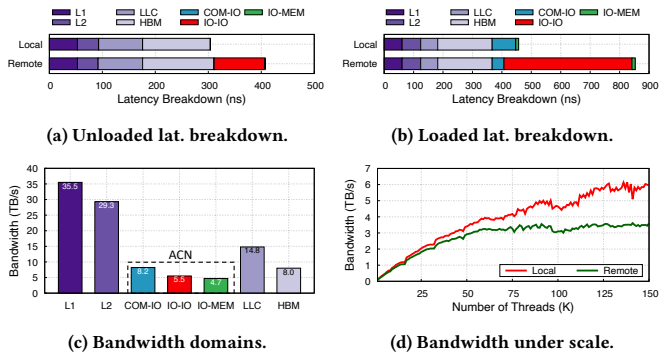
5.5 Case #5: Development Optimization Guidance

We evaluate how `PingPoint` can guide application optimization in a multi-tenant GPU-DB setting [50] with a high-priority (HP) ANN search service (10 μs P99 SLO) co-located with a bandwidth-intensive best-effort (BE) Vector Scan workload. HP runs on a dedicated XCD, while BE uses the remaining XCDs. Our primary metric is HP's *SLO capacity*: the maximum query rate that satisfies the latency SLO. We compare five policies: *Unmanaged* (default placement), *BE migrate* (randomly migrates BE embedding vectors), *BE throttle* (reduces BE execution intensity by 20%, 50%, or 75%), `PingPoint`-*guided* (relocates BE threads and data to avoid ACN contention with HP), and *Ideal* (HP alone).

As shown in Figure 12, `PingPoint`-*guided* achieves an SLO capacity of 1.43M queries/s, within 2% of *Ideal* (1.45M), compared to 1.1M for *Unmanaged* and 1.33M for *BE migrate*. This improvement comes from identifying BE datapaths that contend with HP traffic in the ACN and relocating BE threads and embedding vectors to reduce overlap. While aggressive BE throttling reaches up to 1.4M queries/s, it sacrifices 30.3–47.6% of BE throughput relative to `PingPoint`-*guided*. Appendix §F also shows that ACN-aware GPU partitioning can improve locality: under NPS4, COM-IO links allocate more channels to local traffic than under the default NPS1 configuration, benefiting applications whose working sets fit within a Chiplet Cluster (CC; §2.2). These results demonstrate that `PingPoint` exposes practical opportunities for ACN-aware optimization.

5.6 Case #6: Portability across Chiplet GPUs

We evaluate `PingPoint`'s portability on AMD's latest chiplet-based GPU, MI350X [8], launched in June 2025. Since it has two CCs (Appendix §E), we consider two traffic patterns: (a) `local` (XCD→local CC); (b) `remote` (XCD→non-local CC). Benchmarking follows the methodology in §3.2. Figure 14 shows that ACN bottlenecks persist, and become even more pronounced, on MI350X. The unloaded IO-IO hop latency increases to 96 ns (1.8× MI300X), while the ACN accounts for up to 57.0% of loaded memory-access latency (486 ns of 853 ns). IO-IO links alone contribute 51.0% of the total



(a) Unloaded lat. breakdown.

(b) Loaded lat. breakdown.

(c) Bandwidth domains.

(d) Bandwidth under scale.

Figure 14: MI350X performance results: (a)/(b) show the latency breakdown of a local and remote memory access under unloaded/loaded cases; (c) reports the bandwidth capacity of different path segments; (d) shows the achieved bandwidth when increasing the number of concurrent benchmarking threads.

latency, and remote-CC bandwidth saturates at nearly half of local-CC bandwidth (3.6 TB/s vs. 6.1 TB/s). These results suggest that ACN-induced bottlenecks are not specific to MI300X but remain a first-order performance concern in future chiplet-based GPUs.

Porting `PingPoint` to MI350X only requires a few hardware-specific knobs for programmable benchmarking, including thread-to-XCD identification [77, 116] and cross-HBM memory-allocation granularity (e.g., 4KB vs. 8KB interleaving). `PingPoint` automatically discovers these parameters during initialization of the XCD Pinning and HBM Stack Localizer (§3.2), requiring neither hardware modifications nor privileged access. Moreover, `PingPoint`'s generic approach (§4.2) is also applicable to NVIDIA's chiplet GPUs [88, 126], which have similar metrics to AMD GPUs (Appendix §B).

6 Related Work

Architectural Profiling. Our work is inspired by prior work on building hardware counter-assisted architectural profilers. For example, the Top-Down analysis [173] views the processor pipeline as a frontend-backend structure, and identifies dominant performance bottlenecks via a principled hierarchical approach. Google's GWP [147] randomly samples server counters continuously across the data center, symbolizes the collected samples' call stacks, and aggregates the data into the Dremel database [120] for workload analysis. Researchers then use it to uncover the "data center tax" and explore optimization opportunities [85]. Melody [109] develops a CXL memory characterization framework and introduces a stall cycle-based slowdown prediction approach. Tintin [104] develops an elastic profiling window adjustment algorithm based on learned mismatched errors and introduces a new OS primitive (ePX) to unify architectural profiling. Like these works, `PingPoint` also performs profiling, but it specifically targets ACNs, which require specialized mechanisms that prior solutions do not provide.

Host Networking. Researchers have revisited the design of the host network under high bandwidth and new interconnect protocols. PCIe-bench [123] characterizes the PCIe subsystem and uncovers the impact of PCIe features (like IOMMU). Cai *et al.* [35] present the limitations of the Linux networking stack for 100Gbps links. HostCC [3] develops a host-native congestion-control architecture based on curated signals to allocate bandwidth resources. Vuppalapati *et al.* [166] introduce an abstract domain-by-domain credit-based flow control scheme to study the host network conceptually. PathFinder [107] dissects the `CXL.mem` protocol using Intel performance counters. CEIO [108] proposes proactive rate control and elastic buffering to achieve zero LLC misses in the IO data path. MemChannel [65] develops a CSFQ-inspired transport for switched CXL memory pooling. However, we target new chiplet networks inside accelerators like GPUs, not CPU servers [23].

Network-on-Chip (NoC). The ACN is a special type of NoC. Bjerregaard *et al.* [34] survey prior NoC efforts and define the basic networking structure that motivates `PingPoint`. Our characterization shows that an ACN follows the networking topology [5, 93, 119], routing protocol [1, 53, 59], resource management [42, 105, 121, 125], and traffic control [37, 115, 162] of a NoC, but differs in chiplet-induced communication idiosyncrasies. Besides, we believe that the ACN can greatly benefit from our past exploration on programmable networks [38, 64, 150, 151, 168, 175–177] and in-network computing [69, 111–114, 117, 140, 142, 143, 149].

Chiplet Architecture Optimization. Recent work has improved chiplet accelerator designs. For example, MGvm [141] maps virtual addresses to chiplets and places page tables using chiplet-related statistics. CLAP [136] predicts page groups and pre-organizes them into contiguous physical frames based on chiplet locality. Although these works improve the efficiency of chiplet-based accelerators, they solve different problems from `PingPoint`, which introduces an abstract ACN model and can easily adapt to new chiplet designs.

