USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Building A CSFQ-Inspired Transport for Switched CXL Memory Pooling

Zerui Guo, *University of Wisconsin-Madison;* Emily Shriver, *Intel;*
Ming Liu, *University of Wisconsin-Madison*

https://www.usenix.org/conference/nsdi26/presentation/guo-zerui

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Building A CSFQ-Inspired Transport for Switched CXL Memory Pooling

Zerui Guo[1], Emily Shriver[2], and Ming Liu[1]

[1]University of Wisconsin-Madison [2]Intel

Abstract

Emerging switched CXL memory pooling systems, albeit promising, suffer from significant performance interference due to the shared but performance-uncontrolled data path among concurrent memory streams between a host core and a remote DIMM. We systematically characterize a memory pooling appliance based on the XConn's Apollo CXL switch, and identify three issues: intra-host contention, in-fabric congestion, and unmanaged host-remote DIMM interaction.

This paper presents a new transport layer, **MemChannel**, which provides the `mchannel` abstraction to manage end-to-end fabric bandwidth among competing memory flows and enable application-specific traffic for switched CXL memory pooling. Under the hood, our key idea is to build a *Sender-Driven Fabric-Informed* transport protocol–inspired by Core-Stateless Fair Queueing (CSFQ)–that admits just the right amount of CXL requests to each `mchannel` based on the estimated $Core \leftrightarrow DIMM_{CXL}$ bandwidth availability. To grapple with the ramifications of CXL-induced idiosyncrasies, MemChannel introduces a couple of techniques: time-based rate control, host-side admission control, cross-host bookkeeping, new congestion signals, rate estimation based on the fluid model, and delay-based link capacity adjustment. We build MemChannel from scratch and support unmodified applications. Our evaluations over switched memory pooling demonstrate the effectiveness of MemChannel from performance isolation, scalability, and multi-tenancy perspectives.

1 Introduction

Load-Store interconnects–such as Compute Express Link (CXL)–and the resulting memory pooling have gained significant traction recently. They transparently extend a commodity server's memory capacity, enable elastic memory resource sharing, and allow applications to access remote memory using load/store instructions. Such composable memory technology holds great potential to remedy the DRAM scaling issue, ameliorate hardware resource utilization, and boost the cost efficiency of rack/cluster-scale computing. The last few years have seen several industrial prototypes [6, 11, 15, 16, 18] being developed, evaluated, and sampled.

However, memory pooling, especially under switched deployment, suffers from significant performance interference. Unlike local memory connected via dedicated memory buses, *the data path between a host core and a remote CXL DIMM under switched pooling is shared among concurrent intra-/inter-host memory streams without explicit performance control.* We systematically characterize the problem using the XConn's Apollo CXL switch and Titan evaluation platform and identify three issues (§2.3). First, at the server host, taking the Intel EMR (Emerald Rapids) processor as an example, access contentions happen at both (a) the CHA (caching and home agent) and M2PCIe modules of the host uncore; and (b) the virtual lane inside a CXL host adapter, increasing the CXL memory access latency by up to 13.4× with 82.6% bandwidth drops. Second, within the switch fabric, the link-layer credit-based flow control is workload-agnostic, causing head-of-link blocking and unfair bandwidth partition at the egress port. As such, when a cacheline-sized memory stream interleaves with a 4KB-sized one, it experiences 6.9×/3.8× higher read/write latencies, with 97.8% link bandwidth being taken (even though both have the same amount of outstanding bytes). Third, the hardware prefetching implicitly induces many more loads into the target adapter under regular access patterns and predictable data locality, which is completely transparent to the CXL fabric. We observed that while the switch downstream port is not congested, the target adapter sees twice as much traffic, causing dramatic queue build-up.

Therefore, the switched CXL memory pooling lacks a transport layer that can effectively manage end-to-end fabric bandwidth among competing memory flows and enable application-specific traffic control. However, realizing this is non-trivial. First, CXL packets (FLITs) traverse through the CPU pipeline, system bus, switching fabric, and remote DIMM, whose transmissions are implicit. Specifically, a load/store instruction is issued depending on data locality and the execution condition of micro-architectural components, and its response directly resumes the stalled processor execution without signaling. Second, CXL adapters and switches

employ an ultra-fast data plane with limited in-network computing capabilities, whose hardware architecture is opaque, eluding any active traffic control mechanisms. Third, monitoring the data transmission performance requires us to bridge the gap between high-level memory streams and low-level hop-by-hop link-layer credits. The problem is further exacerbated by the inherent nature of the lossless network and the massive amount of application-induced memory requests.

We design and implement a new transport layer (dubbed **MemChannel**) atop today's `CXL.mem` protocol stack, inspired by a seminal technique – Core-Stateless Fair Queueing (CSFQ) [75]. It achieves max-min bandwidth in the Internet for competing flows without maintaining per-flow states. We find that CSFQ is promising to tackle the above challenges in our context because (a) it is highly scalable, where core switches only maintain a few aggregated variables (e.g., arrival rate, accepted rate, and fair share rate) with fixed computing complexity; (b) edge-driven, requiring minimal in-network support; (c) lightweight on the data plane, whose traffic manipulation primitives (e.g., statistic bookkeeping, labeling, and packet dropping) are compute-efficient.

MemChannel comprises three pieces: (a) host programming interfaces that center around the `mchannel` system object with APIs to support unmodified applications; (b) the host runtime that runs the protocol stack, interacts with CXL fabric components, and performs rate control; and (c) in-fabric system extensions at the switch, adapter, and CXL DIMM, participating in the protocol execution. MemChannel probes the end-to-end bandwidth capability at runtime on the data plane, introduces new fabric congestion signals, and computes the per-`mchannel` transmission rate. *Key to MemChannel is a Sender-Driven Fabric-Informed transport protocol that admits just the right amount of CXL requests to each* `mchannel` *based on the estimated Core* $\leftrightarrow$ $DIMM_{CXL}$ *bandwidth availability.* Specifically, we first apply the fluid model to the entire CXL fabric, figure out how to tailor CSFQ to our context, and derive the theoretical bound. Next, to handle the implicit transmission issue, MemChannel applies the time-based rate control that translates the end-to-end bandwidth usage and availability to the available running time. To avoid in-network data-plane operations, we push rate calculation to the edge, reserve a designated remote memory region for bookkeeping cross-host statistics, and translate in-network packet dropping to endhost admission control. We then follow the fluid model to determine the per-`mchannel` access speed and fair share rate and use a delay-based approach to probe the link bandwidth capacity. Further, we extend the MemChannel to support weight and multi-layer CXL switching.

We prototyped the MemChannel, ported several applications [26, 27, 44, 49] atop, and performed end-to-end evaluation over a real switched memory pooling setup. Our evaluations show that MemChannel fully uses the underlying link bandwidth, mitigates intra-host and cross-host performance interference, provides CSFQ-like fairness, adapts to the fabric bandwidth availability, and scales with the application demands. Applications running over `mchannels` receive efficient multi-tenancy and achieve 2–3.5× improvements.

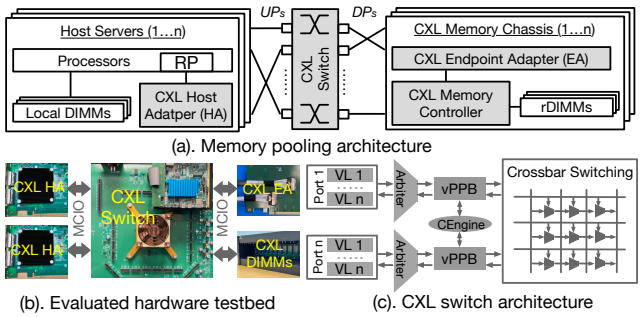


Figure 1: The architecture and hardware testbed of a switched CXL memory pool. (c) shows the architecture of a CXL switch. VL=Virtual Lane. vPPB=Virtual PCI-to-PCI Bridge.

2 Understanding Switched CXL MEM Pooling

2.1 Switched CXL Memory Pooling

CXL [3] is an emerging high-speed cluster interconnect built atop the physical layer of PCIe Express [10]. It provides a load/store interface for CPU, memory, and device communications. The interconnect uses the Flex Bus I/O architecture [4], and is organized into physical, data link, and transaction layers. CXL supports three types of channels: `CXL.cache`, `CXL.mem`, and `CXL.io`. Memory pooling is constructed using `CXL.mem` and the corresponding memory expanders (Type-3 device). Since CXL 2.0, CXL fabrics allow resource pooling via single-level and multi-tiered switching [4]. Figure 1 depicts the system architecture of a switched CXL memory pool.

- **CXL Host Adapter.** It exposes access root ports (RPs) and cooperates with the host memory subsystem. The adapter converts load/store instructions into fabric-routable FLITs and transmits them over the wire. Upon responses, it parses the packets, obtains fetched read data or write completions, and delivers them to the processor pipeline.
- **CXL Switch.** A CXL switch consists of upstream ports (UPs) for host adapters' connectivity, downstream ports (DPs) for remote memory, and internal forwarding tables for traffic orchestration. Upon initialization, it discovers all connected components, configures the routing structure, and fills table entries based on the topology. The switch carries CXL transactions on the data plane and employs some scheduling policies. CXL supports a hybrid of Port-Based Routing (PBR) and Hierarchy-Based Routing (HBR).
- **CXL Endpoint Adapter.** It stays close to target memory devices, operating as a responder for remote memory. The adapter processes CXL protocols and converts between FLITs and memory commands. It also performs integrity checking, request steering (when multiple logic devices are used), and transmission speed synchronization.

