Quantifying the Cost of Software-Managed Synchronization on Multi-Host CXL Memory

Peiran Du*, Weishu Deng*, Yang Song*, Chenfei Liao*, Yujie Yang*, Ren Wang†, Hui Lu*, Jia Rao*
*The University of Texas at Arlington, Arlington, TX, USA
†Intel Labs, Portland, OR, USA
pxd2350@mavs.uta.edu, weishu.deng@mavs.uta.edu, yxs7088@mavs.uta.edu,
cxl4830@mavs.uta.edu, yujie.yang@uta.edu ren.wang@intel.com, hui.lu@uta.edu, jia.rao@uta.edu

***Abstract*—Current Compute Express Link (CXL) platforms enable multi-host shared, byte-addressable memory without cross-host hardware cache coherence or atomic operations, forming a hardware-supported distributed shared memory (DSM). However, this CXL-DSM model requires software-managed cross-host coherence and synchronization, introducing significant challenges for scaling parallel applications beyond raw CXL memory performance. This paper presents an end-to-end characterization of this cost stack on a real four-host CXL platform, spanning memory bandwidth and latency, publish/observe visibility primitives, synchronization mechanisms, and application-level behavior. We show that cross-host visibility incurs ∼1 μs latency and lock acquisition costs 3–5 μs, significantly exceeding local hardware synchronization. These costs dominate in synchronization-intensive workloads and prevent scalable multi-host execution, with the volume of republished shared state, rather than the count of lock or barrier operations, as the deciding factor. In contrast, workloads with read-dominated or partitioned, owner-local writes can achieve net speedup by leveraging additional cores and cache capacity. Across microbenchmarks, graph workloads, and a concurrent hash table, application scalability tracks remote-write intensity, synchronization frequency, and data placement. Our results highlight fundamental limits of non-coherent shared memory and provide practical guidance for designing and deploying applications on current multi-host non-coherent CXL systems.**

***Index Terms*—Compute Express Link (CXL), shared memory, software-managed coherence, non-coherent memory, synchronization, hybrid memory placement, memory disaggregation, performance characterization**

I. Introduction

Compute Express Link (CXL) enables processors to access pooled Type-3 memory devices over a PCIe-based fabric [1], [2], and recent multi-host platforms can map the same physical CXL memory into several hosts, creating a hardware-supported distributed shared memory (DSM). Unlike classical DSM systems that relied on page-granularity data access and coherence [3]–[5], CXL provides byte-addressable, load/store access to a shared memory device. Recent systems already exploit this capability for cross-node communication [6], [7].

However, current multi-host CXL platforms maintain cache coherence only within each host. Cross-host cache-line coherence and atomic RMW operations are not available, and extending hardware-managed directories across hosts would substantially increase complexity [8], [9]. In this non-coherent model, writers must *publish* updates by flushing modified cache lines (e.g., `clwb`/`clflushopt` followed by fences), while readers must *observe* by invalidating and refetching to avoid stale copies. Mechanisms that are cheap on coherent shared memory, such as atomics, spin locks, and barriers, must be rebuilt using explicit visibility operations and polling as the building blocks [8], [10].

In CXL-enabled DSM systems, two key questions arise: (1) *what performance does CXL memory deliver in cross-host settings*, and (2) *what are the costs of software-managed coherence and synchronization, and how do they impact application scalability across multiple hosts?* We address these questions through a systematic measurement study on a real four-host CXL-DSM platform. We characterize each layer of the multi-host CXL cost stack, including raw CXL memory performance, software-managed cross-host visibility, synchronization primitives, and end-to-end application behavior.

Multi-host CXL supplies what classical software DSM lacked: hardware support for a cacheline-granular shared address space, while still leaving cross-host cache coherence to software. This makes CXL-DSM a natural target for porting message-passing or RDMA-based distributed applications to a load/store model, improving programmability and reducing software-stack overhead. Our measurements indicate that the resulting design point is best suited to applications that (i) have irregular sharing or communication patterns awkward to express as message passing, and (ii) rely on read-mostly shared data—precisely the regime where a shared load/store abstraction pays off despite software-managed coherence.

Prior work on CXL has largely focused on three directions. *First*, existing studies characterize CXL devices and architectures, but are limited to single-host settings and raw memory performance [11]–[14], leaving multi-host visibility and synchronization costs unexplored. *Second*, a substantial body of work treats CXL as a capacity extension via tiering and pooling, emphasizing page-level placement and migration policies without addressing fine-grained cross-host sharing [15]–[20]; prior memory-level parallelism (MLP)-aware designs [21] also do not capture the widening DRAM–CXL gap under high concurrency. *Third*, recent efforts demonstrate the feasibility of multi-host CXL sharing and software-based synchronization [6], [7], [10], but do not quantify visibility costs or systematically evaluate synchronization primitives and their application impact.

SC26, November 15-20, 2026, Chicago, Illinois, USA
979-8-3195-4789-7/26/$31.00 ©2026 IEEE

In contrast, this paper provides the first end-to-end characterization of the full cost stack in multi-host CXL systems, offering a clear understanding of when and why CXL-DSM is effective. This paper makes the following contributions:

- **End-to-end characterization of the CXL-DSM cost stack**. We performed the first comprehensive study on a real multi-host CXL platform, covering raw memory performance, software-managed visibility, synchronization primitives, and application-level scaling. Although raw CXL performance is lower than DRAM, particularly under high concurrency, the gap narrows for single-threaded, sequential access, the pattern of mutually exclusive critical sections placed in CXL memory.
- **Quantification of software-managed coherence cost**. We isolate and measure the minimal cross-host visibility overhead: $\sim$0.9–1.5 μs one-way, dominating raw CXL memory latency yet comparable to RDMA.
- **Adaptation and evaluation of synchronization mechanisms**. We implement and compare classical centralized and decentralized locking algorithms on non-coherent shared memory: lock acquisition ($\sim$3–5 μs) and software atomics ($\sim$3 μs) run an order of magnitude slower than local primitives.
- **Application-level analysis and workload-fit guideline**. Synchronization-intensive workloads such as BFS fail to scale, moderately synchronized PageRank gains little, and read-dominated triangle counting and a concurrent hash table realize net speedup; scalability is governed by remote-write intensity, synchronization frequency, and data placement.

Artifact availability. All source code, measurement scripts, and reproduction instructions for every figure and table are publicly available[1] and detailed in the Artifact Description and Artifact Evaluation appendices.

II. System Model and Platform

A. CXL Memory and Usage Models

CXL [1] is an open-standard interconnect that enables processors to access memory devices over the PCIe bus, either within a single host or across multiple hosts via a CXL switch. By decoupling memory from the traditional memory bus, CXL enables memory disaggregation with greater scalability and flexibility. While CXL memory can be accessed via ordinary `load` and `store` instructions, it is noticeably slower than DRAM due to PCIe link limits, CXL protocol overhead, and queuing in the CPU uncore and CXL devices [11]–[14].

Prior work primarily treats CXL as a capacity extension, via transparent page tiering, flat memory modes, CXL-accelerated swap, and elastic memory pooling [12], [15]–[20], [22]. These page-granularity approaches do not address fine-grained cross-host sharing, whose primary challenges lie in software-managed visibility, coherence, and synchronization.

[1]Repository: https://github.com/dupeiran001/cxl-shm-bench (archived at https://doi.org/10.5281/zenodo.21753320).

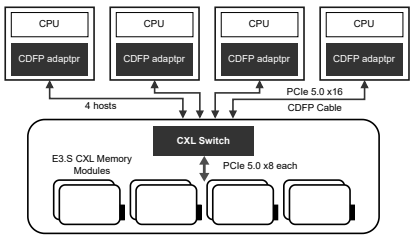


Fig. 1: Architecture of a 4-host CXL-DSM system based on a CXL switch.

B. CXL-enabled Distributed Shared Memory

Classical software DSM pursued shared-memory programmability on distributed hardware [3]–[5], but enforced coherence at page granularity through protection faults and network messages, which caused false sharing and prohibitive overhead for fine-grained access.

