# C$^3$: CXL Coherence Controllers for Heterogeneous Architectures

Anatole Lefort [*]
*Technical University of Munich*
Munich, Germany

David Schall [*]
*Technical University of Munich*
Munich, Germany

Nicolò Carpentieri
*Technical University of Munich*
Munich, Germany

Julian Pritzi
*Technical University of Munich*
Munich, Germany

Soham Chakraborty
*TU Delft*
Delft, Netherlands

Nicolai Oswald
*NVIDIA*
Santa Clara, US

Pramod Bhatotia
*Technical University of Munich*
Munich, Germany

***Abstract*—We introduce C$^3$, a systematic methodology for designing Compute Express Link (CXL) coherence controllers, to overcome interoperability challenges that arise from the mismatch of coherence protocols and memory consistency models in heterogeneous CXL-connected systems. Crucially, CXL lacks a unified heterogeneous computing interface, which can lead to unpredictable and inconsistent behavior when multiple heterogeneous devices decide to share cache-coherent CXL memory. C$^3$ acts as a pivotal interface between diverse heterogeneous compute units, bridging the semantic differences without necessitating disruptive changes to existing system architectures. Our approach hinges on two key principles: delegating memory operations across coherence domains and enforcing atomicity at domain boundaries, thereby preserving the native memory consistency model semantics of each unit. We implement C$^3$ as a generic gem5 model and validate its correctness through exhaustive litmus testing. We also show that C$^3$ incurs minimal performance overhead compared to unified native coherence protocols.**

***Index Terms*—Cache coherence, CXL, disaggregated systems**

## I. Introduction

Modern data centers face growing pressure to reduce energy consumption and improve resource utilization [3], [38], [64]. At the same time, resources tend to be overprovisioned [74], with studies reporting average memory utilization rates below 40% and up to 25% of memory being stranded—allocated but unused [48], [88]. The result is billions of dollars in wasted energy and capital, particularly as DRAM accounts for over half of server hardware costs [62]. These inefficiencies are further exacerbated by the increasing heterogeneity of workloads and compute platforms in modern data centers [10], [33], [60]. Recent developments in high-performance interconnects, such as the recently announced NVLink Fusion technology by NVIDIA [1], and the CXL standard [24], address these wasted capacities by consolidating memory resources into shared pools that can be accessed by different platforms.

Compute Express Link (CXL) has emerged as a transformative interconnect technology enabling highly efficient access to byte-addressable remote memory at the hardware level [23], [24], [26]. CXL notably promises decoupling of compute and memory with independent scaling to reduce stranded memory [62]. The latest version of the specification, CXL v3.0, introduces multi-host coherence, allowing multiple processors—potentially with different architectures and consistency models—to interact with shared memory using conventional load/store semantics [43]. This happens transparently because CXL attached memory is perceived akin to a separate NUMA node with slightly higher latency (50–100ns) [48], [57]. CXL multi-host coherent memory enables the development of distributed, heterogeneous applications.

Despite CXL's promising capabilities and its seemingly simple memory abstraction, enabling coherent memory sharing across multiple heterogeneous hosts, such as CPUs with different ISAs (x86, Arm, RISC-V) or accelerators (GPUs, FPGAs, TPUs), presents major challenges. To date, to the best of our knowledge, no hardware platform supports multi-host coherence CXL—not even for homogeneous systems. We identify two key unresolved challenges that must be addressed to ensure correctness and efficiency.

The first challenge is bridging the *semantic gap* between cache coherence (CC) protocols. Each compute architecture implements its own CC protocol—complex, performance-critical state machines tightly integrated with the processor and memory subsystems. Extending them to support CXL as a single, unified coherence protocol would require disruptive redesigns and significant verification effort, with unclear performance and broader applicability trade-offs. A more practical alternative is to retain each host's existing CC protocol, use CXL for inter-host coherence, and introduce specialized translation logic to reconcile differences between the host and CXL protocols. However, even in the simplest case—integrating MESI-based hosts with CXL's MESI-like protocol—subtle differences lead to a combinatorial explosion of states, making the translation logic highly complicated. Designing this logic in a manageable and correct manner requires a systematic approach, with clear design rules to ensure both correctness and performance.

The second challenge is handling heterogeneous memory consistency models (MCMs), as different compute ar-

[*]Authors contributed equally

chitectures implement distinct memory models with varying guarantees—such as TSO for x86 or relaxed models for Arm [5], [69]. Preserving each architecture's MCM is critical for correctness, and coordinating these models across heterogeneous hosts that share memory through CXL introduces significant complexity and programmability challenges. While prior work has addressed the challenge to combine different MCMs into a unified global model [31], [68], these approaches fall short for CXL because they are either incompatible with the dynamic nature of reconfigurable CXL systems [68], or are too abstract and lack critical specific rules for directly applying them to a complex CXL system [31].