- **CXL Attached Memory.** The remote DIMMs (rDIMMs) are housed in a standalone chassis or appliance, including an SoC backplane, memory controllers, and a power supply. Each module uses a DDR-compatible PHY interface, supporting different kinds of memory media.

Evaluation Target. Several memory appliances [6, 11, 14, 17, 18] have been developed, sampled, and tested in the past few years. We take the recent Titan platform [18] from the XConn Tech as the evaluation target (Figure 1-b). It consists of (1) ASIC-based host and endpoint adapters; (2) an XConn B2 Apollo CXL switch with 14 upstream and downstream ports, 2 CEM [1] slots, and 128 CXL2.0 lanes, supporting $\times 2$, $\times 4$, and $\times 8$ bifurcations; (3) a memory chassis using an MCIO/EDSFF backplane that holds up to 12 vendor-agnostic CXL DIMMs [8, 12] under the compact EDSFF E1.S, E3.S, and E3.L form factors [7]; and (4) an MX8 board operating as the management host and fabric manager.

Software Stack. A switched CXL memory pooling platform has three software components: (1) a fabric manager running on a dedicated host, which interacts with each fabric hardware, discovers the system topology, enumerates hosts and target memory nodes, and monitors their living status; (2) the memory controller firmware for the remote DIMMs, which configures the communication ID, initializes its address space, and processes traversed data; (3) a device driver that makes remote memory as host-managed device memory (HDM) and exposes it as a CPUless NUMA node. The host OS fabricates an attached CXL DIMM as a memory node, manages it with object or tiered memory systems [34, 46, 52, 61, 74, 78, 82, 87], and runs applications atop. After a CXL DIMM is mapped to the host memory subsystem, applications can access it via load/store instructions. Take the Intel x86 processor as an example. A memory read, missed from the last-level cache (LLC), fetches the corresponding cache line from the CXL memory. Memory writes hit the store buffer first, then are issued to the remote memory under eviction. Hardware/Software prefetch and cache coherence-induced events (like read-for-ownership) also cause data loads [47, 50].

2.2 CXL Switching Architecture

A CXL switch provides high-performance and lossless connectivity between upstream and downstream ports (Figure 1-c). Data is transmitted at the FLIT granularity, a fixed-sized amount of data traversed over the underlying link, such as 68B and 256B. An incoming FLIT arrives at one virtual lane (VL) and is then forwarded to an arbiter (multiplexer). Next, the FLIT is delivered to a vPPB (virtual PCI-to-PCI bridge) module, acting as a logical connection point to accommodate different types of CXL devices and facilitate the routing. vPPBs can also be organized hierarchically to construct multiple VCS (virtual CXL Switch) within a physical switch. There is a credit engine (CEngine) interacting with all vPPBs and running (1) a hop-by-hop credit-based flow control [42, 43]; (2) a credit update protocol (like N23) for reliable device-switch

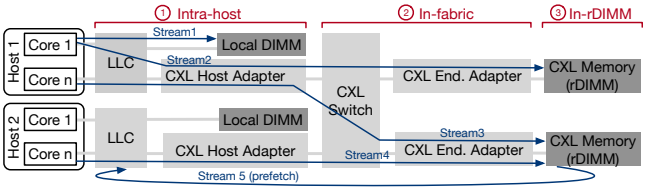


Figure 2: Different access contention points along the CXL data path under switched CXL memory pooling.

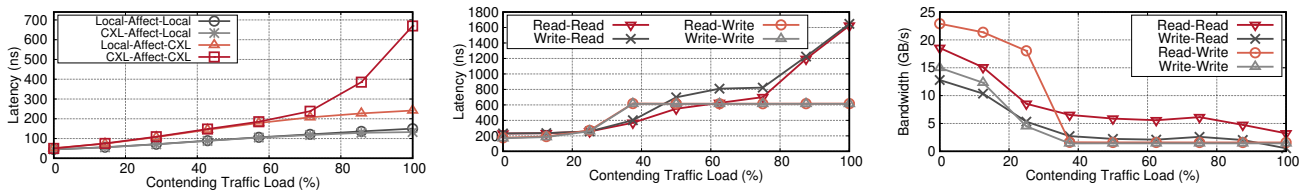
and switch-switch coordination [42]; and (3) an adaptive and statistical credit allocation scheme to maximize bandwidth usage [43]. Last, most of today's CXL switches (like XConn Apollo XC50256 [18] and Omega Fabric [6]) employ a crossbar topology, where the routing logic is determined by the fabric manager during the memory pool initialization. Note that (1) the CXL switch is nearly bufferless but still incurs transmission stall when credit starvation happens; (2) there is little programmability on the switch data-plane, unlike Ethernet ones offering some in-network primitives in the traffic manager; (3) host and endpoint adapters usually adopt a similar switching architecture, just with much fewer ports.

2.3 Characterizing Switched CXL Memory Pooling

Researchers have studied extensively about a direct-attached CXL Type-3 memory expander [34, 45, 47, 50, 55, 71, 76]. This section characterizes the performance of switched CXL memory pooling with a focus on analyzing the performance interference. Figure 2 presents how intra- and inter-host memory requests would interleave at different locations along the CXL data path in a switched memory pool. We configure experiments to locate the contention points, quantify how performance isolation is affected, and explore the root causes.

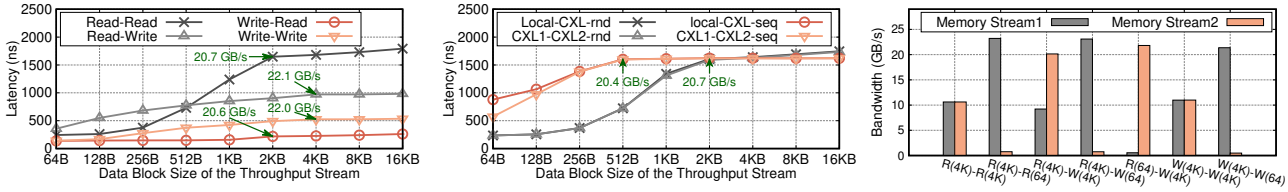
Experimental Methodology. We use 2U Intel Emerald Rapids servers as hosts, where each contains two Xeon Gold 6530 CPUs, 1536GB DDR5 memory, and two Samsung 9MA3 960GB NVMe drives. Each processor has 32 cores running at 2.1GHz and 160MB LLC. All the hosts run Ubuntu 24.04. Each server is equipped with one CXL host adapter enclosing two $\times 8$ CXL ports. Using Intel MLC [5], we observe that the server achieves 220.5ns and 47.2GB/s when accessing the switched CXL memory pool. We developed a micro-benchmark (similar to [9, 51, 76]) that can launch any number of memory streams between host cores and local/CXL DIMMs with different access patterns. It supports various configurations, such as enabling/disabling prefetching, issuing temporal read/write and non-temporal write, injecting NOP instructions, and performing random/sequential/strided accesses. The benchmark uses data and instruction synchronization barriers to complete pending reads and writes in the CPU pipeline before starting the test. When running, it spawns several pinned threads, initiates memory streams, issues reads/writes, and collects execution statistics.

Issue #1: Intra-host Contention. Host uncore and CXL host adapter are the first two contending locations. In an Intel X86 processor, the uncore–enclosing LLC, CHA (Caching and



(a) Uncore contention. (b) Host adapter contention (latency). (c) Host adapter contention (bandwidth).

Figure 3: (a) runs background threads on one socket, issues sequential accesses, and maxes out its memory bandwidth. We report random access latency. The "Local-Affect-Local" and "CXL-Affect-Local" cases are added for comparisons. (b)/(c) present the host contention issue. X–Y shows that the victim stream (X) is impacted by co-located streams (Y). All experiments use one ×8 CXL port.



(a) Host-rDIMM interference. (b) rDIMM-rDIMM interference. (c) Bandwidth partition.

Figure 4: In (a), X–Y describes the case that a latency stream X is co-located with throughput streams (Y), where we report the average latency of X. (b) presents the latency results of four data movements. We add the local-rDIMM scenario for comparison. rnd=Random. seq=Sequential. In (c), we run two throughput streams each over four cores, issuing 64B/4KB read/write requests.

Home Agent), and FlexBus, connected via a mesh NoC [66]– is shared by local and CXL memory requests, e.g., stream1 and stream2 in Figure 2. We configure two kinds of threads: *T1*, whose working set fits into the LLC, accessing local or CXL memory and measuring latency; and *T2*, which generates competing traffic with gradually increased loads by adding the number of threads without overwhelming the host adapter's bandwidth capacity. As shown in Figure 3-a, when the FlexBus is congested, accessing the CXL memory experiences up to 13.4× latency increases, rising from 50.1ns to 669.4ns. Using the recent PathFinder tool [47], we find out that this is because significant queueing happens at the M2PCIe ingress, which slows down CXL request transmission. When contentions occur at the LLC/CHA under mixed local and CXL traffic, we observe that both the LLC miss rate and the total amount of cache snooping requests greatly increase, yielding at most 2.9× CXL access latency increases.