CXL, in contrast, provides hardware-supported, byte-addressable access to shared memory devices across multiple hosts [23]. Commercial CXL products are emerging, and CXL memory support has become an integral feature of recent production server platforms. Although the CXL specification outlines the possibilities of supporting cross-host hardware coherence, neither research prototypes nor commercial products have yet demonstrated its practicality and scalability in real systems. Current CXL coherence mechanisms are strictly host-local [9] and lack the snoop paths, directory structures, and global ordering required for cross-host cache coherence. Extending hardware-managed coherence across hosts would substantially increase hardware complexity and overhead, and constrain scalability.

C. Platform and Memory Model

Figure 1 illustrates our CXL-DSM platform. At its core, the system employs an H3 Falcon C5022 CXL memory server that provides pooled CXL memory to four hosts through an XConn XC50256 CXL switch. The memory server is equipped with eight Micron CZ120 128GB CXL Type-3 memory devices [24]. Each device connects to the memory pooling server via a PCIe 5.0 $\times$8 link, while hosts access the memory server through PCIe 5.0 $\times$16 CDFP connections using PCIe-to-CDFP adapters. The switch exposes one or more memory modules to each host as a single direct access (`DAX`) device, with individual modules mapped at fixed physical offsets within that address space. To fully utilize the aggregate bandwidth across multiple CXL devices, we implemented a software kernel module that performs weighted round-robin page allocation at fault time. Note that neither the CXL switch nor Linux interleaving [25] can interleave within a single `DAX` device. A single or multiple CXL memory devices can be mapped to more than one host, enabling cross-host memory sharing. Processors on any connected host access the mapped CXL device using ordinary `load` and `store` instructions.

Cache coherence model. In the CXL-DSM platform, CXL memory is remotely mapped from the memory server into

Config.	A	B	C	D
CPU	Xeon 6787P, 86 cores, 2.0 GHz, L2: 172 MB, LLC: 336 MB			
DRAM	8 × 32 GB DDR5 (6400 MT/s)			
CXL	–	1×128 GB (local)	1×128 GB (remote)	1–8×128 GB (shared & interleaved)
Hosts	1	1	1	1–4

TABLE I: Hardware and memory configurations.

each host's address space. Within a host, the CPU enforces hardware cache coherence for CXL memory in the same manner as for local DRAM, allowing multiple cores to access it with standard coherence guarantees under the x86 total store order (TSO) memory model. However, CXL Type-3 devices are passive: they do not cache data, track cacheline states, or participate in snooping. Instead, the host treats CXL memory as cacheable and relies on its native cache-coherence mechanisms to ensure correctness within each host.

Across hosts, however, no such hardware coherence exists: CPUs on different hosts may independently cache the data in CXL memory at same address, but their caches are not mutually coherent. There is no hardware mechanism, such as back-invalidation, snoop filtering, or a global directory, to propagate updates across hosts. As a result, memory visibility must be explicitly managed by software rather than implicitly ensured by hardware coherence.

To ensure cross-host visibility, a writer must explicitly *publish* updates by flushing modified cache lines from the local cache hierarchy (e.g., using `clflushopt` or `clwb` followed by a fence), making the data visible in the shared CXL device. `clflushopt` invalidates the cacheline and writes back dirty data; `clwb` preserves the cacheline in cache while writing back dirty data, performing no operation if the line is clean. Correspondingly, a reader on another host must *observe* the update by invalidating any locally cached copy of the target cacheline (e.g., via `clflushopt`) before loading, to ensure it fetches the most recent value from CXL memory rather than a stale cache-resident copy.

Atomic instructions maintain their standard semantics within a single host, where the x86 TSO memory model is preserved. However, these hardware atomics (e.g., `cmpxchg`) do not extend to CXL memory shared across multiple hosts, as their correctness relies on cache-coherence protocols that are confined to a single host. Consequently, cross-host synchronization must be implemented in software using the publish/observe primitives described above.

III. Basic Performance Characterization

In this section, we evaluate the core performance characteristics of CXL-DSM, including bandwidth and latency, and compare them with traditional DRAM and local CXL memory attached via PCIe. We focus on primitive operations for software-managed coherence, such as cacheline write-backs and flushes. These results provide insights into the effective use of shared CXL memory and the overhead of software-managed coherence in multi-host settings. Table I summarizes four configurations: DRAM-only baseline (A), locally attached CXL (B), remote CXL via the memory server (C), and multi-host shared CXL with 1-8 devices across 1-4 hosts (D). Shared configurations use software page-level interleaving across CXL modules to maximize aggregate bandwidth. Unless noted, every reported value is the mean of at least three runs; error bars and ± ranges denote the observed run-to-run variation. Detailed platform configuration and the exact reproduction procedure are provided in the artifact appendix.

A. Bandwidth

Unlike prior single-host CXL characterizations [11], [13], our evaluation focuses on multi-host shared access. We extended LMBENCH [26] to support `DAX` allocation and non-temporal stores. Each thread accesses a 1 GB non-overlapping region performing sequential read or non-temporal write (NTWR) operations; for multi-host experiments, threads are evenly distributed across hosts.

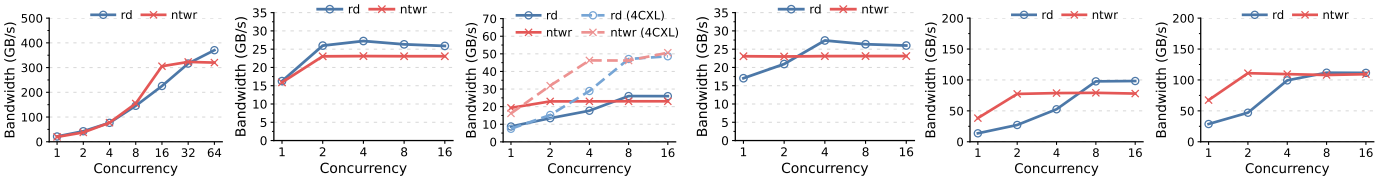
Figure 2 presents bandwidth results (aggregate across all hosts, averaged over 10 runs). CXL memory bandwidth is substantially lower than DRAM, decreasing as memory moves farther from the CPU: per-thread read bandwidth drops from 21 GB/s (DRAM) to 16 GB/s (local CXL) to 9 GB/s (remote CXL), reductions of 23.8% and 57.1% respectively. Page-level interleaving across four CXL devices achieves 50 GB/s peak with 16 threads (Figure 2(c)), approaching the practical PCIe Gen 5 x16 limit (64 GB/s theoretical). The peak bandwidth of a single CXL device remains the same whether accessed by one host or concurrently by multiple hosts (Figures 2(b,d)), confirming that per-device bandwidth is the limiting factor, not cross-host interference. For remote CXL, non-temporal stores outperform loads at low thread counts but are surpassed as concurrency increases. In addition to the per-host PCIe bandwidth limit, the CXL switch itself becomes a bottleneck at approximately 110 GB/s aggregated bandwidth (Figure 2(f)).

Analysis. Since both DRAM and CXL memory use DDR as the underlying medium, the higher CXL latency arises from its PCIe-based, packetized access path, which introduces additional uncore, protocol, switch, and device-side overheads.

This observation has important implications for memory placement. While MLP helps hide CXL's higher latency in single-thread sequential access, resulting in moderate bandwidth degradation (e.g., 21 GB/s on DRAM vs. 16 GB/s on local CXL), the performance gap between DRAM and CXL memory widens as thread-level concurrency increases. Consequently, bandwidth-intensive workloads with high concurrency, common in parallel applications, should be placed on DRAM to fully exploit hardware parallelism (e.g., multiple memory channels). This contrasts with prior work [21], which assumes CXL bandwidth is not saturated and recommends placing high-MLP workloads on CXL memory.

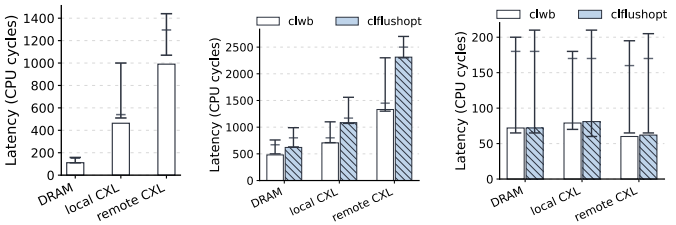
B. Latency

We developed a pointer-chasing benchmark to measure the latency of cross-host publish/observe operations on DRAM, local CXL, and remote CXL. The benchmark traverses a randomly shuffled circular linked list of 64-byte cacheline-aligned



(a) DRAM. **(b)** 1 host, local CXL. **(c)** 1 host, remote CXL. **(d)** 2 hosts, 1 CXL. **(e)** 2 hosts, 4 CXL. **(f)** 4 hosts, 8 CXL.