To overcome these two challenges, we introduce C$^3$: *CXL Coherence Controller*, a hardware component that sits at the intersection of a host's CC protocol and the CXL CC protocol. C$^3$ systematically bridges the semantic and memory consistency gaps across heterogeneous CC domains.

C$^3$ is based on the theoretical foundation of compound MCMs [31]. From this abstract model, we derive two concrete, implementation-aware design rules that account for the distributed and asynchronous nature of CXL systems. The first rule (*Flow Delegation*) defines when memory operations must be forwarded between CC domains to ensure global visibility and consistency. The second rule (*Atomicity*) guarantees that no coherence effects are produced in the origin domain before the completion of a forwarded operation is observed. This implies other memory operations are logically stalled, guaranteeing that all processors observe the same global order of memory requests. Following these rules ensures that the bridged system preserves the high-level axioms of the compound MCM, making C$^3$ a concrete implementation of compound MCMs for CXL systems. Notably, C$^3$ enforces these rules by relying entirely on the native flows of the combined CC protocols, allowing it to bridge protocols without modifying existing coherence state machines or requiring intrusive protocol changes.

We implement C$^3$ as a generic gem5 model that can be instantiated for different combinations of host CC protocols and CXL, allowing us to evaluate the correctness, generality, and performance of C$^3$. Using formal verification and comprehensive litmus testing, we validate that C$^3$ correctly preserves the semantics of each host's MCM and CC protocol across multiple combinations of CC protocols and architectures. Finally, using a wide range of workloads, we show that C$^3$ introduces minimal performance overhead of 3.8-25.4% (average 5.5%) compared to a native system without CXL.

In summary, we make the following contributions:

- A generic and systematic methodology to combine arbitrary host-level cache coherence protocols with CXL through well-defined design rules.
- A generic gem5 model of C$^3$ that enables simulating CXL systems with heterogeneous host cache coherence protocols.
- A hierarchical `CXL.mem` protocol implementation in gem5 for CXL 3.0, made available as open source.
- Correctness verification of our approach using formal methods and litmus tests, demonstrating that memory consistency semantics are preserved across heterogeneous systems.

## II. Background

### A. CXL Remote Memory

Compute Express Links (CXLs) [23], [24], [26] is an open standard for high-speed, efficient communication between host processors and peripherals like accelerators, GPUs, and memory expanders. CXL defines two cache coherence protocols: `CXL.cache` and `CXL.mem`. We base our work on `CXL.mem` and the recently introduced support in CXL 3.0 for multi-headed memory devices and device-initiated invalidation flows. At a high level, this now enables multiple hosts to access and share the same memory device, with standard load and store instructions. This results in a transparent, hardware-managed, cache-coherent remote shared memory — an appealing abstraction for building distributed applications. Given CXL's architecture independence, future applications will likely leverage this multi-host coherent memory to optimally combine diverse hosts for specific tasks. To our knowledge, no prior work has investigated the implications of CXL's multi-host coherent memory in heterogeneous environments.

### B. Memory Consistency Models

Memory consistency models (MCMs) dictate the apparent execution order of memory operations, crucial for correct parallel program behavior [39]. Strong models such as Sequential Consistency (SC) prohibit memory reordering, making them intuitive but performance-limiting. This leads modern CPUs to adopt relaxed MCMs like x86's Total Store Order (TSO), which allows only store-load reordering for different memory locations [78]. In contrast, weak MCMs, like the one used in Arm CPUs, by default allow all memory accesses to be reordered, requiring the program to use explicit barriers when ordering is required [13]. Cache coherence protocols are pivotal in enforcing these models, as they define how memory updates propagate across caches, cores, and physical memory.

In heterogeneous CXL systems, where hosts with differing MCMs access shared memory, establishing a suitable system-wide MCM poses significant challenges. A model that is too weak could violate the native memory ordering guarantees expected by each host, causing program inconsistencies and breaking compiler mappings[1]. Conversely, an overly strong model (like universal SC) would severely penalize architectures designed for weaker consistency. The system-wide model must be appropriately balanced — strong enough to maintain program correctness across all architectures, while preserving individual performance characteristics. Furthermore, any solution must be scalable: integrate with new hosts and different MCMs without fundamental coherence system redesign.