Next, we analyze the host adapter contention by issuing cache-bypassed CXL requests (stream2 and stream3 in Figure 2). We set up two memory streams: *T1* accesses CXL memory with one outstanding load/store (i.e., victim memory stream); *T2* reads/writes to remote memory, with the number of outstanding accesses gradually increased. As shown in Figures 3-b/c, T1 starts to experience a performance drop even when the total traffic only reaches 30%, far below the adapter's bandwidth capacity. By discussing with the device vendor, we learn that this is because the host adapter schedules CXL FLITs over several VLs of the virtual lane in a best-effort fashion, completely agnostic of how requests are issued from host cores, which causes skewed cross-lane FLIT distribution. When the adapter is nearly saturated, we observe that in the Read(T1)-Read(T2) case, the victim stream's bandwidth is reduced by 82.6%, with latencies increased from 230.2ns to 1624.2ns. The other three cases are similar.

Issue #2: In-fabric Congestion. The CXL switch is the second interference point (stream3 and stream4 in Figure 2). Since the CXL switching relies on a hop-by-hop credit-based flow control [38, 42, 43], the amount of credits that a downstream entity receives is proportional to its bandwidth usage and the contention degree at the upstream device. Hence, when a small-sized request stream competes with a large one that causes credit over-subscription, it would experience stalls due to delayed credit replenishment. To quantify this, we configure two kinds of random access memory streams: the latency stream that issues one memory read/write at a time, and the throughput one, which generates larger data blocks, yielding multiple outstanding memory transactions.

We intermix small and large memory requests and deploy memory streams in two directions: host→rDIMM and rDIMM→rDIMM. Regarding the host→rDIMM case, as shown in Figure 4-a, when the fabric is under-utilized, a 64B read mixed with 128B reads and writes causes a 7.5% and 54.2% latency increase, respectively. In the case of a 64B write contending with throughput streams, when the data block size is 256B, its latency is increased by 6.2% and 98.6%. This is due to the head-of-line (HoL) blocking effect at the upstream port. When bandwidth over-subscription happens, one would experience considerable performance drops. For example, when interleaved with a 4KB memory request, a 64B read/write experiences a 6.9×/3.8× slowdown. The rDIMM→rDIMM scenario presents similar behaviors (Figure 4-b), where the 64B CXL memory access latency starts to rise when the data block size is above 256B.

Next, we deploy two competing throughput streams, vary their request configurations, and explore how bandwidth is allocated in different cases. As shown in Figure 4-c, we find that the bandwidth a memory stream receives is mainly proportional to its data block size, regardless of request type. For

Stride Len.	RD Lat.	RD Th.	WR Lat.	WR Th.
1	10.9 ns	21.9 GB/s	10.8 ns	22.0 GB/s
2	15.6 ns	15.3 GB/s	13.2 ns	18.0 GB/s
4	18.7 ns	12.7 GB/s	13.3 ns	17.9 GB/s
6	18.2 ns	13.1 GB/s	12.3 ns	19.4 GB/s
8	18.6 ns	12.8 GB/s	12.4 ns	19.2 GB/s

Table 1: CXL memory performance under a fixed stride size X. The distance between two consecutive requests is $X \times 64$ bytes.

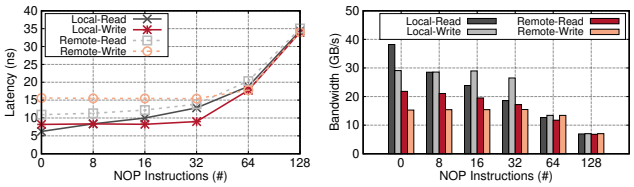
example, when two 4KB read(write) streams interleave, each would achieve 10.6(11.0) GB/s. However, if a 4KB stream is interleaved with a 64B one, it can take nearly 97.8% of the total bandwidth. These results make sense for two reasons. First, memory requests from one memory stream are synchronously served one by one. The number of outstanding cacheline-sized reads/writes depends on the data block size. Second, when multiple concurrent streams compete for the fabric, the CXL switch/adapter mainly performs round-robin scheduling to decide the next issuing request. Thus, the stream with more pending transactions has more opportunities to be scheduled, yielding higher bandwidth.

Issue #3: Unmanaged Host-rDIMM Interaction. The endpoint adapter provisions enough bandwidth for CXL DIMMs. Since it connects directly to the switch, access congestion is first resolved at the downstream port, which should leave the adapter and rDIMM free from contending traffic. However, we find out this is sometimes not the case. Even though we control the aggregated traffic across all hosts to be lower than the link/port bandwidth, contention still happens. This is because host prefetching, under regular access patterns, implicitly generates more memory requests, which are transparent to the CXL fabric but cause bandwidth over-subscription.

We configure our microbenchmark to issue strided memory reads and writes. Table 1 presents the latency and throughput as the stride length increases from 1 to 8. When the stride length is 1, compared with the random access, remote memory read and write achieve 10.9ns and 10.8ns (hitting in the L1 and L2 cache), sustaining at 21.9GB/s and 22.0GB/s bandwidth. Apparently, the access locality results in multiple cacheline reads and writes issuing concurrently to hide latency. When the stride length increases to 8, the read and write bandwidth drops to 12.8 GB/s and 19.2 GB/s. Further, we perturb the CPU prefetching effect by issuing NOP instructions between two consecutive memory accesses and measure its performance impact. We find that both local and CXL memory performance drops significantly when inserting more NOP instructions (Figure 5). When prefetching helps little, a remote memory read/write matches the local one. Therefore, host prefetching causes unmanaged host-rDIMM interaction, which would introduce much more traffic than expected and cause contention within the target memory pool.

2.4 Problem and Challenges

Our characterization study unearths and quantifies three issues (i.e., intra-host contention, in-fabric congestion, and host-rDIMM interaction) that cause performance interference in a switched CXL memory pool. *The fundamental problem is that the data path between a host core and a remote CXL DIMM is shared among concurrent memory requests without explicit performance control.* As such, any on-path entities, like the host memory subsystem, switch, and host/endpoint adapter, could become a communication choke point, which would break the target latency and bandwidth of a memory stream. This calls for a new transport layer for switched memory pooling that effectively allocates communication resources based on the application requirements. Realizing this incurs the following three challenges that have not been tackled before.



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

Figure 5: Performance when perturbing instruction prefetching.

- **#1: Implicit transmission.** Under switched memory pooling, a CXL request traverses across the CPU pipeline, cache hierarchy, system bus, adapters, switching fabric, and remote DIMM. Whether and when to issue a CXL load/store transaction is affected by several factors, such as data locality (L1D, L2, and LLC), queueing occupancy at different micro-architectural components (such as store buffer and line fill buffer), and hardware prefetching. Similarly, a data response returns to the host memory subsystem and directly resumes the processor execution. Therefore, tracking CXL requests, measuring the per-flow performance, and controlling the transmission rate become non-trivial.
- **#2: Hardware non-programmability and opaqueness.** To sustain at sub-microsecond latency and hundreds of Gigabytes per second of bandwidth (CXL 3.2 [4]), CXL adapters and switches streamline their data plane and offer nearly no reconfigurability and telemetry capability, making our conventional in-network transport design [21,22,39,48,92] impractical. Further, there is no consensus system model or open implementation standard (like PISA [29]).
- **#3: Fine-grained cross-layer traffic monitoring.** Measuring end-to-end bandwidth availability is already challenging enough since this requires tracking how many link-layer credits are allocated across the entire data path. However, in a switched pool, the transport layer has to further break down this information at a finer granularity to individual host cores/threads. As described above, the round-trip delay is impractical to obtain, given the integrated pipeline. The inherent nature of lossless fabric (back-pressure and credit starvation) further exacerbates the issue.

3 MemChannel: a Designated Memory Lane

This section introduces the MemChannel transport, including semantics, APIs, and system model. Our system design goals:

- **High Utilization.** MemChannel should fully use the bandwidth at any vantage point and link. It should be able to ramp up the CXL memory access speed for other needed applications when some bandwidth becomes available.
- **Efficient Multi-tenancy.** MemChannel should mitigate the performance interference across the end-to-end CXL data path (§2.3). It should achieve low (tail) latency and approximate Max-Min fairness under contention.
- **Low overheads.** MemChannel should have little impact on the de facto CXL load and store access. It should incur few memory footprints and consume tolerable host CPU cycles when holding existing applications.

3.1 Overview

MemChannel is a transport layer that orchestrates remote memory accesses between host cores and CXL DIMMs in a switched CXL memory pool. It offers a performance-controlled data pipe (`mchannel`), provisions proper communication resources based on workload demands and remote DIMM's free bandwidth, and mitigates interference from contending memory streams. A `mchannel`, established between a host core (C_i) and a CXL DIMM ($rDIMM_j$), is associated with a dedicated application process. One can also group multiple `mchannels` for each application (discussed in §4.4).

Key to MemChannel is a *Sender-Driven Fabric-Informed* transport protocol that admits just enough CXL requests to the switched memory pool based on the estimated $C_i \leftrightarrow rDIMM_j$ bandwidth availability. Essentially, it probes the end-to-end bandwidth capability at runtime on the data plane, introduces new fabric congestion signals, computes the per-`mchannel` transmission rate and admits an adequate amount of CXL load/store commands to the memory pool. MemChannel encompasses three major pieces. One is the programming interface that allows porting unmodified applications. The second is the host runtime that runs the protocol stack, interacts with other CXL fabric components, and performs rate control. The third is in-fabric system extensions at the switch, adapter, and CXL DIMM, which participate in the protocol processing.