GPU Kernel Optimizations. Recently, there has been a shift towards application-defined scheduling [48, 124, 153]. Chiplet-aware scheduling remaps computational work to physical compute chiplets by considering thread-scheduling context [43, 44, 71]. Co-locating thread blocks and memory pages [25, 29, 89] enables application-defined memory allocation. These techniques highlight the variety of ways to improve efficiency for GPU kernels. However, they often rely on ad hoc or heuristic-based solutions. Thus, `PingPoint` can facilitate further development of similar techniques.

7 Conclusion

This paper introduces the Accelerator Chiplet Network (ACN) and shows that its opaque behavior fundamentally limits profiling and optimization with existing tools. By developing a programmable ACN characterization framework and `PingPoint`, we demonstrate that modeling the ACN as a hose-based logical graph with tailored queueing abstractions and in-situ software pinging enables precise link-level diagnosis of congestion and performance bottlenecks. Our evaluation shows that `PingPoint` uncovers hidden contention, guides kernel placement and traffic shaping, and delivers actionable optimization insights with marginal overhead. Given the increasing prevalence of chiplets, `PingPoint` thus provides important, broadly applicable capabilities for chiplet-based systems.

Ethics. This work does not raise ethical issues.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and feedback. This work is supported in part by NSF grants CNS-2212192, CAREER-2238608, and CAREER-2339755. Computational resources are supported in part by Advanced Micro Devices, Inc. under the AMD University Program's AI & HPC Cluster.

References

[1] Ankur Agarwal, Cyril Iskander, and Ravi Shankar. 2009. Survey of Network on Chip (NoC) Architectures & Contributions. *Journal of Engineering, Computing and Architecture* 3, 1 (2009), 21–27.

[2] Sugam Agarwal, Murali Kodialam, and T. V. Lakshman. 2013. Traffic Engineering in Software Defined Networks. In *Proceedings of the IEEE INFOCOM*. IEEE Press, Piscataway, NJ, USA, 2211–2219.

[3] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. 2023. Host Congestion Control. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery, New York, NY, USA, 275–287.

[4] Satyajeet S. Ahuja, Varun Gupta, Vinayak Dangui, Soshant Bali, Abishek Gopalan, Hao Zhong, Petr Lapukhov, Yiting Xia, and Ying Zhang. 2021. Capacity-Efficient and Uncertainty-Resilient Backbone Network Planning With Hose. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery, New York, NY, USA, 547–559.

[5] Ian F. Akyildiz, Fernando Brunetti, and Cristina Blázquez. 2008. Nanonetworks: A New Communication Paradigm. *Computer Networks* 52, 12 (2008), 2260–2279.

[6] AMD. 2023. AMD CDNA™ 3 Architecture. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf. (2023).

[7] AMD. 2023. AMD Instinct™ MI300X Accelerators. https://www.amd.com/en/products/accelerators/instinct/mi300/mi300x.html. (2023).

[8] AMD. 2023. AMD Instinct™ MI350X Accelerators. https://www.amd.com/en/products/accelerators/instinct/mi350/mi350x.html. (2023).

[9] AMD. 2024. AMD AI & HPC Fund: Accelerate Your Research with AMD. https://www.amd.com/en/corporate/hpc-fund.html. (2024).

[10] AMD. 2025. AMD CDNA™ 4 Architecture. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-4-architecture-whitepaper.pdf. (2025).

[11] AMD. 2025. AMD Chiplet Ecosystem. https://www.amd.com/content/dam/amd/en/documents/solutions/technologies/chiplet-architecture-white-paper.pdf. (2025).

[12] AMD. 2025. AMD Global Memory Interconnect. https://docs.amd.com/r/en-US/pg292-ethernet-1-10-25g/GMI-Transmission. (2025).

[13] AMD. 2025. AMD Infinity Fabric™ Link. https://docs.amd.com/v/u/en-US/AMD_Infinity_Fabric_Link_User_Guide_56978. (2025).

[14] AMD. 2025. AMD Instinct MI300X GPU Partitioning Overview. https://instinct.docs.amd.com/projects/amdgpu-docs/en/latest/gpu-partitioning/mi300x/overview.html. (2025).

[15] AMD. 2025. AMD Instinct MI300X system optimization. https://instinct.docs.amd.com/projects/amdgpu-docs/en/docs-30.10/system-optimization/mi300x.html. (2025).

[16] AMD. 2025. Deep dive into the MI300 compute and memory partition modes. https://rocm.blogs.amd.com/software-tools-optimization/compute-memory-modes/README.html. (2025).

[17] AMD. 2025. MI300 and MI200 Series Performance Counters and Metrics. https://rocm.docs.amd.com/en/latest/conceptual/gpu-arch/mi300-mi200-performance-counters.html. (2025).

[18] AMD. 2026. Cooperative Groups. https://rocm.docs.amd.com/projects/HIP/en/latest/tutorial/cooperative_groups_tutorial.html. (2026).

[19] AMD. 2026. MI300X L2 cache counters. https://github.com/ROCm/rocm-systems/blob/develop/projects/rocprofiler-compute/src/rocprof_compute_soc/analysis_configs/gfx942/1700_l2_cache.yaml. (2026).

[20] AMD. 2026. ROCm Compute Profiler documentation. https://rocm.docs.amd.com/projects/rocprofiler-compute/en/latest/. (2026).

[21] AMD. 2026. ROCprofiler-SDK documentation. https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/latest/. (2026).

[22] AMD. 2026. Setting the number of compute units. https://rocm.docs.amd.com/en/latest/how-to/setting-cus.html. (2026).

[23] Seunghyun An, Joontaek Oh, and Ming Liu. 2025. Server Chiplet Networking. In *Proceedings of the ACM Workshop on Hot Topics in Networks*. Association for Computing Machinery, New York, NY, USA, 289–299.

[24] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. 2004. Sizing Router Buffers. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. Association for Computing Machinery, New York, NY, USA, 281–292. https://doi.org/10.1145/1015467.1015499

[25] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, New York, NY, USA, 320–332.

[26] Joshua Bakita and James H. Anderson. 2023. Hardware compute partitioning on NVIDIA GPUs. In *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Press, IEEE, Piscataway, NJ, USA, 54–66.

[27] James Balfour and William J Dally. 2006. Design Tradeoffs for Tiled CMP On-chip Networks. In *Proceedings of the 20th Annual International Conference on Supercomputing*. Association for Computing Machinery, New York, NY, USA, 390–401.

[28] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards Predictable Datacenter Networks. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery, New York, NY, USA, 242–253.

[29] Trinayan Baruah, Yifan Sun, Ali Tolga Dinçer, Saiful A. Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. 2020. Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems. In *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture*. IEEE Press, Piscataway, NJ, USA, 596–609.

[30] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic in-band network telemetry. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery, New York, NY, USA, 662–680.

[31] Carlo Bertolli, Thorsten Blass, Lynd Stringer, Nicole Aschenbrenner, Jan-Patrick Lehr, Doru Bercea, Dhruva Chakrabarti, Lawrence Meadows, and Ron Lieberman. 2024. Performance Analysis of Runtime Handling of Zero-Copy for OpenMP Programs on MI300A APUs. In *Proceedings of the SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Association for Computing Machinery, New York, NY, USA, 1420–1429.

[32] Jay Bharadwaj, Kishore Menezes, and Chris McKinsey. 1999. Wavefront Scheduling: Path-based Data Representation and Scheduling of Subgraphs. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Press, IEEE, Piscataway, NJ, USA, 262–271.