Recently proposed compound memory models (CMMs) [31], [63], [68] address this challenge by combining distinct heterogeneous MCMs into a system-wide MCM

[1]Compiler mappings define how programming language-level synchronization primitives map to memory operations and instructions on the target hardware [87].

| Message | Dir. | MESI Eq. | Description |
|---|---|---|---|
| `MemRd,A` | M2S | `GetM` | Read memory and acquire excl. ownership |
| `MemRd,S` | M2S | `GetS` | Read memory and acquire sharable copy |
| `MemWr,I` | M2S | `WB+PutX` | Writeback, do not keep cachable copy |
| `MemWr,S` | M2S | `WB` | Writeback, retain current copy and state |
| `BISnpData` | S2M | `Fwd-GetS` | Device request sharable copy from host |
| `BISnpInv` | S2M | `Fwd-GetM` | Device request exclusive cachable copy |

TABLE I: Most relevant `CXL.mem` coherence messages and their equivalents in the MESI protocol. Messages consist of opcode and meta value. Dir. indicates the message flow direction: M2S (Host-to-Device) or S2M (Device-to-Host). MESI Eq. shows corresponding messages in the MESI protocol [63].

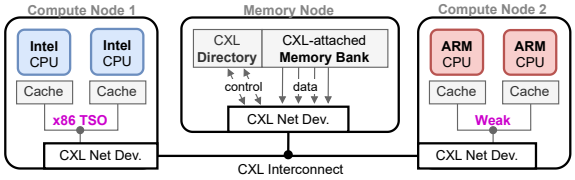



Fig. 1: Remote shared memory exposed by CXL to distinct architectures.

where each thread's native ordering constraints propagate to the global level. This ensures existing software remains correct without modification or re-compilation. CMM formally defines a compositional operational model for the propagation and serialization of memory requests across threads, which guarantees that each thread's local memory order is preserved in the global memory consistency, irrespective of other (heterogeneous) system threads. However, CMM reasons in an abstract framework where memory operations propagate atomically between two threads, without addressing the practical challenges of request concurrency and heterogeneity in real-world distributed coherence protocols. This work demonstrates how to concretely realize CMM principles for heterogeneous MCMs via CXL cache coherence protocol.

### C. Cache Coherence Protocols

Cache coherence (CC) protocols in multi-core systems maintain data consistency across core caches. They also play a fundamental part in MCMs, as they organize cache-to-memory exchanges in responses to core memory operations (*load, store, eviction, fence*). This work focuses on directory-based protocols due to their better scalability in large systems [63]. CPUs commonly use MESI variants [11], [42] (with states like Modified, Exclusive, Shared, Invalid), and extensions like Intel's MESIF or AMD's MOESI. ARM's CHI protocol [15] is another MESI variant with corresponding MOESI states, including an Owner state for direct dirty data sharing [39].

The `CXL.mem` CC protocol is also MESI-based, using the same stable states but with subtle differences in transition definitions and transient states. Tab. I summarizes key `CXL.mem` coherence messages and their MESI equivalents [2].

A key difference is the presence of a *conflict resolution* handshake (`BIConflict`), that we detail in Sec. III-A. MESI-family protocols enforce the Single-Writer-Multiple-Reader (SWMR) invariant by requesting sharer invalidation on writes. This mechanism suits strong MCMs like x86-TSO and latency-sensitive CPUs that require immediate global ordering of memory updates, at the cost of added complexity[3]. In contrast, architectures with weaker consistency models like GP-GPUs using Release Consistency Coherence (RCC) [30], [63] that leverage self-invalidation (e.g., release operations) to guarantee global visibility. This reduces overhead and improves bandwidth, ideal for throughput-oriented GPUs [7], [76]. For example, RCC requires explicit memory fencing instructions (`Load-Acquire`, `Store-Release`) to make accesses globally visible, unlike MESI's implicit guarantees coming from the SWMR invariant.

[2]The full list of CXL messages is in Appendix C of the CXL 3.1 spec. [24].

[3]While x86-TSO benefits from sharer invalidation before every write to provide immediate global order, MESI implementations in weaker MCMs like Arm or RISC-V (without a global write order) may choose to weaken the SWMR invariant by, for instance, allowing cores to immediately acknowledge invalidation but delay its processing until an explicit load fences [58], [59].

## III. Heterogeneous CXL Systems

The CXL standard promises interoperability between any devices that adhere to the CXL specification—CPUs (x86, Arm, RISC-V), GPUs, FPGAs, accelerators, and memory devices. Fig. 1 depicts a simple heterogeneous configuration where two compute nodes—an Intel-x86 node and an Arm node—share a common memory pool through CXL. In this setting, application threads on both compute nodes can transparently perform concurrent accesses to the same cache-coherent disaggregated memory pool.

However, CXL does not specify how to integrate these different architectures to guarantee correct and consistent behavior. Each employs its own MCM and CC protocol, making integration with CXL's CC protocol a non-trivial task — multiple architectures are combined in a single system. In particular, we identified two unresolved challenges: (1) bridging the semantic gap between CXL and diverse hosts CC protocols, and (2) compounding their heterogeneous MCMs.