3.2 The `mchannel` Semantics

MemChannel exposes the `mchannel` as a system object from the host OS perspective, consisting of the following attributes:

- *Host/Core ID and Registered Memory Node*, specifying the requester (source) and responder (destination) of a MemChannel. Akin to commodity systems [6,11,17], we support DAX memory, mounted as a CPUless NUMA node.
- *CXL Data Path*, describing the end-to-end data path between C_i and $rDIMM_j$, provided by the fabric manager, including host adapter, switches, and endpoint adapter.
- *Channel Representation*, like `struct sock`, instantiated by our host runtime, serving as an intermediate connection point between applications and the transport protocol.

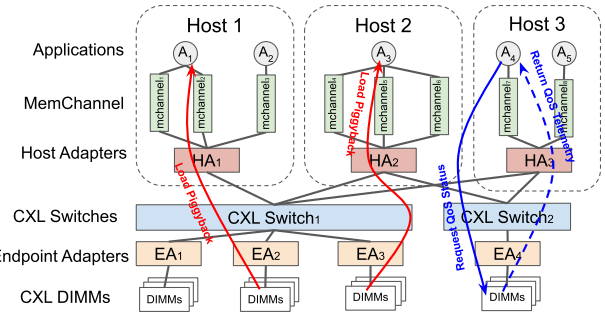


Figure 6: The system model of MemChannel.

- *Performance Attributes*, like bandwidth envelope and interference degree. Developers specify them upon registration or at runtime. When omitted, the `mchannel` is labeled as best-effort and uses what bandwidth is available.

3.3 The `mchannel` Interface

MemChannel provides a user-space CLI (Command-Line Interface), similar to `numactl`, which onboards the unmodified applications and controls their execution behaviors to achieve the performance target. MemChannel runtime offers four lightweight APIs to control the remote memory access:

- `mchannel_init` creates `mchannels` and takes user-specified configurations, such as which applications to run, host core mappings, remote memory nodes, and performance attributes. It then identifies the CXL data path for each `mchannel`, allocates local memory for `mchannel` metadata, maps shared CXL memory for inter-host coordination, and launches the applications as subprocesses.
- `mchannel_open` opens a `mchannel` to a rDIMM in the pool, invoked when a new application thread is launched. It allocates the per-thread `mchannel` states and resources, and then arms a POSIX timer with a scheduling window T_W. When the timer expires, it invokes `mchannel_sched` as the signal handler and readjusts the bandwidth allocation. It also uses the `m_bind` system call to ensure the application places its data on the remote DIMM specified by the `mchannel`, sets the thread's core affinity based on the host core mappings, and opens as well as maps the performance counters for CXL bandwidth monitoring.
- `mchannel_close` closes the `mchannel` to the remote DIMM, cleans up allocated resources, and unsets the timer signal handler. It is called when the application thread exits.
- `mchannel_sched` enforces application-specific bandwidth allocation via our transport protocol (§4), which monitors the remote pool access bandwidth and then controls application execution to meet the application performance needs. For bandwidth measurement, we use off-core response (OCR) architectural event counters to track three types of CXL requests [47, 50]: demand reads, reads for ownership, and hardware prefetches. The results are collected via `rdpmc` commands and aggregated as the remote bandwidth. For bandwidth control, during a scheduling win-

dow T_W, we adjust the POSIX timer expiration such that the application thread runs for T_R and sleeps for $T_W - T_R$.

MemChannel relies on code injection techniques [2], such as `LD_PRELOAD` and `ptrace`, to support unmodified applications. Specifically, MemChannel injects a dynamic link library (DLL) into applications when launching them. Inside the DLL constructor and `pthread_create` interposed by DLL, MemChannel calls `mchannel_open`. The `mchannel` interfaces incur low overheads, because `mchannel_sched` is invoked only once per scheduling window T_W and halts application execution only once during that window. In addition, we utilize lockless data structures to synchronize between `mchannels` and the `rdpmc` command to read performance counters inside `mchannel_sched` instead of system calls.

3.4 System Model

We divide our running environment into six layers (Figure 6): applications, **mchannel**s, host adapters, CXL switches, endpoint adapters, and CXL DIMMs. Our MemChannel layer carries application-induced CXL memory requests and delivers them to the host adapter. The CXL FLIT traverses the underlying fabric as it is. Following the recent CXL specification [4], the memory device continuously reports service loads, reflecting its internal queueing status. There are two ways: (a) *piggyback in memory responses* (red line in Figure 6), where the memory transaction response reserves 2 bits to indicate four levels of loads: light load, optimal load, moderate overload, and server overload; (b) *QoS telemetry messages* (blue line in Figure 6), i.e., the host issues explicit QoS status queries through the CXL Component Command Interface (CCI), whereas the memory expander (rDIMM) returns the average load of the last profiling epoch.

4 An CSFQ-Inspired Transport

This section describes our core transport protocol and shows how we address the above challenges (§2.4).

4.1 Why Core-Stateless Fair Queueing

Core-Stateless Fair Queueing (CSFQ) [75], a seminal work in computer networks, introduced a fair queueing algorithm for the Internet several decades ago. It achieves max-min fair bandwidth allocation among competing flows without maintaining per-flow states. The algorithm divides the rate control logic between the networking edge and core, where (1) edges calculate the flow arrival rate and label the rates into packets; (2) the core estimates the fair rate iteratively and probabilistically drops packets to achieve the fair share rate. CSFQ is (a) scalable, where core routers only maintain a few aggregated variables (e.g., arrival rate, accepted rate, and fair share rate) with fixed computing complexity; (b) edge-driven, requiring minimal in-network support; (c) lightweight on the data plane, the traffic manipulation primitives (e.g., statistic bookkeeping, labeling, and packet dropping) are compute-efficient, making it promising to address our problem.

However, naively applying CSFQ and its successor HCSFQ [84] is still not feasible for three reasons. First, these approaches take packet drops as a congestion signal, but CXL fabric is a lossless network. Second, core switches require the packet-dropping primitive for traffic regulation, which is not supported on CXL switches. Third, CSFQ needs accurate flow rate and link bandwidth availability estimations, but CXL switching lacks such capabilities. Therefore, akin to CSFQ and HCSFQ, we first apply the fluid model to the CXL switching, derive the theoretical guarantees when achieving the max-min fairness (§4.2), and redesign the transport algorithm (§4.3). To handle the implicit transmission issue, MemChannel applies the time-based rate control that translates the end-to-end bandwidth usage and availability to the available running time. To avoid in-network data-plane operations, we push rate calculation to the edge, reserve a designated remote memory region for bookkeeping cross-host statistics, and translate in-network packet dropping to endhost admission control. We then follow the fluid model to determine the per-`mchannel` access speed and fair share rate and use a delay-based approach to probe the link bandwidth capacity.

4.2 Applying Fluid Model to CXL Switching

We apply the fluid model to formalize switched CXL memory pooling systems. The system is viewed as a directed acyclic graph $G(V,E)$, where V represents system components (e.g., the ones in Figure 6), and E represents the traffic flows between components. We assume the flows are continuous memory streams from a host core to a remote CXL DIMM. We consider a single layer of switching in the following discussion, but our analysis applies to multiple layers. A flow starts from a memory channel, goes through a bridge and a switch, and reaches the CXL DIMM. Therefore, graph $G(V,E)$ is a complete multipartite graph, where the vertex set is partitioned into four disjoint subsets $\{V_C, V_H, V_S, V_E\}$ for channels, host adapters, switches, and endpoint adapters, respectively.

Each CXL hardware component has a maximum link transmission capacity C_v. Let α_v be the fair share rate a node v allocated to its successors. We know the application access demand D_c for channel $v_c \in V_C$ via our mechanism (§ 4.3). For the edge between a channel and a host adapter, the aggregated demand is the channel's demand, e.g., $D_{ch} = D_c; v_h \in V_H$. The demand for other edges can be calculated recursively.

$$D_{vw} = \sum_{e_{uv} \in p; \forall p \in P_{vw}} \min(\alpha_v, D_{uv}) \tag{1}$$

$p = (e_{ch}, e_{hs}, e_{se}); e_{ch}, e_{hs}, e_{se} \in E$ describes a flow path. $P_{vw} = \{p | e_{vw} \in p\}$ is all the flows going through edge e_{vw}. Therefore, under contention, where the aggregated memory demand $D_v = \sum_{e_{uv} \in E} D_{uv} > C_v$, one can achieve the max-min fair bandwidth allocation among competing flows when α_v is the unique solution to the following equation.

$$C_v = \sum_{e_{uv} \in E} min(\alpha_v, D_{uv}) \tag{2}$$

Algorithm 1 Transport Algorithm.

```
1:  procedure MCHANNEL_SCHED(mchannel c)
2:      n = read_perf_counters()          ▷ CXL memory access count
3:      c.demand = estimate_rate(c.demand, n, c.T_W);      ▷ Eq. 3
4:      c.rate = estimate_rate(c.rate, n, c.T_R);          ▷ Eq. 3
5:      for On_Path_Device dev in c.path do
6:          α = estimate_α(dev);
7:          c.α = min(α, c.α);
8:      c.T_R = estimate_T_R(c.demand, c.α, T_W);          ▷ Eq. 5
9:      sleep(c.T_W − c.T_R)              ▷ Yield to other mchannels
10: procedure ESTIMATE_α(On_Path_Device dev)
11:     if cur_time ≥ start_time + K then
12:         F, D = 0, 0;                  ▷ local rate F, local demand D
13:         for mchannel c in dev.mchannels do
14:             F += c.rate;
15:             D = max(D, c.demand);
16:         dev.F, dev.D = agg_rate(F, D);    ▷ via shared memory
17:         if localhost is dev.host then
18:             dev.α = min(dev.α × dev.C/dev.F, dev.D);      ▷ Eq. 4
19:         start_time = cur_time;
20:     return dev.α;
21: procedure ADJUST_CAPACITY(On_Path_Device dev)
22:     if dev.F < dev.C and dev.l > δ then
23:         dev.C = dev.C × (1 − (dev.l−δ)/(2(1−δ)))  ▷ Multiplicative Decrease
24:     else if dev.l ≤ δ then
25:         dev.C = dev.C + λ × (δ − dev.l) × K ▷ Additive Increase
26:     dev.C = max(dev.C, dev.max_capacity)
```

Without contention (e.g., $D_v \leq C_v$), all demands can be satisfied by setting α_v to the max, i.e., $\alpha_v = \max_{e_{uv} \in E} D_{uv}$. There are two key differences when applying the fluid model in our case compared with CSFQ. First, memory access rates remain the same along the flow path. As a result, we compute memory access demand instead of using flow arrival rates for edges and vertices. Second, we use the link transmission capacity at each node rather than the default maximum bandwidth specified in the hardware documentation. This is because the number of FLITs a CXL port of an adapter or switch can handle and transmit is not static but depends on the queuing conditions and internal execution status.