[33] Ravi Bhargava and Kai Troester. 2024. AMD Next-Generation "Zen 4" Core and 4th Gen AMD EPYC Server CPUs. *IEEE Micro* 44, 3 (2024), 8–17.

[34] Tobias Bjerregaard and Shankar Mahadevan. 2006. A Survey of Research and Practices of Network-on-Chip. *Comput. Surveys* 38, 1 (2006), 1–es.

[35] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery, New York, NY, USA, 65–77.

[36] Erlangen National High Performance Center. 2025. GPU benchmarks. https://github.com/RRZE-HPC/gpu-benches. (2025).

[37] Chih-Hao Chao, Kai-Yuan Jheng, Hao-Yu Wang, Jia-Cheng Wu, and An-Yeu Wu. 2010. Traffic-and Thermal-aware Run-time Thermal Management Scheme for 3D NoC Systems. In *Proceedings of the 4th ACM/IEEE International Symposium on Networks-on-Chip*. IEEE Press, IEEE, Piscataway, NJ, USA, 223–230.

[38] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, and Zeke Wang. 2024. Demystifying Datapath Accelerator Enhanced Off-path SmartNIC. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP'24)*. 1–12.

[39] Zhiwen Chen, Jiaju Zhang, Shizhao Wang, and Ching-Ping Wong. 2024. Challenges and Prospects for Advanced Packaging. *Fundamental Research* 4, 6 (2024), 1455–1458.

[40] Chips and Cheese. 2023. AMD's CDNA3 Compute Architecture. https://chipsandcheese.com/p/amds-cdna-3-compute-architecture. (2023).

[41] Chips and Cheese. 2025. Inside the AMD Instinct MI300A's Giant Memory Subsystem. https://chipsandcheese.com/p/inside-the-amd-radeon-instinct-mi300as. (2025).

[42] Chen-Ling Chou and Radu Marculescu. 2011. FARM: Fault-Aware Resource Management in NoC-based Multiprocessor Platforms. In *Design, Automation & Test in Europe*. IEEE Press, Piscataway, NJ, USA, 1–6.

[43] Mansi Choudhary, Karthik Sangaiah, Sonali Singh, Muhammad Osama, Lisa Wu Wills, and Ganesh Dasika. 2025. Optimizing Attention on GPUs by Exploiting GPU Architectural NUMA Effects. *arXiv preprint arXiv:2511.02132* (2025), 11.

[44] Sangeeta Chowdhary, Ryan Swann, Sean Siddens, Muhammad Osama, Stephen Neuendorffer, Alexandru Dutu, Karthik Sangaiah, Sandeepa Bhuyan, Samuel Bayliss, and Ganesh Dasika. 2026. Fleet: Hierarchical Task-based Abstraction for Megakernels on Multi-Die GPUs. *arXiv preprint arXiv:2604.15379* (2026), 12.

[45] Google Cloud. 2026. Five techniques to reach the efficient frontier of LLM inference. https://cloud.google.com/blog/topics/developers-practitioners/five-techniques-to-reach-the-efficient-frontier-of-llm-inference. (2026).

[46] GitHub Commit. 2023. Remove non-functional counters for MI200 and MI300. https://github.com/ROCm/rocm-systems/commit/1471dc8a7640687c82197b2e8d0f65c63b1adfa9. (2023).

[47] UALink Consortium. 2025. Ultra Accelerator Link Specification. https://ualinkconsortium.org/specifications/. (2025).

[48] Patrick H. Coppock, Brian Zhang, Eliot H. Solomon, Vasilis Kypriotis, Leon Yang, Bikash Sharma, Dan Schatzberg, Todd C. Mowry, and Dimitrios Skarlatos. 2025. LithOS: An Operating System for Efficient Machine Learning on GPUs. In *Proceedings of the 31st Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3731569.3764818

[49] William James Dally and Brian Patrick Towles. 2004. *Principles and practices of interconnection networks*. Elsevier, New York, NY.

[50] Voltron Data. 2026. Theseus. https://github.com/voltrondata. (2026).

[51] DeepSeek. 2025. DeepSeek-V3. https://www.deepseek.com/en/. (2025).

[52] Behzad Dehlaghi, Nijwm Wary, and Tony Chan Carusone. 2019. Ultra-Short-Reach Interconnects for Die-to-Die Links: Global Bandwidth Demands in Microcosm. *IEEE Solid-State Circuits Magazine* 11, 2 (2019), 42–53.

[53] Andrew DeOrio, David Fick, Valeria Bertacco, Dennis Sylvester, David Blaauw, Jin Hu, and Gregory Chen. 2012. A Reliable Routing Architecture and Algorithm for NoCs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 5 (2012), 726–739.

[54] N. G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, K. K. Ramakrishnan, and Jacobus E. van der Merive. 1999. A Flexible Model for Resource Management in Virtual Private Networks. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. Association for Computing Machinery, New York, NY, USA, 95–108. https://doi.org/10.1145/316188.316209

[55] Niklas Eiling, Stefan Lankes, and Antonello Monti. 2023. Checkpoint/Restart for CUDA Kernels. In *Proceedings of the SC23-W: Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. Association for Computing Machinery, New York, NY, USA, 1729–1737.

[56] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.

[57] Yinxiao Feng, Dong Xiang, and Kaisheng Ma. 2023. Heterogeneous Die-to-Die Interfaces: Enabling More Flexible Chiplet Interconnection Systems. In *Proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, Piscataway, NJ, USA, 930–943.

[58] Gettys, Jim and Nichols, Kathleen. 2012. Bufferbloat: Dark Buffers in the Internet. *Commun. ACM* 55, 1 (2012), 57–65.

[59] Roman Gindin, Israel Cidon, and Idit Keidar. 2007. NoC-based FPGA: Architecture and Routing. In *Proceedings of the 1st International Symposium on Networks-on-Chip*. IEEE Press, Piscataway, NJ, USA, 253–264.

[60] Google. 2025. Gemini 3. https://gemini.google.com/. (2025).

[61] Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu. 2009. Express Cube Topologies for on-Chip Interconnects. In *Proceedings of the 15th IEEE International Symposium on High Performance Computer Architecture*. IEEE Press, Piscataway, NJ, USA, 163–174.

[62] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*. ACM, New York, NY, USA, 202–215.

[63] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the ACM SIGCOMM Conference*. ACM, New York, NY, USA, 139–152.

[64] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. 2023. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*. 916–929.

[65] Zerui Guo, Emily Shriver, and Ming Liu. 2026. Building A CSFQ-Inspired Transport for Switched CXL Memory Pooling. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'26)*. 969–986.

[66] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, USA, 193–208.

[67] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-Driven Streaming Network Telemetry. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery, New York, NY, USA, 357–371.

[68] Bagus Hanindhito and Bhavesh Patel. 2025. Characterizing Performance, Power, and Energy of AMD CDNA3 GPU Family. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA, 905–934.

[69] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. 2023. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*. 33–47.

[70] Jingcao Hu and Radu Marculescu. 2003. Energy-aware Mapping for Tile-based NoC Architectures Under Performance Constraints. In *Proceedings of the Asia and South Pacific Design Automation Conference*. IEEE Press, Piscataway, NJ, USA, 233–239.

[71] William Hu, Drew Wadsworth, Sean Siddens, Stanley Winata, Daniel Y. Fu, Ryann Swann, Muhammad Osama, Christopher Ré, and Simran Arora. 2025. HipKittens: Fast and Furious AMD Kernels. *arXiv preprint arXiv:2511.08083* (2025), 41.