### A. Semantic Gap of Cache Coherence Protocols

The first challenge arises from significant differences between textbook CC protocols and CXL's coherence mechanism. Textbook CC protocols are designed for on-chip networks where vendors have complete control and understanding of the network topology at design time. In contrast, CXL operates in a fundamentally different environment; an off-chip network running on top of PCIe, where it must contend with message reordering, higher latencies, and dynamically changing topologies, where devices from varied vendors can be added or removed at runtime.

For instance, a semantic gap exists even between `CXL.mem` and textbook MESI [63], although they share the same stable states, because their transaction flows significantly differ. Notably, CXL uniquely handles coherence message races. While MESI permits multiple simultaneous transactions and relies on cache controllers to infer serialization order from message

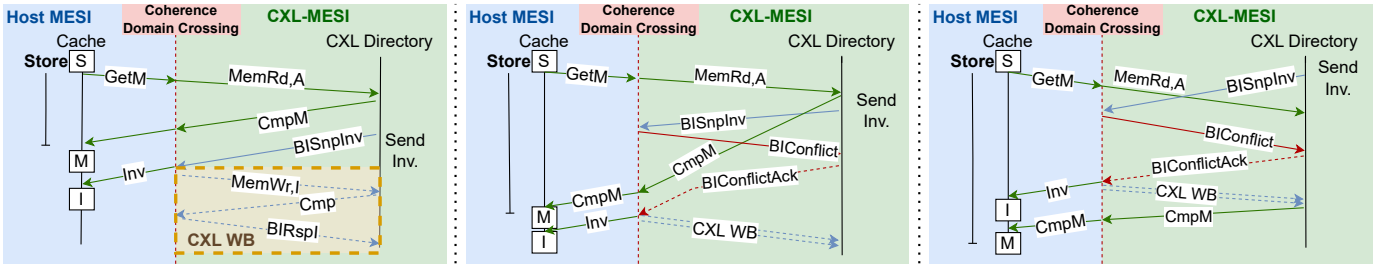



Fig. 2: CXL coherence flow examples. All show the same store operation from a host to the CXL memory racing with an invalidation snoop from the CXL directory. **(Left)** Normal case without reordering. **(Middle)** The completion message is delayed and reordered with the invalidation, creating ambiguity and requiring a handshake. **(Right)** The CXL directory and host process receive the messages in different orders, requiring a handshake to agree on the same order.

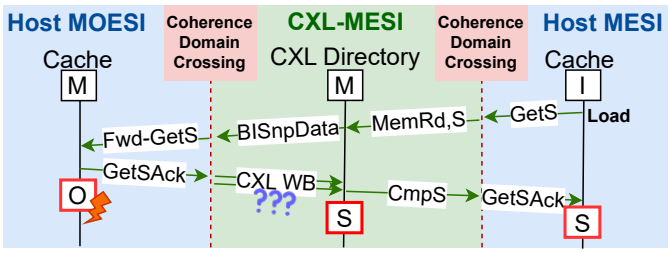



Fig. 3: Example of an inconsistent state between a MOESI cluster and CXL. The cluster subsists in O-state (dirty sharer) when the global CXL directory assumes S-state (clean sharer).

arrivals [66], CXL employs explicit conflict resolution within its coherence flow. If a host awaiting a completion message (`CmpM`) for a pending request observes an invalidation snoop (`BISnpInv`) from the CXL directory, it cannot determine in which order the directory processed these requests. To resolve this ambiguity, the host initiates a handshake by sending a `BIConflict` message to the CXL directory, which replies with `BIConflictAck` that cannot be reordered with the completion message.

Figure 2 illustrates three scenarios of this mechanism in action. In each case, a host in the shared state (S) initiates a store operation by sending a `GetM` request (translated to `MemRd,A` in CXL). At the same time, the CXL directory issues an invalidation snoop (`BISnpInv`) to the same cache line due to another host's activity.

In the **left scenario**, message ordering is preserved: the directory processes the requests in the same sequence they were sent, and the host receives the completion (`Cmp-M`) before the invalidation. The host first performs its write and then a write-back (`CXL WB`[4]) upon receiving the invalidation.

The **middle scenario** shows the completion message being delayed, causing the invalidation to arrive first. This creates an ordering ambiguity, requiring the host to initiate a handshake. Because the completion arrives before the handshake acknowledgment, the host determines it should complete the write before the invalidation.

[4]In the following, we use `CXL WB` to refer to the whole write-back sequence

In the **right scenario**, the directory processes the invalidation first. After the handshake, the host determines it must invalidate its internal caches before eventually receiving permission to write.

These examples demonstrate that despite similar stable states, the semantic differences between MESI-host and CXL memory protocols create significant integration challenges. While some messages translate directly (e.g., `GetM` to `MemRd,A`), others require context-sensitive translation depending on both protocols' states. Notably, the same invalidation message (`BISnpInv`) triggers entirely different action sequences in each scenario.