Fig. 2: Bandwidth test using LMBENCH.



(a) Random `load`. **(b)** `RMW`→flush. **(c)** Clean cacheline flush.

Fig. 3: Latency test using random pointer-chasing.

elements with a 1 GB working set that far exceeds the LLC, so that accesses consistently miss the cache hierarchy. Per-access latency is measured using time-stamp counter (TSC) differences, with distributions constructed via online binning. We evaluate three operations: (1) *Load latency*: servicing an LLC-miss read of a 64-byte cacheline; (2) *Publish latency (RMW→flush)*: a load, store, cacheline writeback (`clwb` or `clflushopt`), and `mfence` to make updates visible across hosts; and (3) *Clean flush latency*: `clflushopt` followed by `sfence`, invalidating local cachelines before reading the latest value from CXL memory.

Figure 3 reports the latency of the three operations on DRAM, local CXL, and remote CXL. Bars show average latency, with markers indicating P_{50}, P_{95}, and P_{99}. Remote CXL significantly increases cache-miss load latency (Figure 3(a)) due to the longer access path through the switch and memory server. The store→flush sequence (Figure 3(b)) follows a similar trend, with an additional 200–350 cycles (`clwb`) from write-back completion across the fabric. In contrast, flush primitives show comparable overhead for `clwb` and `clflushopt`: on clean cachelines (Figure 3(c)), both incur about 70 cycles across all memory types, suggesting that instruction overhead is largely independent of memory distance. Overall, the dominant costs of software-managed visibility on CXL arise from remote cache-miss fetches and write-back completion, while choosing between `clwb` and `clflushopt` matters mainly for dirty-line write-backs, where `clwb` is cheaper, and has minimal impact on clean-line flushes.

C. Minimal Cost of Software Coherence

In this section, we evaluate the cost of software-managed coherence for shared CXL memory in its minimal form. To measure the minimal cost of software-managed visibility, the benchmark uses two threads on separate hosts sharing two cachelines in remote CXL memory. Each cacheline contains a 64-bit sequence number and a payload. As shown in Fig. 4, both threads repeat: ❶ increment the sequence number in one cache line; ❷ flush/writeback the cache line and fence to publish the update; ❸ repeatedly flush and poll the other cache line until its sequence number advances by one. Similar cacheline flush-based software coherence mechanisms have been used in recent work [6], [7].

Accurately quantifying the cost of this straightforward software coherence mechanism is challenging. Unlike the one-sided flush/fence latencies in §III-B, the actual cost of synchronization primitives (locks, atomics, and barriers) built on software-managed coherence is determined by the end-to-end interval from when a writer updates a cacheline to when a reader observes and fetches the new value. For example, a memory fence following a cache-line flush on the writer side only ensures that the writeback has been issued and correctly ordered, not that the update has reached memory. Without hardware-level notification mechanisms such as cacheline back-invalidation, readers must rely on polling to detect value changes. Consequently, the two threads in the software coherence benchmark operate in lock-step to maintain a coherent view of the two shared cachelines.

Table II lists the statistics of the software coherence benchmark with two types of cacheline flush operations. `clflushopt` invalidates a cacheline and writes its contents back to memory if the line is dirty. In contrast, `clwb` preserves the cacheline in the cache and writes back dirty data to memory, performing no operation if the cache line is clean. As shown in Table II, both the publish and observe latencies are significantly higher than those in the one-sided test. In the lock-step benchmark, publish latency includes an RMW operation and a fence. As shown in prior work [27], the read phase of an RMW on a frequently flushed line can stall waiting for a preceding flush to complete. With `clwb`, publish latency is significantly shorter since the cacheline remains cached for the next RMW *publish* operation. The *observe* latency measures the time from when a reader publishes its own update until it sees the remote writer's update, encompassing writer-side publish, reader-side flush/fetch, and several polling attempts. Polling overhead (Figure 3(c)) is insignificant relative to publish and observe latencies. The round-trip time (RTT) sums the *publish* and *observe* phases, covering one update issued by each thread. Consequently, half of the RTT provides a reliable estimate of the per-update cost of software-managed coherence, including the overhead of the associated `load`,

	clflushopt	clwb
Publish latency (ns)	1076.53	602.92
Observe latency (ns)	1828.57	1242.39
Average polling attempts	3.76	2.43
RTT (ns)	2933.96	1873.58
One-way (ns)	1466.98	936.79

TABLE II: The cost of cross-host software coherence.

Operation (8 B)	RDMA IB-NDR (switch)	RDMA 100 GbE (direct)	CXL-DSM (software ld/st)
Write / publish	1.91 μs	1.80 μs	0.94 μs
Read / observe	3.42 μs	3.16 μs	1.24 μs
Atomic (FAA)	3.42 μs	3.14 μs	$\sim$3.0 μs

TABLE III: Per-operation latency (typical, 8 B), RDMA vs. CXL-DSM software visibility.

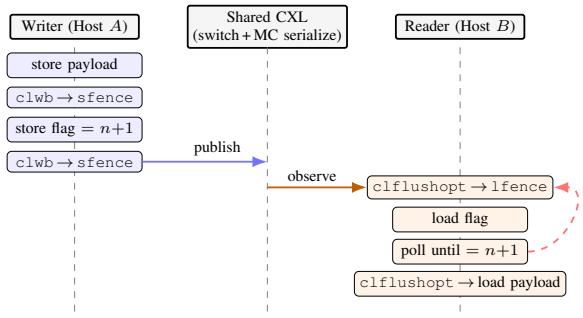


Fig. 4: Software-managed publish/observe on non-coherent CXL.

`store`, `flush`, and `fence` instructions as well as the time for the underlying memory system to properly order these events across two hosts. Note that the true cost of software coherence is substantially higher than the sum of individual operations in Figure 3.

Comparison with network-based synchronization. Table III places the measured CXL-DSM visibility latency in the context of common inter-node synchronization based on network communications [2]. CXL-DSM achieves one-way visibility in the same order of magnitude as one-sided RDMA [28], but with different semantics: CXL preserves a shared address space and direct data-structure access via load/store instructions, eliminating the need for RDMA verbs, queue pairs, memory registration, and connection pairing between endpoints [6]. However, CXL-DSM still requires explicit flushes and polling for cross-host visibility. Compared with the traditional Ethernet interface, CXL-DSM achieves at least an order of magnitude lower latency due to the elimination of the software network stack.

Summary. While peak CXL performance is significantly lower than DRAM under bandwidth-intensive concurrent workloads, the gap is much smaller for single-threaded sequential access, common in mutually exclusive synchronization, because MLP and hardware prefetchers partially hide CXL's higher latency. Moreover, the dominant cost in multi-host software coherence arises from publish/observe operations rather than raw CXL access, and the overall coherence overhead is comparable to RDMA-based communication. These observations highlight an important tradeoff when scaling traditional shared-memory parallel applications on CXL-DSM.

IV. Software Coherence-based Synchronization

Building synchronization primitives on software coherence is substantially more challenging than the two-thread visibility benchmark [29], [30]. Each primitive must pay the $\sim$1 μs visibility cost at least once per operation, support more than two participants, scale under contention, and provide fairness guarantees. In conventional shared-memory systems, efficient synchronization relies on cache coherence and atomic RMW instructions, the basis of OS primitives such as `futex` and `pthread` mutexes. Neither is available across hosts in our multi-host CXL setting; classical software-only mutual exclusion algorithms remain correct when combined with publish/observe operations, but their performance varies substantially. This section quantifies those trade-offs.