4.3 The Transport Algorithm

Access Rate and Demand Estimation. We measure the number of remote cacheline accesses n during a scheduling window T_W by reading performance counters (§3.3). The measured CXL memory access rate for *mchannel* c is $r'_c = \frac{nS}{T_W}$, where S denotes the cacheline size. r'_c reflects the amount of CXL bandwidth consumed by *mchannel* c over T_W. Similarly, the measured memory demand is $D'_c = \frac{nS}{T_R}$, where T_R is the application's running time within T_W. D'_c indicates the memory bandwidth that *mchannel* c would generate if it were not throttled by the transport algorithm. Measuring application demand in this manner is possible because the application runs at full speed without interruptions or slowdowns during T_R, as our algorithm ensures that CXL components are not oversubscribed. Following the definition of CSFQ, we then estimate the memory access rate as the following equation, where K is a constant for adjusting the sampling window.

$$r_c^{new} = (1 - e^{-T_W/K}) r'_c + e^{-T_W/K} r_c^{old} \tag{3}$$

We estimate the access demand D_c by substituting r'_c with D'_c. Hence, hosts update r_c and D_c of each channel at the beginning of the scheduling function (ALG1 L2–4). Since CXL fabric is lossless, the aggregated memory access rate can be calculated directly, rather than using Eq. 3 as in CSFQ.

Fair Share Rate Estimation. MemChannel then calculates the fair share rate α on the host and issues the right amount of load/store instructions to avoid congestion. However, using Eq. 2 to compute α directly requires a host to know the aggregated memory demands on edges (D_{uv}) across all on-path hardware it traverses. Unlike Ethernet, where the arrival rate (i.e., demand) can be directly measured at the switch port, here it must be obtained through recursive calculations (Eq. 1). Specifically, the memory demand of a CXL endpoint adapter depends on incoming memory streams from all corresponding upstream CXL switch ports, which in turn depend on incoming streams from upstream host adapters. This recursive dependency can easily lead to an accumulation of error margins. Furthermore, both demands and fair share rates must be shared across hosts, making the algorithm difficult to scale. Instead, we estimate the fair share rate based on the aggregated remote memory access rate F_v on the host for hardware node v, defined as the number of load/store commands transmitted through v. This approach is straightforward to compute and can be calibrated using hardware bandwidth telemetry.

We next describe how MemChannel performs fair-share rate estimation (Algorithm1 L10–20). We define the aggregated memory access rate as $F_v = \sum_{e_{uv} \in E} f_{uv}$, where f_{uv} is the flow on edge e_{uv} and $f_{cb} = r_c$ for $\forall v_c \in V_C, \forall v_h \in V_H$. Since the access rate is the same within a flow p, the aggregated rate F_v is expressed as $\sum_{e_{cb} \in p, \forall p \in P_v} r_c$, where P_v denotes all flows passing through node v. Under congestion (Eq. 2), we estimate α iteratively using the current congestion condition $\frac{C_v}{F_v}$. If the estimated fair share rate exceeds the largest local demand $\max_{c \in V_L} D_c$, there is no congestion within the local host, and thus α is set to the maximum demand, i.e., $\max_{e_{uv} \in E} D_{uv}$.

$$\alpha_{new} = \min(\alpha_{old} \frac{C_v}{F_v}, \max_{e_{cb} \in p, \forall p \in P_v} D_c) \tag{4}$$

At the end of the sampling window K, hosts calculate the fair share rates of all hardware components used by their `mchannels`. The aggregated memory access rate F_v is the only variable shared across hosts to compute the fair share rates for shared CXL hardware (e.g., switches and expanders). To disseminate this information, each host writes the local sum of the memory access rates for each hardware component to a small, dedicated shared memory address. The aggregated rate F_v can then be obtained by summing these local rates. The `mchannel` fair share rate α_c is the minimum value along

path p (i.e., $\alpha_c = \min_{e_{uv} \in p} \alpha_v$), which is then translated into the running time to control application execution.

Time-based Rate Control. MemChannel controls the FLIT transmission rate by adjusting the application thread running time T_R within each scheduling window T_W (§3.3). After running for T_R, the application thread yields to another and remains in the waiting queue until the next scheduling window begins. A short scheduling window T_W enables fine-grained control over application execution but incurs high context-switching overhead due to frequent timer-expiration interrupts. Conversely, a larger T_W reduces scheduling overhead but causes tail latency increases because requests are less tightly coordinated. We choose $T_W = 100\mu$s in our prototype, which strikes a balance between these two trade-offs.

Similar to TCP, a flow cannot transmit bytes when there is no free space in the congestion window. By temporarily stalling application execution, we can regulate the memory access rate to prevent congestion. To avoid temporal contention, application threads and their `mchannels` run asynchronously, even though they may have the same scheduling window. MemChannel estimates the CXL memory access size B during one scheduling window as $D_c T_R$. To ensure fair sharing of CXL memory, the max-min fair remote memory size B_{fair} that an application can access during T_W is $\min(\alpha_c, D_c) \times T_W$. The goal of our transport is to achieve max-min fair resource allocation without wasting CXL memory bandwidth. Therefore, we set $B = B_{fair}$. Rearranging gives a fair running time during one scheduling window:

$$T_R = \frac{\min(\alpha_c, D_c) T_W}{D_c} \tag{5}$$

New Congestion Signals. In Ethernet, congestion can be readily inferred from explicit signals like packet loss. However, such signals are largely unavailable in the CXL fabric. Instead, the CXL 3.2 specification [4] mandates that memory expanders include a 2-bit device-internal load indication in memory responses (S2M), which encodes four levels of congestion typically derived from the expander's internal queue occupancy. In addition, the specification strongly recommends that memory devices expose a QoS telemetry mechanism that periodically reports (1) egress-port backpressure, indicating that one or more upstream queues between the expander and the host are saturated, and (2) temporary throughput reductions during DRAM refresh operations.

Congestion at the adapter can be queried at the host since the device is attached locally. For CXL DIMMs, we leverage the device-load field to capture both the device's internal queueing pressure and temporary throughput reductions, while disabling the egress-port backpressure signal. The backpressure information is conveyed through telemetry messages and is treated as a congestion signal for CXL switches, since switches expose no explicit congestion notifications.

Transmission Capacity Estimation. Due to the internal complexity of CXL hardware components, the transmission capacity is difficult to measure and may change dynamically. Inspired by the delay-based congestion control [24, 30, 65], we maintain load factors l based on the congestion signals.

$$l_{new} = (1-g) \times l_{old} + g \times L \tag{6}$$

L represents the percentage of responses marked as having moderate or severe overload in the device load field. For telemetry cases, the backpressure average percentage in the QoS response is directly used as L. g is a configurable parameter. Using them, we then divide the operation region into the following categories and apply different scaling strategies:

- **Congestion Avoidance**, occurring when fair share rate is increasing and l is relatively large. This means that some remote memory accesses have experienced high delays. We perform a multiplicative decrease and scale the reducing factor based on the load factor l (ALG1 L22–23).
- **Congestion Free**, where $l < \delta$. This indicates that the rDIMM is able to deliver predefined bandwidth. Depending on how much idleness the channel has observed, our algorithm will add $\lambda \times (\delta - h.l) \times K$ (ALG1 L24–25).

4.4 Algorithm Extensions

Weight Support. MemChannel can be extended to support `mchannels` with different weights w_c, meaning that all congestion channels will receive a fair share rate of $w_c \alpha_c$. The only modification required is to update Eq. 5 accordingly.

$$T_R = \frac{\min(w_c \alpha_c, D_c) T_W}{D_c} \tag{7}$$

HCSFQ Support. Our algorithm achieves the fair share among `mchannels`, equivalent to a single-layer HCSFQ. However, in practice, an application may use multiple `mchannels` (two-layer HCSFQ), and a tenant may run multiple applications (three-layer HCSFQ). Supporting HCSFQ is straightforward in MemChannel. In §4.3, we know (a) the transmission capacity of the hardware, which corresponds to the root node of the tree, and (b) memory access demand and access rate of the `mchannel`, which correspond to the leaf nodes of the tree. The aggregated demand/rate of the parent node is the sum of its children's (i.e., $D_v = \sum D_u$). By applying Eq. 4 recursively from the root to the leaves, we can determine the fair share rate, α, for each layer. We then translate the α of the last layer into the `mchannel` running time using Eq. 5. Note that the tree is used for all hardware components (adapters, switches, and DIMMs) as shown in our system model (§3.4).