The complexity grows further in heterogeneous systems where diverse protocols like MESIF (x86 CPUs), MOESI (Arm CPUs), or RCC (GPUs) connect via CXL. Consider Fig. 3 where a MOESI cluster receives a `BISnpData` message to grant read permissions. In the textbook MOESI, this corresponds to a `Fwd-GetS` message, which prompts the cache with the modified line to send data to the requester and downgrade to the O-state. However, MESI lacks an O-state and expects a CXL writeback, creating a mismatch in protocol definitions. Supporting the required writeback would not only necessitate modifications to the original MOESI protocol, but also further lead to inconsistent system states—the MOESI host enters the O-state while the CXL network and MESI hosts enter the S-state. The inconsistency makes it unclear how subsequent operations should be handled, as the MOESI host believes it holds dirty data requiring future writeback, while other components assume they operate on clean data.

Thus, combining protocols is challenging as it requires concurrently tracking the state of both protocols, and ad hoc solutions are likely to introduce memory consistency bugs that are notoriously difficult to detect and debug.

**Requirement #1.** We need a systematic approach with clear rules to design a translation logic that overcomes the complexity of combining different CC protocols. This logic must be *correct* by construction, *generic* enough to accommodate any architecture, and *non-intrusive* to existing architectures.

**Research gap w.r.t. the state-of-the-art.** Prior works have proposed solutions to combine different CC protocols [21], [31], [67], [68]. Unfortunately, these approaches are not

suitable for CXL systems. HeteroGen [68] proposes to join multiple CC protocols by fusing the state machines of their directory controllers into a unified one. However, this approach requires knowing the entire system a priori, defeating CXL's dynamic topologies and flexibility in adding/removing devices.

HieraGen [67] proposes a hierarchical approach to protocol composition, which is architecturally well-suited to CXL's dynamic and modular topology. However, HieraGen is not designed to support the conflict resolution transactions required by CXL. Specifically, it assumes that every snoop initiated by CXL must be made visible and fully resolved in the local cluster before issuing a global response.

Recently proposed, compound memory models [31] introduces a compositional approach for reasoning about memory consistency in heterogeneous systems. They define an abstract propagation model of memory operations, that enables different MCMs to interoperate while preserving each domain's local semantics. These models are particularly promising for fulfilling requirement #1, as they define a set of rules that guarantee correct global behavior when composing diverse system architectures. However, these rules are specified in an abstract framework that does not capture the full complexity found in a hierarchical composition of CC protocols. For example, while compound models describe how operations must appear to be ordered from a programmer's perspective, they do not specify how coherence agents (e.g., a host and the CXL directory) should resolve message races when they observe events in different orders, in different CC domains. Discrepancies, as illustrated in Fig. 2 (right), are common in CXL due to its switch-based interconnect and unordered message delivery, which require explicit handshaking (e.g., `BIConflict`/`BIConflictAck`) to resolve.

While compound memory models provide a sound theoretical basis, they must be augmented with implementation-level mechanisms and constraints that account for the realities of fabric-level coherence and transient state management in order to support full system correctness in CXL-based heterogeneous environments.

### B. Compounding Memory Consistency Models

The second challenge stems from the fact that each architecture defines its own MCM, imposing different constraints on compilers and programmers. For instance, x86 CPUs in Fig. 1 implement the relatively strict TSO memory model, which preserves most memory orderings by default. In contrast, Arm CPUs implement a weaker memory model that allows aggressive reordering unless explicit barriers are inserted. While modern software practices consider weak ordering and manual placement of fences to enable portable code, compilers ultimately decide which fences to maintain or elide to achieve correctness with minimal performance impact on the target architecture [71]. Thus, in a heterogeneous system with hardware threads assuming distinct MCMs, with explicit and implicit ordering constraints, reasoning about correct memory order becomes significantly more complex. The visibility of memory operations becomes a complicated function of the mixed MCMs and heterogeneous coherence protocols—such as TSO from Intel, weak ordering from Arm, and MESI-CXL-MOESI merged coherence protocol that connects all nodes.

**Requirement #2.** A heterogeneous CXL system needs a MCM that is *flexible* enough to accommodate diverse architectures while *maintaining compatibility* with existing software by preserving architecture-specific native MCM semantics. In detail, we want a program to expect the same concurrent behaviors and possible memory instruction re-orderings, irrespective of whether other machines access the same CXL memory region.

**Research gap w.r.t. the state-of-the-art.** Prior work has addressed MCM diversity in heterogeneous systems through two main approaches. The first approach is introducing new MCMs, as in Memglue [21], specifically for heterogeneous systems. However, this is impractical as it is incompatible with existing binaries, and necessitates complete recompilation and new compiler mappings or potential code changes, on top of new hardware shims between protocols.