[72] Yoonjae Hwang, Sungwook Moon, Seungki Nam, and Jeong HoonAhn. 2022. Chiplet-based System PSI Optimization for 2.5D/3D Advanced Packaging Implementation. In *Proceedings of the 72nd IEEE Electronic Components and Technology Conference*. IEEE Press, Piscataway, NJ, USA, 12–17.

[73] IDTechEx. 2025. Chiplet Technology 2025-2035: Technology, Opportunities, Applications. https://www.idtechex.com/en/research-report/chiplet-technology-2025/1041. (2025).

[74] imec. 2025. Chiplets: Piecing Together the Next Generation of Chips. https://www.imec-int.com/en/articles/chiplets-piecing-together-next-generation-chips-part-i. (2025).

[75] Intel. 2025. Intel's Gaudi 3 AI Accelerators. https://www.intel.com/content/www/us/en/products/details/processors/ai-accelerators/gaudi.html. (2025).

[76] Intel. 2025. The Intel VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. (2025).

[77] GitHub Issue. 2023. Obtain CU physical ID from HIP kernel? https://github.com/ROCm/ROCm/issues/2059. (2023).

[78] GitHub Issue. 2025. Matmul performance issue on MI300X. https://github.com/triton-lang/triton/issues/4959. (2025).

[79] GitHub Issue. 2025. MI300X SCLK stuck around 60MHz under full load. https://github.com/ROCm/ROCm/issues/4414. (2025).

[80] GitHub Issue. 2025. RCCL performance degradation on AMD Instinct MI300X GPU with AMD Pollara AI NIC. https://github.com/ROCm/ROCm/issues/5717. (2025).

[81] GitHub Issue. 2025. rccl tests on two Mi300x nodes gives me poor performance using mpirun. https://github.com/ROCm/rccl-tests/issues/146. (2025).

[82] Charles Jamieson, Anushka Chandrashekar, Ian McDougall, and M. D. Sinclair. 2022. GAP: gem5 GPU Accuracy Profiler. In *4th gem5 Users' Workshop*. 3.

[83] JEDEC. 2025. High Bandwidth Memory (HBM3) DRAM. https://www.jedec.org/standards-documents/docs/jesd238b01. (2025).

[84] Zhixian Jin, Christopher Rocca, Jiho Kim, Hans Kasan, Minsoo Rhu, Ali Bakhoda, Tor M. Aamodt, and John Kim. 2024. Uncovering Real GPU NoC Characteristics: Implications on Interconnect Architecture. In *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, Piscataway, NJ, USA, 885–898.

[85] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[86] David G. Kendall. 1953. Stochastic Processes Occurring in the Theory of Queues and Their Analysis by the Method of the Imbedded Markov Chain. *The Annals of Mathematical Statistics* 24, 3 (1953), 338–354.

[87] Keysight. 2025. What is a Chiplet, and Why Should You Care? https://www.keysight.com/blogs/en/tech/sim-des/what-is-a-chiplet-and-why-should-you-care. (2025).

[88] Brucek Khailany and Jonah Alben. 2024. NVIDIA Blackwell Platform: Advancing Generative AI and Accelerated Computing. In *Proceedings of the IEEE Hot Chips 36 Symposium*. IEEE Press, Piscataway, NJ, USA, 1–33.

[89] Mahmoud Khairy, Vadim Nikiforov, David Nellans, and Timothy G. Rogers. 2020. Locality-Centric Data and Threadblock Management for Massive GPUs. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Press, Piscataway, NJ, USA, 1022–1036.

[90] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[91] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J. Wobker. 2015. In-band Network Telemetry via Programmable Dataplanes. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery, New York, NY, USA, 1–2.

[92] Hyojoon Kim and Nick Feamster. 2013. Improving Network Management with Software Defined Networking. *IEEE Communications Magazine* 51, 2 (2013), 114–119.

[93] John Kim, James Balfour, and William Dally. 2007. Flattened Butterfly Topology for On-Chip Networks. In *Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, USA, 172–182. https://doi.org/10.1109/MICRO.2007.15

[94] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. IEEE Press, Piscataway, NJ, USA, 77–88.

[95] Jongman Kim, Dongkook Park, Theo Theocharides, Narayanan Vijaykrishnan, and Chita R. Das. 2005. A Low Latency Router Supporting Adaptivity for On-Chip Interconnects. In *Proceedings of the 42nd Annual Design Automation Conference*. Association for Computing Machinery, New York, NY, USA, 559–564. https://doi.org/10.1145/1065579.1065726

[96] Jayacharan Kolla, Pedram Alizadeh, and Gilbert Lee. 2025. Understanding RCCL Bandwidth and xGMI Performance on AMD Instinct MI300X. https://rocm.blogs.amd.com/software-tools-optimization/mi300x-rccl-xgmi/R