4.5 Bound Analysis

We present the following theorem to show that our algorithm provides performance bounds for applications. Consider a memory channel with a fixed fair share rate α_c and weight w_c during a sampling window K. In the ideal case, the memory channel should not issue load/store traffic exceeding $w_c \alpha_c K$. We can prove that no matter how the application tries to game the system, our mechanism ensures that the bytes of

load/store traffic from a memory channel are no more than: $w_c\alpha_c K(1+2\frac{T_W}{K})$, where T_W is the scheduling window.

Proof. Consider any interval $K=[t',t'')$, and let $h=\lfloor\frac{K}{T_W}\rfloor$. In an interval, there will be h full scheduling windows, at most one scheduling window before t', and at most one scheduling window after t''. Thus, the maximum number of bytes transmitted during K is $\sum_{i=0}^{h+1} B_i$. Based on Eq. 7, we have:

$$B_i \le B_{fair} = \min(w_c\alpha_c, D_c)T_W \le w_c\alpha_c T_W. \tag{8}$$

Therefore, we can derive the bound as follows:

$$\begin{aligned}\sum_{i=0}^{h+1} B_i &\le (h+2)w_c\alpha_c T_W \le \left(\frac{K}{T_W}+2\right)w_c\alpha_c T_W \\ &= w_c\alpha_c K\left(1+2\frac{T_W}{K}\right).\end{aligned} \tag{9}$$

This bound indicates that the application can issue at most $2\frac{T_W}{K}$ fractional more requests in the short run, ensuring that our algorithm achieves a max-min fair bandwidth allocation.

5 Evaluation

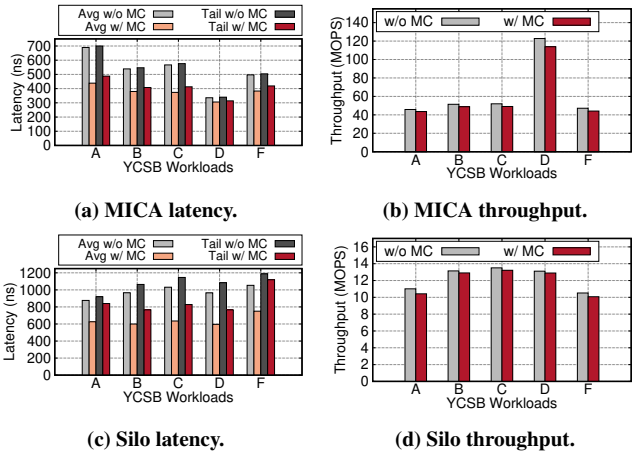
5.1 Experimental Methodology

Testbeds and Workloads. We use the same experimental setup as §2.3. Our evaluations use a diverse set of memory-intensive workloads as prior studies [53, 79, 82, 91]. *(a). In-memory databases.* We run a hashtable-based in-memory key-value store, MICA [49], configured with 8-byte keys and 100-byte values, and we focus on its in-memory components: circular logs, lossy concurrent hash indexes, and bulk chaining. We also evaluate a B-tree–based database, Silo [77], designed for low-latency transaction processing, with a 1KB value size. The client traffic is generated via YCSB [33]. *(b). Graph analytics.* We deploy the GAP [27] benchmark suite, including several graph kernels: Breadth-First Search (BFS), Single-Source Shortest Paths (SSSP), PageRank (PR), Connected Components (CC), and Betweeness Centrality (BC), which use a Kronecker graph and a uniform random graph. *(c). High-performance computing (HPC).* We also use SPEC CPU 2017 [13] and PARSEC [28], including scientific and numerical applications that stress memory access.

Performance Metrics. We report application throughput, latency, and performance slowdown. MemChannel obtains per-*mchannel* bandwidth using performance counters (§ 3.3) and aggregates them to derive both application-level and host-level memory bandwidth. We measure CXL memory access latency via CHA queues, i.e., dividing occupancy by the number of CXL requests, following the Colloid's approach [79].

5.2 Application Performance over MemChannel

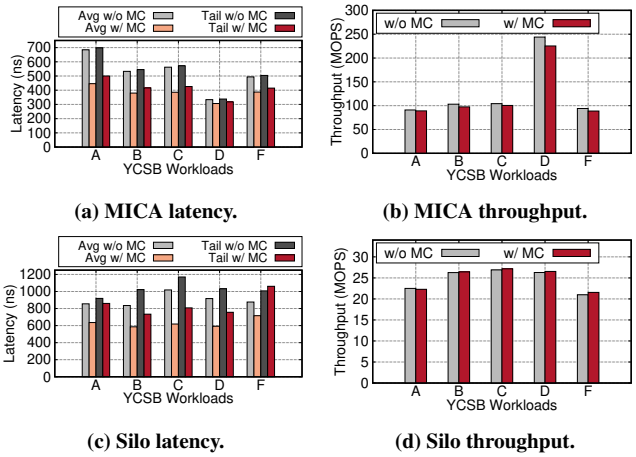
We first examine how much performance MemChannel delivers to applications when accessing the switched CXL memory pool. In this experiment, we use MICA and Silo and report the throughput and latency. With one ×8 CXL port, as shown in Figure 7-a/c, applications show nearly no throughput degradation, and MemChannel can provide 19.6 GB/s



(a) MICA latency. (b) MICA throughput.

(c) Silo latency. (d) Silo throughput.

Figure 7: Throughput and latency of MICA and Silo over one ×8 adapter to the memory pool, comparing between w/ and w/o MemChannel scenarios. We use 32 threads in this experiment.



(a) MICA latency. (b) MICA throughput.

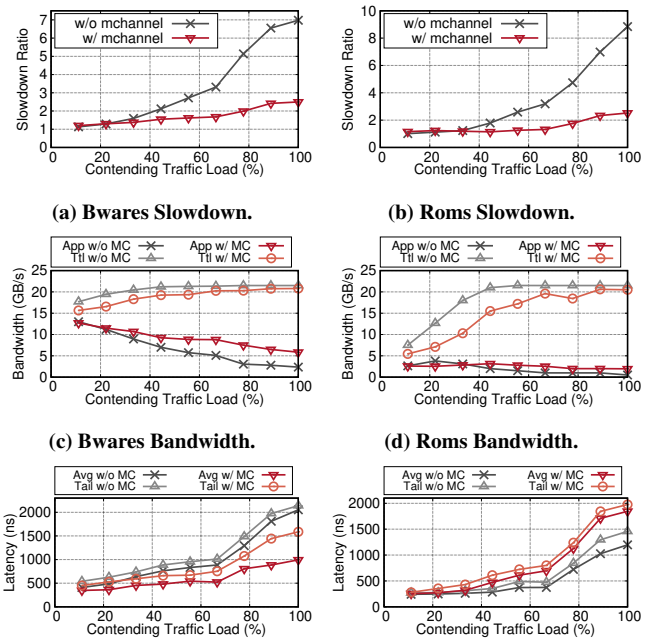
(c) Silo latency. (d) Silo throughput.

Figure 8: Throughput and latency of MICA and Silo over two ×8 adapters to the memory pool, comparing between w/ and w/o MemChannel scenarios. We use 64 threads in this experiment.

bandwidth to the remote memory pool. However, MemChannel reduces the in-fabric congestion and yields latency savings. For example, as shown in Figure 7-b/d, MemChannel reduces the average CXL memory access latency of Silo by 28.6%/38.0%/38.4%/38.3%/28.8%, and tail latency by 8.7%/28.0%/27.7%/29.3%/5.8% across five workloads compared with the cases when disabling MemChannel. MICA shows similar results. We then use two CXL adapters and run 64 threads. The average throughput over five applications reaches 225.3 MOPS under *mchannel* (Figure 8), achieving 41.0GB/s CXL bandwidth to the remote memory pool. In terms of latency, similarly, MemChannel mitigates the fabric contention and achieves 23.9% and 19.6% lower average and tail latencies compared to the cases without MemChannel.

5.3 Intra-Host Performance Isolation

We create intra-host contention by sharing the host adapter between an application and competing background traffic. We use Bwaves and Roms from CPU SPEC as our applications. As shown in Figure 9-a/b, applications running without



(a) Bwares Slowdown. (b) Roms Slowdown.

(c) Bwares Bandwidth. (d) Roms Bandwidth.

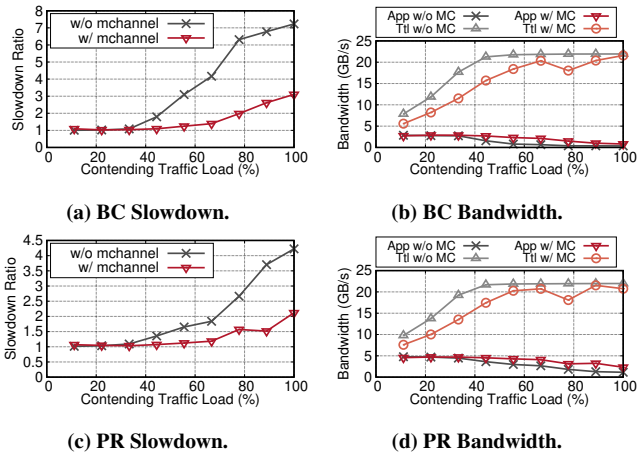
(e) Bwares Latency. (f) Roms Latency.