The second approach is to fuse together MCMs specifically such that they both retain their original properties. Prior work proposes automated tools [31], [68] that combine diverse MCMs by synthesizing a merger of CC protocols, such that each architectures' MCM semantics are preserved without requiring code changes. While more promising, these solutions are either incompatible with CXL's dynamic nature or too abstract to address real CXL system complexities, as discussed in Sec. III-A. What is needed is a mechanism that enables the generation of concrete CC controller instances that operate in CXL systems, that provide by construction the same high-level guarantees as in compound memory models [31].

### C. Design Rules for Heterogeneous CXL Systems

*1) The case for a coherence controller:* To fulfill requirements #1 and #2, we propose the notion of a *Coherence Controller*—a specialized component that sits at the intersection of two CC domains and translates coherence requests between them. The goal of this controller is to abstract away the complexity of connecting heterogeneous CC protocols, hiding all the translation intricacies without requiring any changes to the existing protocols. This controller must be generic enough to accommodate diverse architectures while maintaining the correct memory consistency semantics within each protocol.

*2) Intuition:* To achieve our goals, we devise our coherence controllers specifically to always result in a compound memory model [31] when combining heterogeneous MCMs. Compound MCMs ensure two appealing properties: (1) preserve local MCM axioms within each cluster – maintaining the same ordering and atomicity guarantees as standalone systems, and (2) establish a global order across cluster boundaries (system-wide) of memory requests which become globally visible.

In a nutshell, within the abstract framework of [31], any two memory operations $(o, o')$ with an ordering constraint $(o \rightarrow o')$ must propagate in the same order to all affected threads ($o$ *propagates before* $o'$); and $o'$ must be stalled until $o$ completely

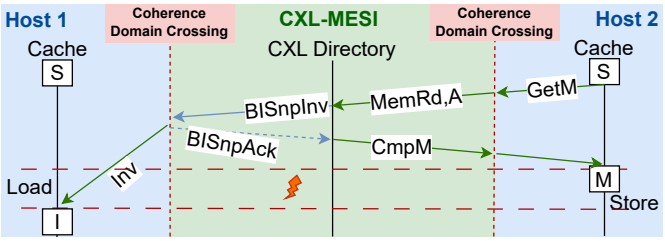



Fig. 4: Example of race condition by not obeying Rule II.

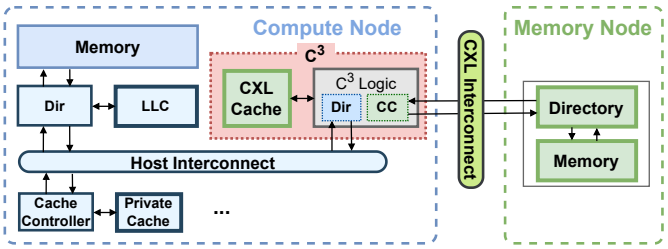



Fig. 5: C$^3$ architecture.

propagates. Overall, this ensures that propagation and visibility of operations follows their dependency order constraints.

We translate these abstract principles into two concrete design rules that define the interactions between coherence transactions in our controllers. A first rule to define *when* exactly coherence requests must be forwarded between clusters, and a second rule to ensure atomicity of requests propagation between clusters. When followed, both rules ensure the controller is correct by construction and realizes a compound MCM compatible with all participating architectures.

*3) **Rule I: Flow Delegation**:* All operations that cannot be satisfied locally or have globally visible effects must be forwarded and handled at the global level. Similarly, all global requests or *snoops* that affect the local domain must be forwarded and handled at the local level.

**Rationale:** For remote memory operations, the global directory is the only entity that can guarantee memory consistency across all compute nodes. It unilaterally decides on the serialization order of global requests and is able to grant and revoke permissions to different compute nodes. However, as the global and local levels do not talk the same "language", the global directory is unable to directly affect local caches. Instead, it must delegate all operations that require local actions to local coherence domains.

This rule guarantees that, first, the global level is always aware of operations made visible to other threads in local domains, such that it can take appropriate actions to ensure their global consistency—e.g., retrieving the latest copy upon a read request or invalidating other sharers upon a write request. Second, that the global level can enforce coherence actions without modifications to the local CC protocols.

**Example:** Consider multiple hosts sharing data across CXL (all in S-state) when one host attempts to modify the data. Simply acknowledging the request locally without first propagating it globally to invalidate other sharers breaks MESI's SWMR guarantees, causing other hosts to operate on stale data.

*4) **Rule II: Atomicity**:* Upon forwarding a request to another protocol domain, irrespective of the direction (local to global or global to local), the bridge must not produce any coherence effects in the origin protocol domain before observing completion in the target domain.