EADME.html. (2025).
[97] Ganesh Kumar, Dheemanth Nagaraj, Vincent R. Freytag, Eric Delano, and Gregory S. Averill. 2012. Cache and Home Agent Memory Management. https://patents.google.com/patent/US8327228B2/en. (2012).
[98] HT Kung, Trevor Blackwell, and Alan Chapman. 1994. Credit-based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation and Statistical Multiplexing. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*. Association for Computing Machinery, New York, NY, USA, 101–114.
[99] HT Kung and Koling Chang. 1995. Receiver-Oriented Adaptive Buffer Allocation in Credit-Based Flow Control for ATM Networks. In *Proceedings of the IEEE INFOCOM*. IEEE Press, Piscataway, NJ, USA, 239–252.
[100] HT Kung and Robert Morris. 1995. Credit-Based Flow Control for ATM Networks. *IEEE Network* 9, 2 (1995), 40–48.
[101] John H. Lau. 2022. Recent Advances and Trends in Advanced Packaging. *IEEE Transactions on Components, Packaging and Manufacturing Technology* 12, 2 (2022), 228–252.
[102] John H. Lau. 2025. Current Advances and Outlooks in Hybrid Bonding. *IEEE Transactions on Components, Packaging and Manufacturing Technology* 15, 4 (2025), 651–681.
[103] Yiran Lei, Dongjoo Lee, Liangyu Zhao, Daniar Kurniawan, Chanmyeong Kim, Heetaek Jeong, Changsu Kim, Hyeonseong Choi, Liangcheng Yu, Arvind Krishnamurthy, Justine Sherry, and Eriko Nurvitadhi. 2026. FAST: An Efficient Scheduler for All-to-All GPU Communication. In *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, USA, 2515–2531.
[104] Ao Li, Marion Sudvarg, Zihan Li, Sanjoy Baruah, Chris Gill, and Ning Zhang. 2025. Tintin: A Unified Hardware Performance Profiling Infrastructure to Uncover and Manage Uncertainty. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, USA, 575–593.
[105] Bin Li, Li Zhao, Ravi Iyer, Li-Shiuan Peh, Michael Leddige, Michael Espig, Seung Eun Lee, and Donald Newell. 2011. CoQoS: Coordinating QoS-Aware Shared Resources in NoC-Based SoCs. *J. Parallel and Distrib. Comput.* 71, 5 (2011), 700–713.
[106] Shenggao Li, Mu-Shan Lin, Wei-Chih Chen, and Chien-Chun Tsai. 2024. High-Bandwidth Chiplet Interconnects for Advanced Packaging Technologies in AI/ML Applications: Challenges and Solutions. *IEEE Open Journal of the Solid-State Circuits Society* 4 (2024), 351–364.
[107] Xiao Li, Zerui Guo, Yuebin Bai, Mahesh Ketkar, Hugh Wilkinson, and Ming Liu. 2025. Understanding and Profiling CXL.mem Using PathFinder. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery, New York, NY, USA, 758–779.
[108] Bowen Liu, Xinyang Huang, Qijing Li, Zhuobin Huang, Yijun Sun, Wenxue Li, Junxue Zhang, Ping Yin, and Kai Chen. 2025. CEIO: A Cache-Efficient Network I/O Architecture for NIC-CPU Data Paths. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery, New York, NY, USA, 381–394.
[109] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H Noh, and Huaicheng Li. 2025. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, New York, NY, USA, 1203–1217.
[110] Kefei Liu, Zhuo Jiang, Jiao Zhang, Shixian Guo, Xuan Zhang, Yangyang Bai, Yongbin Dong, Feng Luo, Zhang Zhang, Lei Wang, Xiang Shi, Haohan Xu, Yang Bai, Dongyang Song, Haoran Wei, Bo Li, Yongchen Pan, Tian Pan, and Tao Huang. 2024. R-Pingmesh: A Service-aware ROCe Network Monitoring and Diagnostic System. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery, New York, NY, USA, 554–567.
[111] Ming Liu. 2020. *Building Distributed Systems Using Programmable Networks*. University of Washington.
[112] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*. 318–333.
[113] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. 795–809.
[114] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*. 363–378.
[115] Weichen Liu, Jiang Xu, Xiaowen Wu, Yaoyao Ye, Xuan Wang, Wei Zhang, Mahdi Nikdast, and Zhehui Wang. 2011. A NoC Traffic Suite Based on Real Applications. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*. IEEE Press, IEEE, Piscataway, NJ, USA, 66–71.
[116] LLVM. 2026. HW_REG_XCC_ID. https://llvm.org/docs/AMDGPU/gfx940_hwreg.html. (2026).
[117] Liang Luo, Ming Liu, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2017. Motivating in-network aggregation for distributed deep neural network training. In *Workshop on Approximate Computing Across the Stack*.
[118] Chandra Sekhar Mandalapu, Chintan Buch, Priyal Shah, Roden Topacio, Patrick Cheng, Liwei Wang, Raja Swaminathan, Alan Smith, John Wuu, Kaushik Mysore, and Arsalan Alam. 2024. 3.5D Advanced Packaging Enabling Heterogenous Integration of HPC and AI Accelerators. In *Proceedings of the 74th IEEE Electronic Components and Technology Conference*. IEEE Press, Piscataway, NJ, USA, 798–802.
[119] Radu Marculescu, Umit Y. Ogras, Li-Shiuan Peh, Natalie E. Jerger, and Yatin Hoskote. 2008. Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 1 (2008), 3–21.
[120] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.
[121] Orlando Moreira, Jacob Jan-David Mol, and Marco Bekooij. 2007. Online Resource Management in a Multiprocessor with a Network-on-Chip. In *Proceedings of the ACM symposium on Applied Computing*. Association for Computing Machinery, New York, NY, USA, 1557–1564.
[122] Moscibroda, Thomas and Mutlu, Onur. 2009. A Case for Bufferless Routing in On-chip Networks. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, New York, NY, USA, 196–207.
[123] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery, New York, NY, USA, 327–341.
[124] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 595–610. https://doi.org/10.1145/3600006.3613163
[125] Vincent Nollet, Théodore Marescaux, Prabhat Avasare, Diederik Verkest, and J-Y Mignolet. 2005. Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles. In *Design, Automation and Test in Europe*. IEEE Press, Piscataway, NJ, USA, 234–239.
[126] NVIDIA. 2025. NVIDIA B200 GPU Accelerator. https://www.nvidia.com/en-us/data-center/dgx-b200/. (2025).
[127] NVIDIA. 2025. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems. (2025).
[128] NVIDIA. 2025. NVIDIA NVLink: High-Speed GPU Interconnect. https://www.nvidia.com/en-us/products/workstations/nvlink-bridges/. (2025).
[129] NVIDIA. 2026. Cooperative Groups. https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cooperative-groups.html. (2026).
[130] NVIDIA. 2026. NVIDIA CUDA Profiling Tools Interface (CUPTI). https://developer.nvidia.com/cupti. (2026).
[131] NVIDIA. 2026. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute. (2026).
[132] OpenAI. 2025. GPT-5. https://openai.com/gpt-5/. (2025).
[133] Muhammad Osama, Ryan Swann, Karthik Sangaiah, Sonali Singh, Ganesh Dasika, and Rajneesh Bhardwaj. 2025. Deep dive into the MI300 compute and memory partition modes. https://rocm.blogs.amd.com/software-tools-optimization/compute-memory-modes/README.html. (2025).
[134] Nathan Otterness and James H. Anderson. 2021. Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices". In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*. IEEE Press, Piscataway, NJ, USA, 24–34.
[135] Konstantinos Parasyris, Giorgis Georgakoudis, Esteban Rangel, Ignacio Laguna, and Johannes Doerfert. 2023. Scalable Tuning of (OpenMP) GPU Applications via Kernel Record and Replay. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3581784.3607098
[136] Junhyeok Park, Sungbin Jang, Osang Kwon, Yongho Lee, and Seokin Hong. 2025. Leveraging Chiplet-Locality for Efficient Memory Mapping in Multi-Chip Module GPUs. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, New York, NY, USA, 1040–1057. https://doi.org/10.1145/3725843.3756090
[137] Sudeep Pasricha and Nikil Dutt. 2010. *On-Chip Communication Architectures: System on Chip Interconnect*. Morgan Kaufmann, San Francisco, CA.
[138] Suchita Pati, Shaizeen Aga, Nuwan Jayasena, and Matthew Sinclair. 2025. GOLDYLOC: Global Optimizations & Lightweight Dynamic Logic for Concurrency. *ACM Trans. Archit. Code Optim.* 22, 2, Article 78 (July 2025), 28 pages.

https://doi.org/10.1145/3730584

[139] PCI-SIG. 2025. PCI Express Specification. https://pcisig.com/specifications. (2025).

[140] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 663–679.

[141] Bharadwaj Pratheek, Neha Jawalkar, and Arkaprava Basu. 2022. Designing Virtual Memory System of MCM GPUs. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, Piscataway, NJ, USA, 404–422. https://doi.org/10.1109/MICRO56248.2022.00036

[142] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. 2020. Clara: Performance Clarity for SmartNIC Offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*. 16–22.

[143] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. 2021. Automated SmartNIC Offloading Insights for Network Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. 772–787.

[144] Vishnu Ramadas, Daniel Kouchekinia, Ndubuisi Osuji, and Matthew D. Sinclair. 2023. Closing the Gap: Improving the Accuracy of gem5's GPU Models. In *5th gem5 Users' Workshop*. 2.

[145] Vishnu Ramadas, Daniel Kouchekinia, and Matthew D. Sinclair. 2024. Further Closing the GAP: Improving the Accuracy of gem5's GPU Models. In *6th Young Architects' Workshop (YArch)*. 2.

[146] reddit. 2025. Why Mi300x is technically superior but training performance was still lagging behind. Now faster then H100! https://www.reddit.com/r/AMD_Stock/comments/1hbraqd/why_mi300x_is_technically_superior_but_training/. (2025).