Correctness of software-managed synchronization. The classical mutual-exclusion algorithms we use (Lamport's fast mutex, tournament Peterson, and Bakery) are proven correct over an abstract machine of atomic, order-respecting reads and writes to shared registers (no atomic RMW needed). Our task is to realize that abstraction on non-coherent CXL memory, which provides neither cross-host coherence nor cross-host atomics. Two properties suffice. *(i) Untorn access.* A cacheline-aligned 8-byte load or store returns or installs a single value, never a partial one; we place each synchronization variable (intent flags, tickets, turn variables, release metadata) in its own cacheline, so a reader observes either the old or the new value, never a torn one. *(ii) Serialized visibility.* Because synchronization accesses bypass the local cache via explicit flush/invalidate, every such read and write is resolved at the CXL switch and pooling-server memory controller, which serialize them into a single global order. We expose this order with a *publish/observe* protocol (Fig. 4): a writer writes a value and then issues `clwb`→`sfence` to *publish* it, and a reader issues `clflushopt`→`lfence` before each load to *observe* the latest value, polling a monotonically increasing sequence number until the update is visible. Program order plus the publish fence guarantees that intent flags and tickets become visible before the synchronization decisions that depend on them, and that a lock holder writes back and fences protected data before releasing, so the next holder reloads current data on acquisition. Under these two properties the algorithms operate exactly over the ordered read/write-register model they assume, so their safety and progress proofs carry over unchanged.

A. Centralized Locking

Normal spin locks rely on cache coherence and atomic RMW instructions, neither available across hosts in our setup.

[2]Measured on a pair of NVIDIA BlueField-3 cards. The switched path traverses an NVIDIA MQM9700 (Quantum-2 NDR InfiniBand) switch running MLNX-OS via an 800 Gb/s OSFP-to-2×400 Gb/s splitter cable (one 400 Gb/s leg per host); the direct path connects the same cards' second port over a 100 Gb/s QSFP copper cable back-to-back

We therefore implement a centralized lock server as a baseline to coordinate lock acquisition and release. The server maintains per-lock metadata tracking ownership and a FIFO waiter queue. Hosts communicate with the server via message queues, each being a pair of 64-byte CXL cachelines, using the publish/observe protocol of §III-C. Multiple message queues per host allow concurrent requests for different locks.

To acquire a lock, a thread first reserves a message queue on its host using a local atomic instruction, so same-host contention is resolved locally before contacting the server. The winner writes the target lock ID to one cache line and polls the second line for the server's response. The server continuously polls all message queues, granting the lock if free or enqueuing the requester; release follows the same message-queue path. This design provides first-come, first-served (FCFS) fairness but is both a single point of failure and a performance bottleneck. Without hardware cacheline invalidation, polling is the only viable method for detecting updates across hosts, and it consumes substantial CPU time as the system scales.

B. Decentralized Locking Algorithms

Unlike centralized locking that concentrates contention on a single shared lock variable, decentralized locking distributes synchronization across per-thread local variables and uses a coordination protocol to regulate access. Polling is only necessary under contention and can be entirely avoided in uncontended cases, freeing the CPU for useful work. We evaluate three classical software-only algorithms that rely solely on ordinary loads and stores, making them well suited to non-coherent shared memory systems when combined with publish/observe operations.

Lamport FE (LFM) [31]. Lamport's fast mutual exclusion separates lock acquisition into a low-contention fast path and a contention-handling slow path. A thread announces its intent for synchronization using a per-thread intent flag and a shared coordination variable, entering the critical section directly when no competitor is observed and falling back to a polling-based slow path under contention. Because LFM uses ordinary loads and stores rather than hardware atomics, it achieves the lowest shared-state traffic but sacrifices strict fairness.

Peterson (tournament) [32]. Peterson's algorithm ensures mutual exclusion for two threads using per-thread intent flags and a shared turn variable. Each thread signals intent, yields priority to the other via the turn variable, and then spins until it is safe to enter, incurring coordination overhead even in the uncontended case while guaranteeing progress and starvation freedom. For more than two threads, Peterson-style locks can be extended via tournament trees, increasing coordination depth and overhead.

Bakery [33]. Lamport's bakery algorithm generalizes mutual exclusion to N participants through ticket selection and lexicographic ordering. It provides fully decentralized FCFS fairness but incurs $O(N)$ reads per acquisition, stressing read/write amplification and visibility costs on non-coherent shared memory under contention.

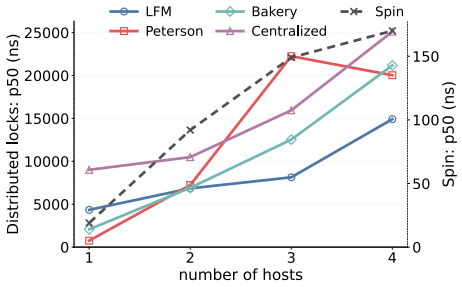


Fig. 5: Scalability of decentralized locks. The hardware spinlock reference uses the right-hand y-axis.

C. Synchronization Performance

Table IV summarizes the properties and per-operation memory access of the four designs. Although all build on the same publish/observe substrate, they differ in fairness, starvation freedom, and shared-state traffic, illustrating that even with identical visibility costs, synchronization design choices can lead to significantly different overheads.

We benchmark lock acquisition using the aforementioned approaches on shared CXL memory with a zero-length critical section to isolate synchronization overhead. Unless otherwise specified, each host runs a single contending thread, and we vary (1) the number of hosts (threads) from 1 to 4 and (2) the number of independent locks (1, 2, 4, 8) to control contention levels. We report median (P_{50}) latency as the primary metric and also examine tail latency (P_{95}/P_{99}) to capture variability from polling and cross-host coherence overheads.

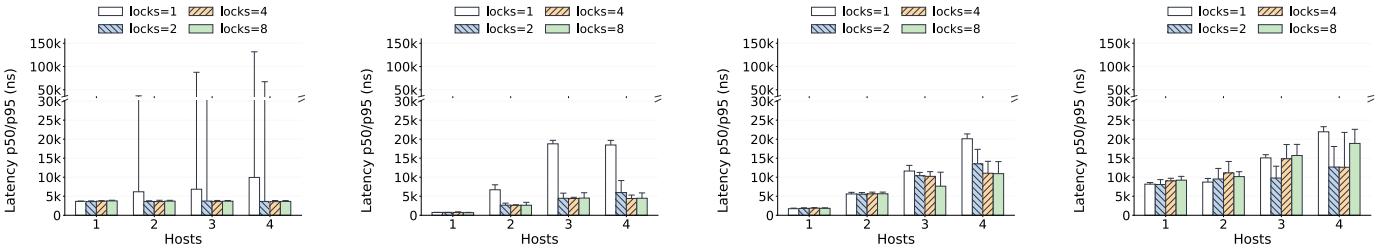
Figure 5 compares median lock acquisition latency as the number of participating hosts (threads) increases. Due to the CXL memory server's limitation, we extrapolate scalability trends of software locks up to 4 hosts. For reference, we include a conventional spin lock using hardware atomics on a single host, which achieves an order-of-magnitude lower latency, highlighting the overhead of software-managed coherence. With a zero-length critical section, acquisition latency for all locks grows nearly linearly with the number of hosts. Among the software-managed designs, LFM achieves the lowest latency under increasing contention, largely due to its efficient fast path.

Figure 6 varies the number of independent locks (1–8) to control contention, with hosts requesting locks in a round-robin manner. As contention decreases through sharding (e.g., 4 or 8 locks), the median latencies of all distributed designs converge, since most acquisitions complete without prolonged waiting. Under maximum contention (1 lock), LFM achieves the lowest median but exhibits a larger P_{95}–P_{50} gap, indicating higher variability in its polling and handoff behavior: while most acquisitions are fast, a subset incurs additional visibility delays or repeated lost handoffs, inflating tail latency. In contrast, Bakery and Peterson locks have higher medians but more stable tails, while centralized locking consistently performs the worst, reflecting inherent coordination overhead even under low contention.

Algorithm	Fair	Starv.-free	Scale	R/W per Acquire		R/W per Release	
				R	W	R	W
Lamport fast ME (LFM) [31]	No	No	N	$2-\infty$	$3-4$	0	2
Peterson (tournament) [32]	No	Yes	N	$2L{+}1-\infty$	$2L{+}\delta$	1	L
Bakery [33]	Yes (FCFS)	Yes	N	$3N{-}2-\infty$	3	0	1
Centralized locking	Yes	Yes	N	–†	1	–†	1

$L = \lceil \log_2(N) \rceil$ denotes tournament depth, $\delta \in \{0, 1\}$ denotes an optional one-time extra store by the first caller
† Polling-based waiting yields unbounded reads; counts depend on contention and are not directly comparable.

TABLE IV: Properties of centralized and decentralized locking algorithms.



(a) LFM. **(b)** Peterson (tournament). **(c)** Bakery. **(d)** Centralized.

Fig. 6: Lock latency with varying locks and hosts.