Figure 9: Application slowdown, CXL bandwidth, and latency when varying contending traffic under host adapter contention.

MemChannel experience 7.0× and 8.8× slowdowns as the contending traffic load increases to 100% for bwaves and roms, respectively. In contrast, applications running under MemChannel only see a 2.5× slowdown. This improvement occurs because MemChannel provides performance isolation among co-located channels through its transport algorithm (§ 4.3). To better understand this effect, we further measure the CXL access bandwidth of the applications and the total bandwidth, as shown in Figure 9-c/d. MemChannel allocates more bandwidth to applications based on max-min fairness. Under heavy contention (e.g., when the contending traffic load is $\geq$ 80%), MemChannel achieves an average of 96.1% bandwidth utilization across two applications, while reducing average latency by 51.6% and 35.1% and tail latency by 25.8% and 26.3% for Bwaves and Roms, as shown in Figure 9-e/f.

5.4 Inter-Host Performance Isolation

We then set up in-fabric congestion as §2.3 and evaluate how MemChannel mitigates the issue. We use two graph applications, i.e., Betweeness Centrality (BC) and PageRank (PR), in this experiment. As shown in Figure 10-a/c, applications running without MemChannel experience 7.2× and 4.2× slowdowns as the contending traffic load increases to 100% for BC and PR, respectively. In contrast, applications running under MemChannel only see 3.1× and 2.1× slowdown, respectively. The performance improvement comes from the fact that MemChannel allocates bandwidth across competing applications based on their performance requirements. Again, we measure the application CXL access bandwidth and the total bandwidth, as shown in Figure 10-b/d. MemChannel allocates bandwidth in a max-min fairness manner at the CXL



(a) BC Slowdown. (b) BC Bandwidth.

(c) PR Slowdown. (d) PR Bandwidth.

Figure 10: Application slowdown and CXL bandwidth when varying contending traffic under in-fabric congestion.

switching point. Under heavy contention (e.g., more than 80% traffic load), MemChannel achieves an average of 97.2% bandwidth utilization across two applications.

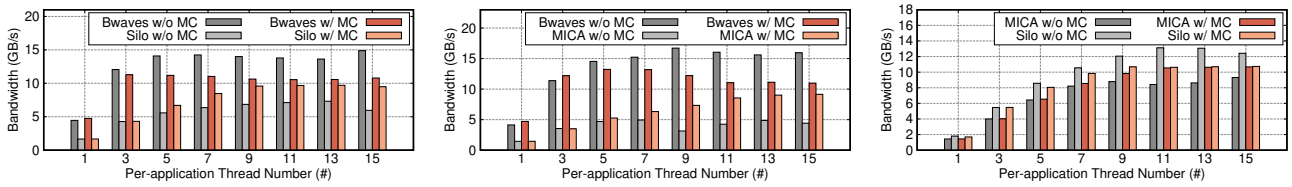
5.5 Fairness

We achieve a fair bandwidth share by calculating the `mchannel` rate, and MemChannel enforces it via the rate control algorithm. To measure effectiveness, we co-locate pairs of SPEC CPU, MICA, and Silo applications, gradually increasing the number of threads for both applications to induce contention in the CXL fabric, and report the resulting memory bandwidth for each. As shown in Figure 11, under light contention (e.g., one thread), MemChannel matches the baseline by allocating bandwidth according to application demand. Under heavy contention (e.g., 15 threads), compared with the case without MemChannel, MemChannel significantly reduces the bandwidth gap between two applications: from 8.9GB/s to 1.3GB/s for Bwaves+Silo, from 11.5GB/s to 1.8GB/s for Bwaves+MICA, and from 3.1GB/s to 0.1GB/s for MICA+Silo. Our transport controls the application's remote access bandwidth, ensuring a fair share of hardware resources.

We compare our runtime with TPP [61] under different background traffic generated by the same SPEC application as above (Figure 12). We run five types of YCSB workloads over MICA, with the local-to-remote memory capacity ratio set to eight. Without background traffic, TPP slightly outperforms our system by 1.2× in terms of throughput and reduces latency by 16% across all five workloads. This is because sufficient remote memory bandwidth is available to support page demotion and promotion. However, under moderate contention, our runtime matches the performance of TPP. Under heavy contention, our runtime achieves a 1.5× throughput improvement and reduces latency by 43%. This is due to the performance isolation capability provided by MemChannel.

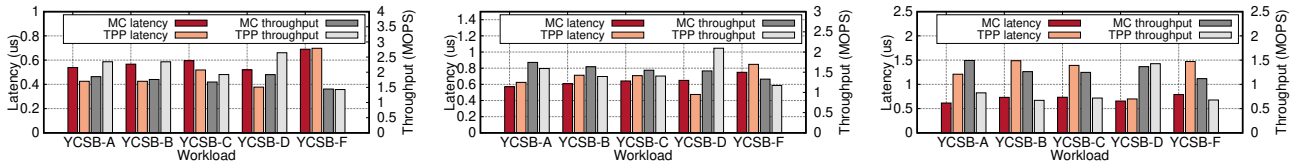
5.6 System Overheads

MemChannel incurs marginal overheads to the application. Fair-share rate calculation and assignment are performed only



(a) Bwaves+SILO. (b) Bwaves+MICA. (c) MICA+Silo.

Figure 11: Compare the CXL memory bandwidth when running two applications with and without MemChannel support on a host. We gradually increase the thread number for both applications to create contention the memory fabric.



(a) No Contention. (b) Moderate Contention. (c) Heavy Contention.

Figure 12: Latency and throughput for MICA running in MemChannel and TPP under five YCSB workloads. Y-1 axis is latency, and Y-2 axis is throughput. WS application is used to generate background traffic.

once per scheduling window for all on-path devices, based on shared aggregated application demands and rates. Each core needs to read only one single performance counter via *rdpmc*. The primary overhead introduced by MemChannel comes from POSIX timer timeout interruptions, used to suspend application execution. Based on our testing, this overhead amounts to 1.7% when the scheduling window is set to 100 us, enabling fine-grained control of CXL memory access.

5.7 Discussion

CXL ecosystems are under significant development. We believe that MemChannel can still be applied, but requires several changes. First, MemChannel currently advocates the device-internal load to indicate the traffic condition of each remote DIMM and uses end-to-end delay to estimate the fabric bandwidth capacity. With in-fabric explicit congestion notification, our transport can be further improved. Second, MemChannel is evaluated at rack scale rather than at cluster scale. It will be interesting to explore how MemChannel operates in a scale-out CXL fabric with PBR. Third, the recent CXL specification introduces the CXL bundled port to support devices with higher bandwidth requirements. Since it enables logical aggregation of multiple CXL ports, MemChannel should account for the intra-bundle traffic condition and develop a new device-load estimation technique.

6 Related Work

CXL Memory System. Researchers have explored how to build CXL-based remote memory systems extensively [34, 46, 47, 50, 55, 61, 76, 82, 91]. For example, Pond [46] analyzes the memory stranding issue within Azure and proposes a CXL-based small pooled system architecture. Melody [50] develops a framework to characterize CXL memory performance.

Load-Store Interconnect and Programmable Networks. Load-store fabrics [3, 19, 20, 38] have gained great interest recently to build scale-up networks for resource disaggregation [36, 40, 41, 57, 62–64, 83, 85, 88], but are mostly used in an exclusive deployment. As it moves to multi-tenant cases, similar transport techniques like MemChannel would be needed. We believe our past exploration on programmable networks [32, 35, 72, 73, 81, 86, 89, 90] and in-network computing [37, 54, 56, 58–60, 67–70] can shed great light on.

Congestion Control in Host Networks. Researchers have built models to understand congestion happening within a single host [23, 25, 31, 80]. HostCC [23] locates and identifies both host-internal and network fabric congestion, significantly reducing host queueing and packet loss while improving throughput and tail latency. Midhul Vuppalapati *et al.* develop a domain-by-domain credit-based flow control model to capture the subtle interplay that leads to latency inflation and host resource underutilization [80]. MemChannel coordinates tenants in a load–store memory pooling setting and enables fair sharing of memory bandwidth.

7 Conclusion

This paper presents MemChannel, a transport layer for switched CXL memory pooling. MemChannel introduces the `mchannel` abstraction for end-to-end fabric bandwidth management among competing memory streams and enables application-specific traffic control. Key to MemChannel is a Sender-Driven Fabric-Informed transport protocol–inspired by Core-Stateless Fair Queueing (CSFQ)–that admits just enough CXL requests to each `mchannel` based on the estimated $Core \leftrightarrow DIMM_{remote}$ bandwidth availability. Our evaluations demonstrate that MemChannel achieves high utilization, performance isolation, scalability, and multi-tenancy.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd, Paolo Costa, for their comments and feedback. This work is supported in part by NSF grants CNS-2106199, CNS-2212192, CAREER-2339755, and Intel faculty awards.

References

[1] An Introduction to Form Factors for PCI Express. https://pcisig.com/introduction-form-factors-pci-express, 2025.

[2] Code Injection. https://en.wikipedia.org/wiki/Code_injection, 2025.

[3] Compute Express Link (CXL). https://computeexpresslink.org, 2025.