[147] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* 30, 4 (2010), 65–79.

[148] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, USA, 72–83. https://doi.org/10.1109/MICRO.2012.16

[149] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. 740–755.

[150] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. 1–16.

[151] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable Calendar Queues for High-speed Packet Scheduling . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. 685–699.

[152] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *International Conference on Learning Representations*. OpenReview.net, 19.

[153] Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen, and Haibo Chen. 2025. XSched: Preemptive Scheduling for Diverse XPUs. In *Proceedings of the 19th Symposium on Operating Systems Design and Implementation*. USENIX Association, USA, 671–692.

[154] Alan Smith and Vamsi Krishna Alla. 2025. AMD Instinct™ MI300X: A Generative AI Accelerator and Platform Architecture. *IEEE Micro* 45, 3 (2025), 41–48.

[155] Alan Smith, Eric Chapman, Chintan Patel, Raja Swaminathan, John Wuu, Tyrone Huang, Wonjun Jung, Alexander Kaganov, Hugh McIntyre, and Ramon Mangaser. 2024. 11.1 AMD Instinct MI300 Series Modular Chiplet Package–HPC and AI accelerator for Exa-Class Systems. In *Proceedings of the IEEE International Solid-State Circuits Conference*, Vol. 67. IEEE Press, Piscataway, NJ, USA, 490–492.

[156] Alan Smith and Norman James. 2022. AMD Instinct™ MI200 Series Accelerator and Node Architectures. In *Proceedings of the IEEE Hot Chips 34 Symposium*. IEEE Press, Piscataway, NJ, USA, 1–23.

[157] Alan Smith, Gabriel H. Loh, Samuel Naffziger, John Wuu, Nathan Kalyanasundharam, Eric Chapman, Raja Swaminathan, Tyrone Huang, Wonjun Jung, Alexander Kaganov, Hugh McIntyre, and Ramon Mangaser. 2025. Interconnect Design for Heterogeneous Integration of Chiplets in the AMD Instinct™ MI300X Accelerator. *IEEE Micro* 45, 1 (2025), 57–66.

[158] Alan Smith, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Samuel Naffziger, Mike Mantor, Mark Fowler Nathan Kalyanasundharam, Vamsi Alla, Nicholas Malaya, Joseph L. Greathouse, Eric Chapman, and Raja Swaminathan. 2024. Realizing the AMD Exascale Heterogeneous Processor Vision: Industry Product. In *Proceedings of the 51st International Symposium on Computer Architecture*. IEEE Press, Piscataway, NJ, USA, 876–889.

[159] Alan Smith, Gabriel H. Loh, John Wuu, Samuel Naffziger, Tyrone Huang, Hugh McIntyre, Ramon Mangaser, Wonjun Jung, and Raja Swaminathan. 2024. AMD Instinct MI300X Accelerator: Packaging and Architecture Co-Optimization. In *Proceedings of the IEEE Symposium on VLSI Technology and Circuits*. IEEE Press, Piscataway, NJ, USA, 1–2.

[160] Synopsys. 2025. What are Chiplets? https://www.synopsys.com/glossary/what-are-chiplets.html. (2025).

[161] Suyash Tandon, Leopold Grinberg, Gheorghe-Teodor Bercea, Carlo Bertolli, Mark Olesen, Simone Bna, and Nicholas Malaya. 2024. Porting HPC Applications to AMD Instinct™ MI300A using Unified Memory and OpenMP®. In *Proceedings of the ISC High Performance*. Association for Computing Machinery, New York, NY, USA, 1–9.

[162] Leonel Tedesco, Aline Mello, Leonardo Giacomet, Ney Calazans, and Fernando Moraes. 2006. Application Driven Traffic Modeling for NoCs. In *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design*. Association for Computing Machinery, New York, NY, USA, 62–67. https://doi.org/10.1145/1150343.1150364

[163] Andrew Tee, Nicholas Curtis, Noah Wolfe, and Daniel Wong. 2025. The MALL is Open: Exploring Shared Caches and Latency in AMD CDNA™ 3 GPUs. In *Proceedings of the SC25-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA, 1110–1116.

[164] Tenstorrent. 2025. Tenstorrent Blackhole Accelerator. https://tenstorrent.com/en/hardware/blackhole. (2025).

[165] vLLM. 2025. Optimization and Tuning. https://docs.vllm.ai/en/v0.7.3/performance/optimization.html. (2025).

[166] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. 2024. Understanding the Host Network. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery, New York, NY, USA, 581–594.

[167] Jacob Wahlgren, Gabin Schieffer, Ruimin Shi, Edgar A León, Roger Pearce, Maya Gokhale, and Ivy Peng. 2025. Dissecting CPU-GPU Unified Physical Memory on AMD MI300A APUs. In *Proceedings of the IEEE International Symposium on Workload Characterization*. IEEE Press, Piscataway, NJ, USA, 368–380.

[168] Chendong Wang, Joontaek Oh, and Ming Liu. 2026. Co-Designing Traffic Control with NVMe-oF for Disaggregated Storage: A Comparative Study of Switched and Switchless SAN Architectures. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'26)*. 2651–2669.

[169] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU Microarchitecture Through Microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. IEEE Press, Piscataway, NJ, USA, 235–246. https://doi.org/10.1109/ISPASS.2010.5452013

[170] Bowen Wu, Wei Cui, Carlo Curino, Matteo Interlandi, and Rathijit Sen. 2025. Terabyte-Scale Analytics in the Blink of an Eye. *Proc. VLDB Endow.* 19, 2 (Oct. 2025), 141–155. https://doi.org/10.14778/3773749.3773754

[171] Yu Xia, Vishnu Ramadas, Matthew Poremba, and Matthew D. Sinclair. 2025. Narrowing the GAP: Enhancing gem5's GPU Memory Bandwidth Accuracy. In *6th gem5 Users' Workshop*. 2.

[172] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO Queues Are All You Need for Cache Eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 130–149. https://doi.org/10.1145/3600006.3613147

[173] Ahmad Yasin. 2014. A Top-Down Method for Performance Analysis and Counters Architecture. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Press, Piscataway, NJ, USA, 35–44.

[174] Minlan Yu. 2019. Network Telemetry: Towards a Top-Down Approach. *ACM SIGCOMM Computer Communication Review* 49, 1 (2019), 11–17. https://doi.org/10.1145/3314212.3314215

[175] Jie Zhang, Hongjing Huang, Xuzheng Chen, Xiang Li, Jieru Zhao, Ming Liu, and Zeke Wang. 2025. RpcNIC: Enabling Efficient Datacenter RPC Offloading on PCIe-attached SmartNICs. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA'25)*. 1379–1394.

[176] Chenxingyu Zhao, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2025. White-Boxing RDMA with Packet-Granular Software Control. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 427–449.

[177] Chenxingyu Zhao, Hongtao Zhang, Jaehong Min, Shengkai Lin, Wei Zhang, Kaiyuan Zhang, Ming Liu, and Arvind Krishnamurthy. 2026. SG-IOV: Socket-Granular I/O Virtualization for SmartNIC-Based Container Networks. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1727–1748.

[178] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M. Dai, Quoc V. Le, and James Laudon. 2022. Mixture-of-Experts with Expert Choice Routing. *Advances in Neural Information Processing Systems* 35 (2022), 7103–7114.