These results suggest two practical guidelines. First, local spin locks remain the most efficient within a single host but do not extend to multi-host sharing; the gap is an order of magnitude, underscoring the fundamental cost of replacing hardware coherence with software-managed visibility. Second, among distributed schemes, LFM offers the best median scalability at the cost of higher tails, while fairness-oriented designs trade median latency for predictability. Third, beyond algorithm choice, sharding across multiple locks is often the most effective way to reduce overhead.

D. Additional Synchronization Primitives

We also implement barriers and software atomics on the same coherence substrate.

Barrier. We implement a shared barrier in CXL memory using an LFM lock, an arrival counter, and a generation number. Each arriving host acquires the lock, increments the counter, and releases the lock; non-final arrivals then poll the generation field via repeated `clflushopt`→read until it advances. The last arrival resets the counter, increments and publishes the new generation, and releases the lock. Because waiting hosts do not hold the lock while polling, barrier latency is dominated by the skew in thread arrival rather than lock hold time.

Atomic operations in a multi-host setting, atomic operations can be viewed as a group of operations *lock* → *read-modify-write* → *unlock* based on software-enabled cross-host locking. To reduce contention on shared lock variables, we employ LFM and lock sharding. A shared static hash table in CXL memory maps each atomic object to an LFM lock instance. We observed that the cross-host atomic operation achieves a 3000 ns latency, comparable to the LFM lock latency in Figure 6(a).

E. Synchronization Patterns in Real Applications

With cross-host lock acquisition costing 3–5 μs and software atomics around $\sim$3 μs, understanding the synchronization intensity of real applications is critical for assessing the feasibility of CXL-DSM. We profile representative workloads from PARSEC [34] and SPLASH-2/2x by interposing on synchronization primitives via `LD_PRELOAD`, capturing atomic operation rates, critical-section durations, and barrier wait times. All measurements are conducted on a single coherent host to characterize intrinsic synchronization behavior, independent of CXL overhead.

Table V summarizes the results, from which three key observations emerge. First, locks are ubiquitous and typically guard *short* critical sections: the median duration across all workloads is 2.4,μs, ranging from 56 ns (`fluidanimate`) to 40.9 μs (`dedup`), with acquisition rates spanning four orders of magnitude ($\mathcal{O}(10^2)$–$\mathcal{O}(10^6)$/s). Because these critical sections are often shorter than the 3–5 μs cross-host lock acquisition cost, synchronization overhead would dominate useful work on CXL-DSM. Second, barriers can be a major contributor to execution time in phase-based workloads: for example, `lu_ncb` spends 38.9% runtime in barrier waits, with a maximum single-barrier wait of 10.1 ms. Third, atomic operations are prominent in only a few cases (e.g., `canneal` at 167 Mops/s), indicating that for most traditional multithreaded workloads ported as-is, the primary scalability bottleneck on CXL-DSM lies in lock and barrier overhead rather than atomic throughput.

These profiles enable quantitative feasibility estimates for porting workloads to CXL-DSM. `fluidanimate` acquires locks at 2.7 M/s with a 56 ns average critical section; at $\sim$5 μs per cross-host acquisition, synchronization alone would consume 13.5 s per second of execution, infeasible without restructuring. `dedup`, with longer critical sections (40.9 μs)

Benchmark	Atomic (Mops/s)	Lock acq (K/s)	CS avg (µs)	CS max (µs)	CS time (%)	Barrier waits (K/s)	Barrier avg (µs)	Barrier max (µs)	Barrier time (%)	Total Time (s)
PARSEC										
parsec.canneal	166.68	0.08	28.75	52.13	0.01	10.5	94.4	2111.2	6.19	0.20
parsec.dedup	–	139.1	40.90	3175.23	11.38	–	–	–	–	3.13
parsec.fluidanimate	–	2695.5	0.06	48.09	0.95	–	–	–	–	2.90
parsec.streamcluster	–	421.1	5.36	93.28	7.06	–	–	–	–	1.35
SPLASH-2 / SPLASH-2x										
splash2x.lu_ncb	–	14.0	2.83	54.98	0.25	7.2	865.4	10102.7	38.96	0.58
splash2.radiosity	–	3471.5	1.27	264.43	27.67	0.7	138.1	382.9	0.59	0.42
splash2.raytrace	–	1416.4	3.54	376.76	31.37	–	–	–	–	0.05

TABLE V: Synchronization characteristics of representative multithreaded workloads (region of interest only).

and a lower lock rate (139 K/s), adds $\sim$0.7 s/s, significant but potentially viable with batching or coarsening. `canneal`, dominated by infrequent barriers and lightweight atomics, is the most natural CXL-DSM candidate, whereas `radiosity` and `raytrace`, which spend 28–31% of time in critical sections even on coherent hardware, face a $\sim$100$\times$ increase in per-lock cost, prohibitive without fundamental restructuring.

These observations motivate our choice of GAP Benchmark Suite (GAPBS) graph kernels in §V. Rather than porting synchronization-intensive legacy workloads to CXL-DSM, we focus on applications whose algorithmic structure can be adapted to minimize cross-host synchronization and shared-state publication through owner-local updates, bulk-synchronous execution, and read-dominated access patterns.

V. Application Scalability in CXL-DSM

Building on the cost-stack measurements above, we evaluate four GAPBS [35] graph kernels and a concurrent hashtable, spanning different synchronization patterns, to understand how CXL-DSM's visibility and synchronization tax translates to real application performance and when multi-host shared memory is beneficial.

All graph data is stored in compressed sparse row (CSR) format in a shared CXL memory region, software-interleaved at page granularity across 4$\times$128 GB CXL modules to balance memory traffic. All hosts access the same shared CSR without replication. Intra-host parallelism uses OpenMP, while inter-host work is statically partitioned by vertex ranges and coordinated via LFM locks, barriers, and software atomics (§IV). We evaluate four configurations: local DRAM (single host, up to 64 threads), remote CXL (single host via the switch), and shared CXL on two and four hosts (up to 32 threads per host). Table VI summarizes the key characteristics that determine CXL-DSM fit. The deciding factors are remote-write intensity, synchronization frequency, and whether the workload can keep hot metadata in local DRAM. The following subsections group kernels by workload class, from coordination-heavy to read-dominated.

A. Write-Heavy, coordination-intensive: BFS

We implement a level-synchronous breadth-first search (BFS) based on the direction-optimizing approach of Beamer et al. [36], which exploits the frontier's bell-shaped growth

Kernel	Remote writes?	Owner-local updates?	Barrier intensity	Bulk-sync?	Publish lines/s	Sync share
BFS	High (TD)	Partial	Per-level	Yes	$\sim 4\times10^9$	3.5%
PR	Low	Yes (pull)	Per-iter.	Yes	$\sim 1\times10^8$	0.4%
PR-SPMV	Low	Yes (pull)	2$\times$/iter.	Yes	$\sim 2\times10^8$	0.7%
TC	Minimal	Yes	End-only	No	$\sim 1\times10^7$	0.3%

TABLE VI: Workload characteristics, measured publish rate, and synchronization-time share (4 hosts).

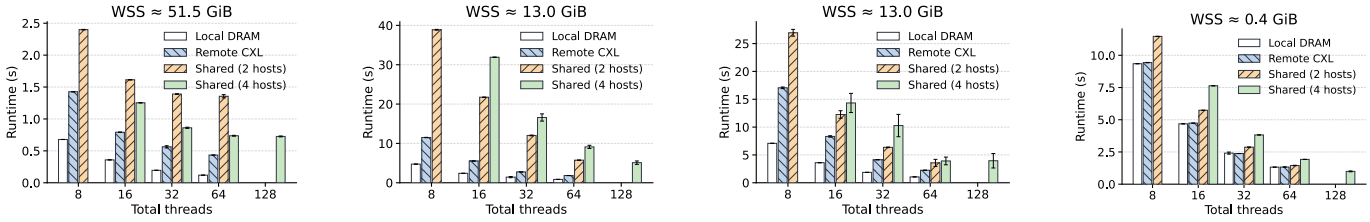
by dynamically switching between top-down (TD) traversal, efficient for small frontiers, and bottom-up (BU) traversal, efficient for large frontiers, using heuristics based on frontier size and unexplored edges.