[4] Compute Express Link (CXL) Specification. https://www.computeexpresslink.org/download-the-specification, 2025.

[5] Intel Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html, 2025.

[6] IntelliProp's Omega Fabric. https://www.intelliprop.com/products-page, 2025.

[7] JEDEC Memory Module Reference Base Standard – for Compute Express Link (CXL). https://www.jedec.org/standards-documents/docs/jesd317a, 2025.

[8] Micron Memory Expansion Using CXL. https://www.micron.com/products/memory/cxl-memory, 2025.

[9] pChase: A Pointer Chasing Benchmark. https://github.com/maleadt/pChase, 2025.

[10] PCI Express (Peripheral Component Interconnect Express). https://pcisig.com, 2025.

[11] Samsung CXL Memory Module - Box (CMM-B). https://semiconductor.samsung.com/news-events/tech-blog/cxl-memory-module-box-cmm-b/, 2025.

[12] SMART CXL Memory Modules. https://www.smartm.com/product/list/cxl-memory?utm_source=CXL&utm_medium=Website&utm_term=CXL-Website-TR&utm_content=CXL-Website-Link&utm_campaign=CXL-Website, 2025.

[13] SPEC CPU 2017. https://www.spec.org/cpu2017, 2025.

[14] The Falcon C5022. https://www.h3platform.com/product-detail/overview/35, 2025.

[15] The Intel® Agilex™ 7 FPGA I-Series Development Kit. https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/i-series/dev-agi027.html, 2025.

[16] The Leo CXL™ Memory Connectivity Platform. https://www.asteralabs.com/products/cxl-memory-platform/leo-cxl-memory-connectivity-platform/, 2025.

[17] UnifabriX MAX. https://www.unifabrix.com/technology, 2025.

[18] XConn Titan Evaluation Kit. https://www.xconn-tech.com/products, 2025.

[19] The NVIDIA NVLink. https://www.nvidia.com/en-us/data-center/nvlink/, 2026.

[20] Ultra Accelerator Link (UALink). https://www.ualinkconsortium.org, 2026.

[21] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. ABM: Active buffer management in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM'22)*, pages 36–52, 2022.

[22] Saksham Agarwal, Qizhe Cai, Rachit Agarwal, David Shmoys, and Amin Vahdat. Harmony: A congestion-free datacenter architecture. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, pages 329–343, 2024.

[23] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host congestion control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 275–287, 2023.

[24] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[25] Seunghyun An, Joontaek Oh, and Ming Liu. Server Chiplet Networking. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks (HotNets'25)*, page 289–299, 2025.

[26] Nikolas Askitis and Ranjan Sinha. HAT-trie: a cache-conscious trie-based data structure for strings. In *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*, pages 97–105, 2007.

[27] Scott Beamer, Krste Asanović, and David Patterson. The GAP Benchmark Suite, 2017.

[28] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[29] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, page 99–110, 2013.

[30] Lawrence S Brakmo, Sean W O'malley, and Larry L Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994.

[31] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 65–77, 2021.

[32] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP'24)*, pages 1–12, 2024.

[33] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[34] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC'22)*, pages 287–294, 2022.

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

[36] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, page 1012–1027, 2023.

[37] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. 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*, page 33–47, 2023.

[38] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. Understanding Routable PCIe Performance for Composable Infrastructures. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, 2024.

[39] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM'20)*, pages 422–434, 2020.

[40] Sheng Jiang and Ming Liu. Building an Elastic Block Storage over EBOFs Using Shadow Views. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*, pages 1137–1153, 2025.

[41] Yuyuan Kang and Ming Liu. Understanding and Profiling NVMe-over-TCP Using ntprof. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*, pages 1117–1136, 2025.

[42] HT Kung, Trevor Blackwell, and Alan Chapman. 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*, 1994.

[43] NT Kung and Robert Morris. Credit-based flow control for atm networks. *IEEE network*, 9(2):40–48, 1995.

[44] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and Efficient Work-Stealing for Weak Memory Models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[45] Philip Levis, Kun Lin, and Amy Tai. A Case Against CXL Memory Pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets'23)*, page 18–24, 2023.

[46] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*, page 574–587, 2023.

[47] Xiao Li, Zerui Guo, Yuebin Bai, Mehash Ketkar, Hugh Willkinson, and Ming Liu. Understanding and Profiling CXL.mem Using PathFinder. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM'25)*, 2025.

[48] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. HPCC: High precision congestion control. In *Proceedings of the ACM special interest group on data communication (SIGCOMM'19)*, pages 44–58. 2019.

[49] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast In-Memory Key-Value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[50] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S Berger, Marie Nguyen, Xun Jian, Sam H Noh, and Huaicheng Li. Systematic cxl memory characterization and performance analysis at scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1203–1217, 2025.

[51] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S Berger, Marie Nguyen, Xun Jian, Sam H Noh, and Huaicheng Li. Systematic cxl memory characterization and performance analysis at scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1203–1217, 2025.

[52] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. Tiered Memory Management Beyond Hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI'25)*, pages 731–747, 2025.

[53] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. Tiered memory management beyond hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 731–747, 2025.

[54] Ming Liu. *Building Distributed Systems Using Programmable Networks*. University of Washington, 2020.

[55] Ming Liu. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*, page 118–126, 2023.

[56] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*, page 318–333, 2019.

[57] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. Fine-Grained Replicated State Machines for a Cluster Storage System . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 305–323, 2020.

[58] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 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)*, page 795–809, 2017.

[59] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 363–378, 2019.

[60] Liang Luo, Ming Liu, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Motivating in-network aggregation for distributed deep neural network training. In *Workshop on Approximate Computing Across the Stack*, 2017.

[61] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 3*, page 742–755, 2023.

[62] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*, page 106–122, 2021.

[63] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An elastic zoned namespace for commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, pages 461–477, 2023.

[64] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization. *ACM Trans. Storage*, 20(3), June 2024.

[65] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*, page 537–550, 2015.

[66] Nevine Nassif, Ashley O. Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexandra M. Kern, Bill Bowhill, David R. Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad M. Haq, Roopali Sharma, and Kevin Duda. Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC'22)*, volume 65, pages 44–46, 2022.

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

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

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

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

[71] Debendra Das Sharma, Robert Blankenship, and Daniel S Berger. An introduction to the compute express link (cxl) interconnect. *arXiv preprint arXiv:2306.11227*, 2023.

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

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

[74] Kevin Song, Jiacheng Yang, Zixuan Wang, Jishen Zhao, Sihang Liu, and Gennady Pekhimenko. HybridTier: an Adaptive and Lightweight CXL-Memory Tiering System. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 112–128, 2025.

[75] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *Proceedings of the ACM SIGCOMM'98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 118–130, 1998.

[76] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, page 105–121, 2023.

[77] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[78] Midhul Vuppalapati and Rachit Agarwal. Tiered Memory Management: Access Latency is the Key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 79–94, 2024.

[79] Midhul Vuppalapati and Rachit Agarwal. Tiered Memory Management: Access Latency is the Key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 79–94, 2024.

[80] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. Understanding the host network. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 581–594, 2024.

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

[82] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad:{Non-Exclusive} Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, pages 19–35, 2024.

[83] Xincheng Xie, Wentao Hou, Zerui Guo, and Ming Liu. Building massive MIMO baseband processing on a Single-Node supercomputer. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*, pages 1221–1242, 2025.

[84] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. Twenty years after: Hierarchical {Core-Stateless} fair queueing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, pages 29–45, 2021.

[85] Hua Zhang, Xiao Li, Yuebin Bai, and Ming Liu. Understanding and Optimizing Database Pushdown on Disaggregated Storage. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'26), Volume 2*, 2026.

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

[87] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, pages 658–674, 2023.

[88] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), June 2022.

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

[90] Chenxingyu Zhao, Hongtao Zhang, Jaehong Min, Shengkai Lin, Wei Zhang, Kaiyuan Zhang, Ming Liu, and Arvind Krishnamurthy. 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 (ASPLOS'26), Volume 2*, 2026.

[91] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, pages 37–56, 2024.

[92] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*, page 523–536, 2015.

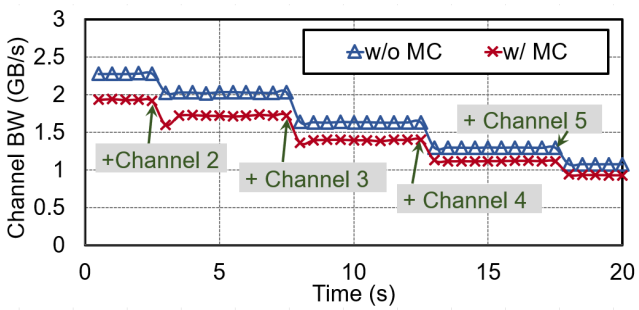


Figure 13: Channel bandwidth when adding four channels compared between with and without MemChannel cases.

A Appendix

A.1 MemChannel Adaptiveness

We further examine how the channel bandwidth adjusts when adding more `mchannels` at runtime. We first run a synthetic workload with and without MemChannel. Then, at 2.5s, 7.5s, 12.5s, and 17.5s, we gradually add one more synthetic workload and measure the average channel bandwidth every half second. When a `mchannel` is added to the system, MemChannel updates the fair share rate and regulates the remote memory access rate (Figure 13), causing a bandwidth drop at 3s, 8s, 13s, and 18s. Subsequently, the transport algorithm recalculates the hardware processing capacity based on congestion signals, increasing the average bandwidth.