APPENDICES

Appendices are supporting material that has not been peer-reviewed.

A Acronyms and Terminologies

Table 3 defines the important chiplet and memory acronyms used in the paper.

B Limitations of Existing GPU Profilers

Vendor-provided profilers, such as AMD's ROCProfiler [20, 21] and NVIDIA's Nsight [127, 130, 131], expose and leverage hardware performance counters to provide cycle-accurate kernel performance metrics across architectural components. However, most provided counters focus on compute units and L1/L2 caches within an XCD, and largely treat the communication fabric beyond the L2 as a flat memory interface [88, 126]. As a result, ACN behavior can only be inferred through partial and indirect information. For example, ROCProfiler exposes 259 counters for AMD MI300X/MI350X GPUs [7, 8]; 236 are XCD-related, while only 28/24 are related to IOD/HBM [17]. Table 4 shows a detailed list of these counters. However, even these IOD/HBM counters mostly focus on L2-to-Infinity Fabric (`TCC_EA0`) information such as request volume and credit stalls of XCD-side L2 misses and evictions. NVIDIA's Nsight for B100/B200 [88, 126] (partial list of their relevant counters outlined in Table 5) exhibits similar limitations. Thus, activity on both individual ACN links (§2.2) and non-XCD components such as the IOD NoC, LLC, and HBM-facing memory controllers, are collapsed into coarse signals.

Furthermore, adding additional counters would not completely remove current limitations due to the difficulty of *mapping low-level events to link-level models* of COM-IO, IO-IO, and IO-MEM behavior, and *composing these link models into path-level analysis across multiple chiplets*. Conversely, `PingPoint` provides this missing information through a system model (§4.2) that enables converting controlled path-level probes (§3.2) into link-level diagnosis (§4.7), thereby making ACN bottlenecks interpretable and actionable.

C Reverse-engineered CU Affinity Mask

Vendor-provided software stacks from AMD and NVIDIA expose Compute Unit (CU) masks for configuring the GPU's CU usage [22, 26]. As shown in Figure 15, we reverse-engineer how this bit layout maps to the architectural hierarchy on AMD MI300X/MI350X. They have 8 XCDs, 4 Shader Engines (SEs) per XCD, and 9/8 CUs per SE.

For the MI300X, the CU mask is a three-dimensional structure, `cu_mask[c][s][x]`, where c is the CU index within an SE, s is the SE index within an XCD, and x is the XCD index. Physically, the three dimensions are packed into word, byte, and bit positions, respectively. The 32-bit word `cu_mask[c]` encodes CU index c across all 4 SEs and 8 XCDs. Within each word, the four bytes correspond to SE3, SE2, SE1, and SE0 from MSB to LSB, and each byte contains 8 bits corresponding to XCD0–XCD7. Thus, each bit conceptually controls whether CU c in SE s of XCD x is enabled.

For the MI350X, the CU mask has different three-dimensional structure, `cu_mask[w][s][x]`, where w is the mask-word index, s is the SE index within an XCD, and x is the XCD index. The difference is that w does not map directly to the CU index for all SEs. Instead, bit `cu_mask[w][s][x]` controls CU w in even-indexed SEs, i.e., SE0 and SE2, but CU $w+1$ in odd-indexed SEs, i.e., SE1 and SE3, on XCD x. Thus, each bit must be interpreted using an SE-dependent translation from mask-word index w to the enabled CU position.

Both CU masks have a practical caveat. As noted in Algorithm 6, although each byte nominally contains one bit per XCD, we only observe reliable control at byte granularity: each byte is effectively usable as either `0x00` or `0xff`, not as an arbitrary 8-bit subset of XCDs, forcing CU selection uniformly across all XCDs. Consequently, thread-to-CU pinning necessitates an additional XCD-filtering step using a prelude kernel in Algorithm 1 from §3.2.

D Hardware Counters for XCD stalls

We use ROCProfiler [20, 21] to examine hardware counters that capture stalls and credit-based flow control at three levels of an XCD: (a) `TA_ADDR_STALLED_BY_TC_CYCLES` (CU-level), which counts cycles the CU's address path stalls by the cache/memory pipeline; (b) `TCP_PENDING_STALL_CYCLES` (L1/L2-level), which counts cycles the L1D cache stalls while waiting for data from L2 cache; and (c) `TCC_EA0_{RD|WR}REQ_DRAM_CREDIT_STALL` (L2-Fabric-level), which counts cycles in which the read/write interface to Efficiency Arbiter (EA) stalls due to a lack of credits. If the L2-Fabric credit stall

Acronym	Definition
ACN	Accelerator Chiplet Network
XCD	Accelerator Complex Die; Compute chiplet in MI300X/MI350X
IOD	Input/Output Die; IO chiplet in MI300X/MI350X
HBM	High-bandwidth memory
CC	Chiplet Cluster; a tile composed of physically proximate chiplets
COM-IO	Compute-to-IO link between an XCD and an IOD
IO-IO	IO-to-IO link between two IODs
IO-MEM	IO-to-memory link between an IOD and an HBM stack
IO-EXT	IO-to-other interconnect link between an IOD and host/other device

Table 3: Acronyms.

Counter	Description
`TCC_MISS`	#L2 misses
`TCC_WRITEBACK`	#L2 writebacks (dirty, uncached writes, atomics)
`TCC_NORMAL_WRITEBACK`	#L2 writebacks (non-writeback reqs)
`TCC_ALL_TC_OP_WB_WRITEBACK`	#L2 writebacks (`TC_OP` writeback reqs)
`TCC_NORMAL_EVICT`	#L2 evictions (non-invalidate or probe reqs)
`TCC_ALL_TC_OP_INV_EVICT`	#L2 evictions (`TC_OP` invalidate reqs)
`TCC_EA0_WRREQ`	#Transactions to `TC_EA_wrreq` IF (non-probes)
`TCC_EA0_WRREQ_64B`	#64B transactions (write, CMPSWAP) to `TC_EA_wrreq` IF
`TCC_EA0_WR_UNCACHED_32B`	#32B/64B write or atomic to TC_EA_wrreq IF due to uncached traffic
`TCC_EA0_WRREQ_STALL`	#Cycles a write req is stalled
`TCC_EA0_WRREQ_DRAM_CREDIT_STALL`	#Cycles EA write req is stalled due to HBM credits
`TCC_TOO_MANY_EA_WRREQS_STALL`	#Cycles L2 cache is unable to send EA write req due to it reaching maximum pending EA write reqs
`TCC_EA0_WRREQ_LEVEL`	#Accumulated EA write reqs in flight
`TCC_EA0_ATOMIC`	#Atomic reqs to `TC_EA_wrreq` IF
`TCC_EA0_ATOMIC_LEVEL`	#Accumulated EA atomic reqs in flight
`TCC_EA0_RDREQ`	#Read reqs to EA
`TCC_EA0_RDREQ_32B`	#32B read reqs to EA
`TCC_EA0_RD_UNCACHED_32B`	#32B EA reads due to uncached traffic (64B counts as 2)
`TCC_EA0_RDREQ_DRAM_CREDIT_STALL`	#Cycles EA read req is stalled due to HBM credits
`TCC_EA0_RDREQ_LEVEL`	#Accumulated EA read reqs in flight
`TCC_EA0_RDREQ_DRAM`	#EA read reqs to HBM
`TCC_EA0_WRREQ_DRAM`	#EA write reqs to HBM
`GRBM_TC_BUSY`	#Cycles any of the texture cache blocks are busy
`GRBM_EA_BUSY`	#Cycles the EA block is busy

Table 4: ROCProfiler's IOD/HBM-related hardware counters. IF=Interface. TCC=L2. EA=Efficiency Arbiter.