Top-down (TD) Top-down BFS expands the frontier by traversing adjacency lists and claiming unvisited neighbors via atomic CAS on a shared `parent[]` array, making it inherently write-heavy. Under vertex-range partitioning, most updates target remote hosts, generating cross-host writes. On our CXL platform, each CAS costs $\sim$3 μs (vs. $\sim$100 ns locally), making naive top-down traversal prohibitively expensive.

Bottom-up (BU) Bottom-up BFS reverses the search: each unvisited vertex scans its neighbors for a frontier node, writes its own `parent[]` entry upon success, and stops early. This is owner-local under vertex partitioning, avoiding cross-host writes. The main cost is read-only access to a compact shared frontier bitmap, making bottom-up efficient when the frontier is large despite its worst-case $O(|E_{unexplored}|)$ work.

Adapting BFS to non-coherent shared memory We implement BFS as a bulk-synchronous computation: hosts synchronize at each level using a cross-host barrier (§IV), publish frontier updates via `clwb`$\rightarrow$`sfence`, and refresh shared state with `clflushopt`$\rightarrow$`lfence`. To reduce communication, we use coarsened per-host frontier bitmaps so only modified pages are flushed. A coordinator host determines whether new frontier entries exist and whether traversal should proceed.

Bottom-up fits this model naturally: each host updates its own vertex range and publishes once per level. In contrast, top-down cannot use per-vertex remote atomics ($\sim$ 30x slower than local CAS), so each host buffers remote discoveries in per-destination buffers, synchronizes at a barrier, exchanges them via shared CXL memory, and replays them using fast local atomics. This design replaces fine-grained remote synchronization with bulk communication and owner-local replay.



(a) BFS (sync-heavy). **(b)** PR (owner-local, relaxed). **(c)** PR-SPMV (owner-local, strict). **(d)** TC (read-dominated).

Fig. 7: GAPBS kernel runtimes across configurations. Labels indicate the dominant access pattern.

Results. As shown in Figure 7(a), BFS scales well on local DRAM (5.7× from 8 to 64 threads) and moderately on remote CXL (3.3×), but poorly in the shared multi-host setting, with diminishing returns beyond 16 threads per host. At 32+32 threads, two-host execution (1.38 s) is 2.5× slower than 32 threads on a single host over CXL (0.55 s). Doubling to four hosts helps only partially: four-host BFS reaches 0.74 s at 128 threads—1.9× faster than two hosts, yet still 1.7× slower than single-host remote CXL and 6× slower than DRAM—and plateaus beyond 64 total threads, confirming that cross-host coordination, not compute, bounds BFS. This behavior stems from BFS's synchronization pattern. Each level incurs a global barrier, frontier publication/refresh, and during top-down—buffering, exchanging, and replaying cross-host updates. With millisecond-scale barriers and, above all, the per-level publication and exchange of frontier state, cross-host communication quickly dominates execution even with two hosts and would worsen at larger scales; the lock and barrier operations themselves account for only ∼3.5% of runtime (Table VI). Moreover, placing all data in CXL memory forfeits local DRAM bandwidth; a hybrid design keeping key structures in DRAM could significantly reduce overhead.

B. Bulk-synchronous, owner-local updates: PR and PR-SPMV

PageRank (PR) and its sparse matrix–vector (SpMV) variant (PR-SPMV) represent the middle of the synchronization spectrum: writes are owner-local, but cross-host interaction occurs at every iteration boundary through a dense shared vector.

Both kernels use a pull-based PageRank formulation, where each vertex gathers contributions from its in-neighbors. Under vertex partitioning, updates are owner-local, avoiding cross-host writes, but reads span all neighbors, generating cross-host read traffic proportional to the edge count. The shared state includes a rank vector and a precomputed contribution vector. Each iteration requires hosts to refresh the shared contribution vector and publish updates via `clwb→sfence`. For the 13 GiB graph, the large contribution vector (hundreds of MB) makes this refresh a dominant overhead on CXL-DSM.

PR vs. PR-SPMV. The two variants differ in consistency. PR-SPMV uses strict Jacobi iteration with separate arrays and global barriers, ensuring consistent visibility but incurring two barriers per iteration. PR uses a relaxed Gauss-Seidel approach with in-place updates, reducing iterations but introducing non-deterministic cross-host visibility, which may affect convergence under CXL-DSM.

Results. Figure 7(b–c) shows that both kernels scale across all configurations, but shared execution is consistently the slowest. At 64 threads, PR runs in 0.86 s (DRAM), 1.85 s (remote CXL), and 5.74 s (two-host shared CXL), while PR-SPMV runs in 1.05, 2.25, and 3.57 s. We distinguish two comparisons. *At equal thread count*—64 total, i.e. 32+32 on two hosts vs. 64 on one host over CXL—multi-host is slower for both kernels: 3.1× for PR (5.74 vs. 1.85 s) and 1.6× for PR-SPMV (3.57 vs. 2.25 s), reflecting the per-iteration cost of publishing and refreshing the dense contribution vector. *As a scale-up*—adding a second host to grow from 32 to 32+32 threads—PR-SPMV improves: 32+32 threads (3.57 s) beat 32 threads on one host over CXL (4.13 s), so the added compute outweighs the synchronization overhead; PR does not benefit similarly (5.74 vs. 2.77 s). Beyond two hosts, however, neither kernel improves further: four-host PR-SPMV (median 3.5 s at 128 threads) is indistinguishable from two hosts given the run-to-run spread, and PR only marginally moves (5.22 s)—the dense-vector refresh, not compute, dominates at scale.

C. Read-Dominated: TC

Triangle counting (TC) contrasts sharply with update-intensive workloads. It is dominated by irregular, read-only traversal of adjacency lists, with *no* shared mutable state during the main computation. We use the GAPBS ordered-intersection method [35]: for each edge (u, v) with $u < v$, the algorithm computes $|N(u) \cap N(v)|$ via merge-based intersection, counting only neighbors $w > v$ to avoid duplicates, with total work proportional to $\sum_{(u,v)\in E} \min(\deg(u), \deg(v))$.

This algorithm is well suited for non-coherent shared memory for three reasons. First, it is *read-only*: threads traverse CSR adjacency lists and maintain private counts, requiring no locks, atomics, or cross-host writes until a final reduction. Second, it performs sequential scans over sorted arrays, benefiting from hardware prefetching and CXL's streaming bandwidth. Third, it is embarrassingly parallel, allowing edges or vertex ranges to be partitioned across hosts with no inter-host dependencies during computation. The main challenge is load imbalance. In power-law graphs, a few high-degree hubs dominate the work, and edges between such vertices require costly large-list intersections. Under naive vertex-range

partitioning, hosts owning these hubs become bottlenecks. To mitigate this, we optionally rebalance work by partitioning edges based on cumulative degree, ensuring each host handles roughly equal intersection cost.

Results. Figure 7(d) shows that TC scales similarly across all configurations, with only a modest penalty for shared CXL execution. At 64 threads, DRAM and remote CXL both achieve $\sim$1.3 s, while two-host shared CXL reaches 1.49 s (1.15$\times$ slower). The near parity reflects TC's latency-bound, irregular access pattern, where CXL latency is hidden by parallelism across independent intersection operations.

TC benefits from scale-out. Already at 32+32 threads, two-host execution (1.49 s) beats 32 threads on one host over CXL (2.43 s); and at four hosts TC reaches 0.97 s at 128 threads—faster than single-host DRAM (1.28 s at 64 threads). TC is thus the one kernel where multi-host CXL-DSM outperforms single-host DRAM despite DRAM's significantly higher bandwidth, and the advantage *grows* with host count—demonstrating that read-dominated, synchronization-light workloads scale on CXL-DSM even without hardware coherence.

TC represents the read-dominated end of the synchronization spectrum. Its strong performance on CXL-DSM stems from its fundamental properties: no cross-host writes, an embarrassingly parallel structure, and the ability to tolerate remote read latency through thread-level parallelism. Because TC is limited by read throughput rather than synchronization, it could further benefit from hybrid placement, keeping hot adjacency lists (e.g., high-degree hubs) in local DRAM while storing the bulk of the graph in shared CXL, to narrow the gap with the DRAM-only baseline.