**Rationale:** By preventing coherence actions in the origin protocol domain (effectively stalling) until receiving a completion message from the secondary domain, this rule guarantees all forwarded requests appear atomic in the origin domain. Otherwise, the system may violate causality or multi-copy atomicity–the property where writes propagate to all cores simultaneously— potentially breaking the memory consistency guarantees in the origin domain.

**Example:** In Fig. 4, the remote memory sends an invalidation to Host 1 to be able to grant Host 2 write permissions. Violating Rule II, by having the controller immediately respond with creates a race condition between when Host 2 receives the `GetM` completion and writes the value and when Host 1 has invalidated all caches.

## IV. C$^3$: CXL Coherence Controller

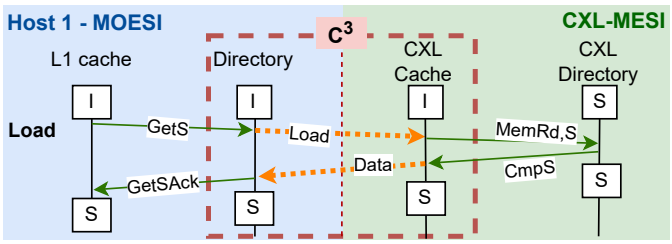
Building on the two previous rules, we present C$^3$, a generic CXL coherence controller that bridges the semantic gap between arbitrary host CC protocols and CXL, while maintaining its MCM.

C$^3$ sits at the junction of two cache coherence domains—in our case, a host protocol (local) and the CXL protocol (global)—and is responsible for translating and forwarding coherence flows crossing the protocols' boundary. The main idea of C$^3$ is to nest coherence transactions from one domain into the other and produce the coherence effects required by either protocol using the native protocol flows of the other domain. This allows C$^3$ to remain generic and be fully interoperable with legacy memory subsystems, as it requires no modification to the existing caches or directory controllers of hosts or memory devices. All logic to translate flows between the two coherence domains is confined within C$^3$ itself, enabling easy drop-in integration within existing systems.

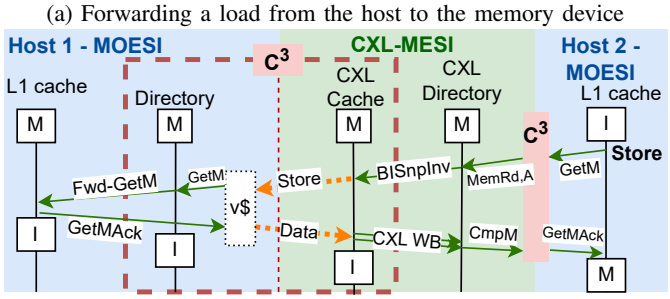
C$^3$ connects heterogeneous hosts via `CXL.mem` and CXL standard 3.0, which enables symmetrical coherence between hosts and is applicable to CXL HDM-DB (supported by Type 2 and Type 3 devices). However, we note that our design principles are generic and C$^3$ can be adapted for other CXL revisions as well as serve as a blueprint for other interconnects such as NVLink Fusion [1].

### A. Overview

Fig. 5 shows the high-level architecture of C$^3$ connecting a compute node to a remote memory device via CXL. C$^3$ consists of two main components: the *CXL cache* and the *C$^3$-logic*. The CXL cache represents the analog part of the shared last-level cache (LLC) but instead of caching data from the local memory, it holds copies of data mapped to the remote memory region. We note that for simplicity, we consider the




(a) Forwarding a load from the host to the memory device




(b) Forwarding a store from the memory device to the host. The C$^3$ instance for host 2 is simplified for brevity.

Fig. 6: Conceptual operations C$^3$ performs to map coherence flows between coherence domains.

CXL cache as a separate entity; in practice, however, it may be part of the LLC (e.g., one or more slices of the LLC [96]).

The C$^3$-logic is the core of C$^3$ and is responsible for translating coherence flows between the host and CXL domains. Conceptually, it combines the functionality of a *local* directory controller (Dir) and a *global* cache controller (CC). That is, from within the host, C$^3$ appears as a directory controller responsible for managing coherence for data mapped to the remote memory region, and from the perspective of the memory device, it functions as an ordinary cache controller.

### B. The Compound State Machine

The entirety of C$^3$-logic is in its finite state machine (FSM), a fusion of FSMs of the local directory controller and the global cache controller. To construct C$^3$'s FSM in a generic way while obeying the two rules described in Sec. III-C, we rely on three key concepts: (1) *coherence flow delegation*, (2) *transaction nesting*, and (3) *state compounding*.