Counter family	Description	Example counters
`ctc__rx_*`	Blackwell C2C receive traffic	`ctc__{rx|tx}_bytes`
`ctc__tx_*`	Blackwell C2C transmit traffic	`ctc__{rx|tx}_bytes_data_user`
`lts__*ltcfabric*`	Cross-partition L2 traffic	`lts__t_sectors_srcunit_ltcfabric`
`dram__*`	HBM endpoint traffic	`dram__.*bytes.*`, `dram__.*sectors.*`

Table 5: Nsight's chiplet-to-chiplet (C2C) and HBM-related hardware counters.

Algorithm 6 CU Masking (MI300X).

```
1: fn mask_cu (cu_mask, enabled_cu_ids):
2:   for i = 0 to CUS_PER_XCD - 1:
3:     if (enabled_cu_ids >> i) & 1:
4:       // Each byte enables a CU across all 8 XCDs in MI300X due to caveat
5:       cu_idx = i / SES_PER_XCD; se_idx = i % SES_PER_XCD
6:       cu_mask[cu_idx] |= (0xffu << (SES_PER_XCD-1-se_idx) * BITS_PER_BYTE)
```

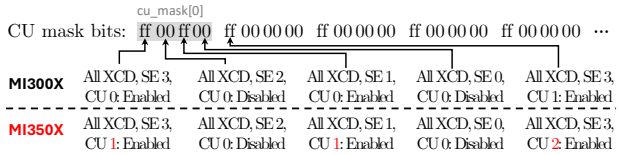


Figure 15: Reverse-engineered MI300X/MI350X CU mask.

		Specification
Accelerator Overall	Clock/Power	2200 MHz/1000W (TDP)
	Microarch.	CDNA4
	Chiplet Config.	8 XCDs (3nm) and 2 IODs (6nm)
	Throughput	2307 TFLOPS (FP16), 4614 TFLOPS (FP8)
Compute Chiplet (XCD)	CU #	32 (4 for yield management)
	L1D	32KB per-CU, 128B CL, 1 CH, per-CH 64W4S
	L2	4MB per-XCD, 128B CL, 16 CHs, per-CH 16W128S
IO Chiplet (IOD)	XCD-IOD	3D, Copper-to-Copper, 9.0 TB/s per CC
	IOD-IOD	2.5D, USR-PHY, 5.5 TB/s per CC
	LLC	128MB per-IOD, 64B CL, 64 CHs, per-CH 16W2048S
	IOD-HBM	2.5D, HBM-PHY, 4.0 TB/s per CC
HBM Module	Size/Bus/Rate	288GB HBM3E, 8192 bit width, 8.0 Gbps
	Paging	2MB page under 8KB interleaved

Table 6: Hardware specification of the AMD MI350X. CL=Cacheline. CH=Channel. xWyS=x-way y sets. CC=Chiplet Cluster.

counters are unavailable [46], we use `TCC_EA0_{RD|WR}REQ_LEVEL`, which accumulates cycles with in-flight read/write requests at EA.

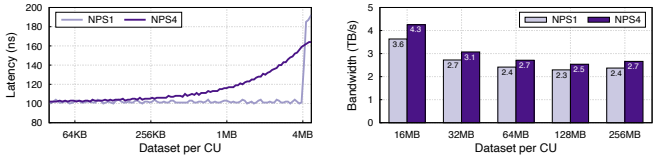
We further detail Figure 4d from §3.4, which shows the normalized values of these three counters as COM-IO load increases. The plotted series, labeled (a) CU backpressure, (b) L1-L2 stalls, and (c) Queue@COM-IO IF, correspond to counters (a), (b), and (c) above. Starting at the COM-IO queueing onset identified earlier in §3.4, 43.7% load (≈500 GB/s), the L1-L2 and L2-Fabric stall counters increase by 4.89× and 4.38×, respectively, as their Miss Status Holding Registers (MSHRs) become saturated, while the CU stall counter remains nearly constant. After COM-IO load exceeds 74.0% (≈850 GB/s), backpressure reaches the CU, causing it to stall and stop issuing new instructions. This increases the CU stall counter by 2.27×. These XCD-internal mechanisms are triggered by queueing at the COM-IO link and bound its HoL-induced overheads (§3.4).

E MI350X Specification

Table 6 summarizes the AMD MI350X [8] hardware specification. MI350X is based on the CDNA4 architecture [10] and contains 8 XCDs, 2 IODs, and 8 HBM stacks. Each IOD, together with its attached 4 XCDs and 4 HBM stacks, forms a Chiplet Cluster (CC), yielding two CCs in total (§2.1). For memory allocation, MI350X alternates 8KB chunks across HBM stacks attached to the two IODs, with the interleaving order flipping at 2MB page granularity.

F GPU Partitioning

Multi-chiplet GPUs like MI300X support compute and memory partitioning modes [14]. For compute, (a) Single Partition X-celerator (SPX) treats the GPU as a single device, while (b) Core Partitioned X-celerator (CPX) views each XCD as an individual logical GPU. For memory, (a) Unified Memory (NPS1) exposes all 8 HBM stacks as one unified memory pool, whereas (b) Partitioned Memory (NPS4)



(a) Latency. **(b) Bandwidth.**

Figure 16: MI300X performance under GPU partitioning.

makes only the pair of HBM stacks within a CC directly visible to the logical devices in that CC. Switching modes requires a GPU driver reset for hardware reconfiguration, utilizing Single Root I/O Virtualization (SR-IOV) [16] to apply the new crossbar configuration for CPX and modifying address interleaving for NPS. The default configuration is SPX+NPS1. However, since CPX+NPS4 makes all memory requests *local* (§3.2), we also investigate whether the channel partitioning is further adjusted with respect to GPU partitioning.

Figure 16a compares the latency of NPS4 against NPS1 within the L2 range (32KB to 4MB). Surprisingly, we observe a smooth increasing trend in NPS4, unlike the stepped jump in NPS1. This behavior implies a potential shift in cache management strategies due to the elimination of Non-Uniform Cache Access (NUCA) penalties; specifically, the L2 cache appears to move away from strict working set residency guarantees toward a more probabilistic or holistic approach since all evictions in NPS4 have uniform costs of spilling to local LLC. With NPS4's modified address interleaving, each XCD's L2 presumably caches data predominantly from its local memory partition.

Moreover, we observe that NPS4 dedicates a larger portion of the COM-IO channels to the local CC. Figure 16b presents the bandwidth achieved by an XCD when comparing NPS4 and NPS1, both with CPX, using dataset sizes (16MB to 256MB) larger than L2 cache (4MB). NPS4 achieves an average of 13.0% higher bandwidth than NPS1. This result contrasts with NPS1 mode, in which ordinary contiguous memory requests are uniformly distributed across CCs and consistently outperform explicit local-CC access by 12.8%. This suggests that the strict hardware isolation in NPS4 drives a reconfiguration of the COM-IO channels to optimize local data paths. In turn, for workloads that fit within the partitioned memory capacity, this provides NPS4 net latency and bandwidth advantages, similar to those reported in prior studies [16, 163].