D. A Concurrent Hash Table

The graph kernels are all bulk-synchronous: compute a phase, barrier, exchange shared state. To test whether the workload-fit findings survive a fundamentally different access pattern, we implement a chained hash table that lives entirely in shared CXL memory and is read and written concurrently by all hosts. The table is an array of 2^{20} buckets, each a singly-linked chain of two-cacheline key/value nodes. Within a host, threads serialize on process-local mutex stripes; across hosts, each bucket carries its own LFM lock (§IV), so operations on distinct buckets proceed in parallel, and chain nodes are bump-allocated from per-host slabs, so inserts on different hosts never contend on a shared allocator. All shared accesses use the publish/observe protocol of §III-C: a lookup invalidates and re-reads each line as it walks the chain, an insert writes the node and publishes the head pointer last, and an update rewrites the value line in place. Host 0 preloads 10^6 keys; every thread then issues uniform-random operations at a fixed read ratio for ten seconds, and a cross-host self-check validates that every host observes a consistent table.

Results. Figure 8 shows throughput scales near-linearly with host count: at 32 threads per host, a read-mostly (95%) mix sustains 4.1 Mops on one host (remote CXL), 8.3 Mops on two hosts, and 16.6 Mops on four hosts—1.2$\times$ single-host DRAM (13.6 Mops) despite DRAM's far higher bandwidth. Throughput barely changes as the mix shifts from 100% reads to 50% writes (16.6 vs. 16.7 Mops at four hosts), because the per-bucket locks and per-host slabs keep writes *distributed*: hosts inserting into different buckets neither contend nor republish shared state to one another.

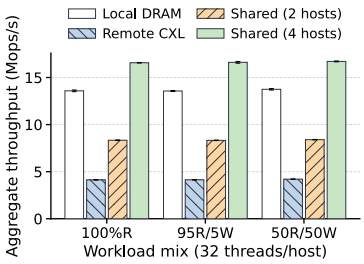


Fig. 8: Concurrent hash-table throughput (32 threads/host) across DRAM, remote CXL, and 2-/4-host shared CXL for read-mostly to write-heavy mixes.

E. CXL-DSM Scalability Analysis

CXL-DSM viability is determined by the ratio of cross-host communication cost (shared-state publication plus synchronization) to useful computation, yielding a clear workload-fit spectrum. Read-dominated workloads with minimal synchronization (e.g., triangle counting) are an excellent fit: they avoid cross-host writes, exhibit embarrassingly parallel structure, and can leverage additional hosts for net speedup. Workloads with owner-local writes but dense shared state and bulk-synchronous execution (e.g., PageRank variants) achieve only moderate gains, as the cost of repeatedly publishing and refreshing large shared vectors offsets added compute capacity; these could improve with hybrid data placement. In contrast, synchronization-intensive workloads (e.g., BFS) are poor fits: frequent barriers, cross-host coordination, and fine-grained updates cause publication and synchronization overhead to dominate, negating any benefit from multi-host scaling. The concurrent hash table (§V-D) sharpens this spectrum: the deciding property is not read-only access per se but *partitioned* writes—workloads whose writes target disjoint shared state republish little across hosts and scale near-linearly, with read-mostly sharing as the common special case—whereas a workload that forces every host to observe every other host's writes (as in top-down BFS) cannot scale regardless of the data structure.

What actually dominates. To separate the cost of the lock/barrier *protocol* from the cost of *moving shared state across hosts*, we instrument the kernels to count synchronization operations and published/observed cachelines within the region of interest (Table VI, right columns). Lock and barrier *operations* consume only a small fraction of runtime (e.g., 1.9–3.5% for BFS), but the *volume* of cross-host publication is what separates the kernels: BFS publishes 1.1–4.2 $\times$ 10^9 cachelines/s (frontier and `parent[]` updates) at two to four hosts, whereas triangle counting publishes only 4–12 $\times$ 10^6/s.

Because each published line carries the $\sim$1 μs cross-host visibility cost of §III-C, aggregate publication traffic—not the number of locks or barriers—tracks the observed fit ordering. The design implication is concrete: reducing *how much* shared state is republished each round matters far more than optimizing the lock algorithm itself.

Table VI summarizes these characteristics. The boundary between fit categories is determined by three measurable factors: 1) the volume of shared state republished per round, set by the fraction of writes targeting remote hosts; 2) the frequency of synchronization operations relative to useful computation; and 3) whether bandwidth-sensitive metadata (e.g., frontiers, contribution vectors, score arrays) can reside in local DRAM. Using the measured costs–visibility ($\sim$1 μs), lock acquisition ($\sim$3–5 μs), and barriers (millisecond-scale) from §III–§IV–designers can perform quick feasibility estimates before undertaking a full CXL-DSM migration.

VI. Related Work

Traditional DSM. IVY [3], TreadMarks [4], and the foundational work of Li and Hudak [5] provided a shared address space through page-granularity coherence, whose page-fault and false-sharing overheads made fine-grained sharing impractical. CXL-DSM operates at cache-line granularity with $\sim$1 μs visibility cost, an order-of-magnitude lower per-update penalty, but provides no automatic coherence: software must manage visibility explicitly, a cost we quantify in §III-C.

Far memory and disaggregation. Infiniswap [37], LegoOS [38], AIFM [39], remote-memory systems [40], Carbink [41], and MIND [42] move memory beyond the socket at page or object granularity, exposing remote paging or key–value interfaces rather than a shared byte-addressable address space. They therefore neither encounter nor measure the cache-line-level visibility and synchronization overheads that dominate CXL-DSM.

CXL memory characterization. Prior studies characterize CXL devices and their architectural implications in single-host settings [11]–[14], without addressing visibility, synchronization, or application-level effects of multi-host sharing. Our results extend this analysis to 1–4 hosts and 1–8 devices (§III), revealing switch bandwidth limits ($\sim$110 GB/s) and the widening DRAM–CXL gap under concurrency.

CXL tiering and pooling. A large body of work manages CXL as a slower page-granularity tier or pooled elastic capacity [15]–[20] without addressing cross-host sharing. Our results refine MLP-aware tiering [21]: MLP hides CXL latency for single-thread sequential access, but the DRAM–CXL gap widens under high concurrency (§III), limiting such policies for bandwidth-intensive parallel workloads.

Multi-host CXL sharing and synchronization. Prior efforts implement MPI-style communication over CXL shared memory, demonstrating the feasibility of flush-based software coherence [6], [7]; other work demonstrates a correct Peterson lock on a two-host CXL system [10]. Neither line of work isolates the cost of visibility operations, compares alternative synchronization designs, or links these costs to application scalability.

Software coherence vs. CXL.cache and future hardware. Current multi-host CXL platforms expose only CXL.mem: a shared load/store address space without cross-host hardware coherence. CXL.cache is unavailable in today's multi-host pooling configurations and would reintroduce the metadata, invalidation-propagation, ordering, and arbitration complexity that motivates its exclusion. Looking forward, back-invalidation (curbing explicit invalidation and polling) or fabric-level atomics (cutting lock-acquisition latency) would directly reduce the costs we measure. A promising middle ground, motivated by our ordering-based atomicity emulation, is a fabric that exposes lightweight sequencing information for serialized reads and writes, shifting the atomicity burden to software while approaching hardware-assisted performance without full cross-host coherence.

Scope and limitations. Our measurements come from one CXL platform, and absolute values will vary with other switches, memory devices, and host processors; what is architectural is that the absence of cross-host invalidation and atomics imposes a software visibility and synchronization tax, whose exact magnitude is platform-specific. All conclusions rest on measured data at up to four hosts, across the microbenchmarks and all five workloads. Extrapolation beyond four hosts is grounded in the measured bottlenecks rather than larger-scale runs: the switch saturates near 110 GB/s and per-device bandwidth is fixed, so per-host bandwidth falls and publication costs compound as participants grow.

VII. Conclusions and Future Work

This paper presents the first end-to-end measurement study of a real multi-host CXL platform, covering memory performance, software-managed coherence, synchronization, and application impact. Scalability is governed not by raw CXL latency but by the cost of software-managed cross-host visibility ($\sim$1 μs per update, lock acquisition 3–5 μs, millisecond-scale barriers), and more precisely by the *volume* of shared state republished each round rather than the count of synchronization operations. The workload-fit rule is concrete: CXL-DSM scales when cross-host republication stays low. Read-dominated workloads fit best (triangle counting beats single-host DRAM at four hosts), workloads with partitioned, owner-local writes scale as well (the hash table scales near-linearly even under 50% writes), workloads that force every host to observe every write (top-down BFS) remain bottlenecked, and bulk-synchronous workloads with dense shared state (PageRank) fall in between, benefiting from hybrid DRAM/CXL placement and reduced publication frequency.

Future work includes hybrid placement policies that keep synchronization- and bandwidth-critical structures in local DRAM, runtime selection of synchronization primitives by contention and tail-latency requirements, and revisiting this cost stack as CXL moves toward cross-host coherence. Blocking primitives (e.g., mutexes) remain unsupported, as they require cross-host notification unavailable today.

REFERENCES

[1] https://www.computeexpresslink.org/.

[2] Compute Express Link Consortium, "CXL® Specification - Compute Express Link," Compute Express Link Consortium, Tech. Rep. Version 4.0, 2025. [Online]. Available: https://computeexpresslink.org/cxl-specification/

[3] K. Li, "IVY: a shared virtual memory system for parallel computing," *Proceedings of the International Conference on Parallel Processing*, vol. 2, pp. 94–101, 1988.

[4] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Tread Marks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *USENIX Winter 1994 Technical Conference (USENIX Winter 1994 Technical Conference)*, 1994.

[5] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, Nov. 1989, https://doi.org/10.1145/75104.75105.

[6] X. Wang, B. Ma, J. Kim, B. Koh, H. Kim, and D. Li, "cMPI: Using CXL Memory Sharing for MPI One-Sided and Two-Sided Inter-Node Communications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2025, https://doi.org/10.1145/3712285.3759816.

[7] X. Yang, Y. Zhang, H. Chen, F. Li, G. Fan, Y. Kong, B. Wang, J. Fang, Y. Wang, T. Huang, W. Hu, J. Kao, and J. Jiang, "Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases," in *Companion of the 2025 International Conference on Management of Data*. Berlin Germany: ACM, Jun. 2025, pp. 689–702, https://doi.org/10.1145/3722212.3724460.

[8] S. Jain, N. Yeleswarapu, H. A. Maruf, and R. Gupta, "Memory Sharing with CXL: Hardware and Software Design Approaches," Apr. 2024.

[9] Y. Xu, S. Mahar, Z. Liu, M. Shen, and S. Swanson, "CXL Shared Memory Programming: Barely Distributed and Almost Persistent," 2024, arXiv:2405.19626.

[10] J. Suetterlein, J. Manzano, and A. Marquez, "Synchronization for CXL Based Memory," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '24. New York, NY, USA: Association for Computing Machinery, Dec. 2024, pp. 178–185, https://doi.org/10.1145/3695794.3695810.

[11] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim, "Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, Dec. 2023, pp. 105–121, https://doi.org/10.1145/3613424.3614256.

[12] H. Ji, S. Vanavasam, Y. Zhou, Q. Xia, J. Huang, Y. Yuan, R. Wang, P. Gupta, B. Chitlur, I. Jeong, and N. S. Kim, "Demystifying a CXL Type-2 Device: A Heterogeneous Cooperative Computing Perspective," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Nov. 2024, pp. 1504–1517, https://doi.org/10.1109/MICRO61859.2024.00110.

[13] J. Liu, H. Hadian, Y. Wang, D. S. Berger, M. Nguyen, X. Jian, S. H. Noh, and H. Li, "Systematic CXL Memory Characterization and Performance Analysis at Scale," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Rotterdam Netherlands: ACM, Mar. 2025, pp. 1203–1217, https://doi.org/10.1145/3676641.3715987.

[14] Y. Yang, L. Xiang, P. Du, Z. Lin, W. Deng, R. Wang, A. Kudryavtsev, L. Ko, H. Lu, and J. Rao, "Architectural and System Implications of CXL-enabled Tiered Memory," Mar. 2025.

[15] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, Mar. 2023, pp. 742–755, https://doi.org/10.1145/3582016.3582063.

[16] "MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination | Proceedings of the 29th Symposium on Operating Systems Principles," https://dl.acm.org/doi/10.1145/3600006.3613167.

[17] L. Xiang, Z. Lin, W. Deng, H. Lu, J. Rao, Y. Yuan, and R. Wang, "Nomad: Non-Exclusive memory tiering via transactional page migration," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 19–35. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/xiang

[18] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble Page Management for Tiered Memory Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Providence RI USA: ACM, Apr. 2019, pp. 331–345, https://doi.org/10.1145/3297858.3304024.

[19] Y. Zhong, D. S. Berger, C. Waldspurger, R. Wee, I. Agarwal, R. Agarwal, F. Hady, K. Kumar, M. D. Hill, M. Chowdhury, and A. Cidon, "Managing Memory Tiers with CXL in Virtualized Environments," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 37–56.

[20] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 574–587, https://doi.org/10.1145/3575693.3578835.

[21] J. Liu, H. Hadian, H. Xu, and H. Li, "Tiered Memory Management Beyond Hotness," 2025.

[22] T. Heo, Y. Wang, W. Cui, J. Huh, and L. Zhang, "Adaptive Page Migration Policy With Huge Pages in Tiered Memory Systems," *IEEE Transactions on Computers*, vol. 71, no. 1, pp. 53–68, Jan. 2022, https://doi.org/10.1109/TC.2020.3036686.

[23] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct Access, High-Performance Memory Disaggregation with DirectCXL," in *USENIX ATC*, 2022.

[24] "Micron memory expansion module using cxl," https://www.micron.com/products/memory/cxl-memory?gad_source=1&gclid=CjwKCAjw3624BhBAEiwAkxgTOkvyPWsMrSX3UT_4-u2_l3B7TyIJhpzhbsrwnxbtfuLbMzZIppxCvBoCfsgQAvD_BwE.

[25] "Multi-Level Interleave — The Linux Kernel documentation," https://docs.kernel.org/driver-api/cxl/linux/example-configurations/multi-interleave.html.

[26] L. McVoy and C. Staelin, "Lmbench: Portable Tools for Performance Analysis," in *USENIX 1996 Annual Technical Conference (USENIX ATC 96)*, 1996.

[27] L. Xiang, X. Zhao, J. Rao, S. Jiang, and H. Jiang, "Characterizing the performance of intel optane persistent memory: A close look at its on-DIMM buffering," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, Mar. 2022, pp. 488–505, https://doi.org/10.1145/3492321.3519556.

[28] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016, pp. 437–450.

[29] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, Feb. 1991, https://doi.org/10.1145/103727.103729.

[30] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[31] L. Lamport, "A fast mutual exclusion algorithm," *ACM Transactions on Computer Systems*, vol. 5, no. 1, pp. 1–11, Jan. 1987, https://doi.org/10.1145/7351.7352.

[32] G. L. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, no. 3, pp. 115–116, Jun. 1981, https://doi.org/10.1016/0020-0190(81)90106-X.

[33] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," *Communications of the ACM*, vol. 17, no. 8, 1974, https://doi.org/10.1145/361082.361093.

[34] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: Association for Computing Machinery, Oct. 2008, pp. 72–81, https://doi.org/10.1145/1454115.1454128.

[35] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," May 2017.

[36] ——, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society Press, 2012, pp. 1–10, https://doi.org/10.1109/SC.2012.50.

[37] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*. USENIX Association, Mar. 2017, pp. 649–667. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi17/nsdi17-gu.pdf

[38] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, Oct. 2018, pp. 69–87. [Online]. Available: https://www.usenix.org/system/files/osdi18-shan.pdf

[39] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "AIFM: High-Performance, Application-Integrated far memory," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, Nov. 2020, pp. 315–332. [Online]. Available: https://www.usenix.org/system/files/osdi20-ruan.pdf

[40] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, "Remote memory in the age of fast networks," in *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, 2017. https://doi.org/10.1145/3127479.3131612. [Online]. Available: https://pages.cs.wisc.edu/~jayneel/papers/socc17_remote_memory.pdf

[41] Y. Zhou, H. M. G. Wassel, S. Liu, J. Gao, J. Mickens, M. Yu, C. Kennelly, P. Turner, D. E. Culler, H. M. Levy, and A. Vahdat, "Carbink: Fault-Tolerant Far Memory," in *USENIX OSDI*, 2022.

[42] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "MIND: In-Network Memory Management for Disaggregated Data Centers," in *ACM SOSP*, 2021, https://doi.org/10.1145/3477132.3483561.