**Coherence flow delegation.** Recall from Sec. III-C that Rule I requires all requests to be forwarded when they cannot be satisfied within the origin cache coherence domain. To achieve this in a generic way without disruptive modifications in the existing system, C$^3$ translates each coherence request from one domain to a corresponding coherence flow of the other domain. Conceptually, this is done by "simulating" the original core accesses (`load`/`store`) that triggered a certain coherence transaction in the other protocol domain.

To illustrate this concept, Fig. 6 shows two scenarios. In the first scenario (Fig. 6a), C$^3$ receives a `GetS` from one of the host's caches using the MOESI protocol. C$^3$ translates this request into the CXL-MESI flow that would be triggered as if the directory were performing a *load* to the CXL cache. In the second scenario (Fig. 6b), C$^3$ receives a `BISnpInv` message from the CXL directory in response to the store operation from host 2. This time C$^3$ translates the message into the MOESI flow that corresponds to a *store* to a virtual cache sitting between the CXL cache and C$^3$'s directory. It's important to highlight that simulating core accesses is a conceptual representation and the actual translation mappings are statically backed into C$^3$'s FSM at synthesis time. However, as loads and stores are universal to any CC protocol, this representation allows reasoning about C$^3$'s translation behavior in a protocol-agnostic way.

**Transaction nesting.** Rule II requires that all forwarded requests must appear atomic to the origin cache coherence domain to preserve protocol invariants. This is achieved by strictly nesting the forwarded transaction within the original transaction: C$^3$ stalls requests to the same cache line from the origin domain until the forwarded transaction completes. In practice, C$^3$ consumes the origin request, forwards it to the other domain, and enters a transient state that progresses only upon receiving CXL completion messages or snoops.

For instance, in Fig. 6a, C$^3$ stalls host requests to the same cache line until receiving the completion message for the forwarded CXL load, while remaining able to process other CXL requests and snoops. Such situations are shown in Fig. 2 (Sec. III), where C$^3$ receives invalidations before the completion message for `MemRd,A`. To resolve these cases, C$^3$ performs a handshake to disambiguate race conditions between forwarded requests and incoming snoops.

**State compounding.** As described above, C$^3$'s FSM is practically a fusion of the FSMs of the local directory controller and the global CXL cache controller. This means that the FSM's states represent the combined (Cartesian) product of the states from the local and global CC protocols (stable and transient states), allowing C$^3$ to maintain the view and state of cache lines for both protocols simultaneously. For instance, with MOESI protocols in both the local and global domain, the state pair $(I, I)$ denotes that the cache line is absent from the host, and the pair $(I, S)$ indicates that C$^3$'s CXL cache holds a clean copy although none of the host private caches shares it.

Monitoring both the local and global state allows C$^3$ to decide whether a cross-domain transaction is necessary as required by Rule I. In the scenario depicted in Fig. 6b, the state combination $(M, M)$ requires forwarding, while a state combination like $(I, M)$ would not, and the CXL cache can reply immediately to the CXL directory.

Note that, enforced by the two design rules, many state combinations are never reachable. For example, as Rule I requires globally visible effects to be forwarded, the CXL cache must remain inclusive of all remote data cached in the host caches. Otherwise, the CXL directory would not know if the host has a copy of data that needs to be invalidated upon receiving a `MemRd,A` request. Thus, state combinations like $(S, I)$ or $(M, I)$ must never be reachable which is ensured

| Message | $S$ | X-Access | Action | $S_{Next}$ |
|---|---|---|---|---|
| BISnpInv | $M,M$ | Store | Fwd-GetM to Host $ | $MI^A,MI^A$ |
| BISnpInv | $I,M$ | — | Data to CXL Dir | $I,I$ |
| Any | $MI^A,MI^A$ | — | Block | $MI^A,MI^A$ |
| BISnpData | $M,M$ | Load | Fwd-GetS to Host $ | $MS^{AD},MS^{AD}$ |
| ... | | | | |

TABLE II: Fragment of C$^3$'s translation table for incoming CXL directory messages. $S$ represents the current compound state, X-Access indicates the conceptual cross-domain access, and $S_{Next}$ shows the resulting state after the transition.

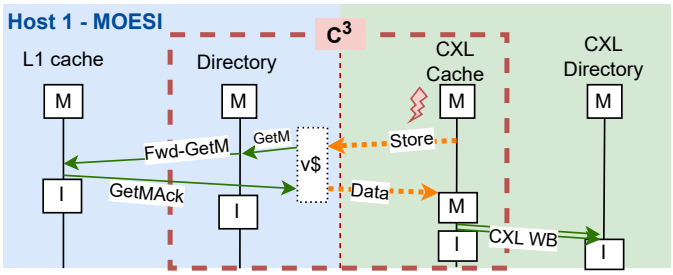



Fig. 7: Conceptual operations of a CXL cache eviction.

during construction of the C$^3$-logic [5]

### C. Flow Translation

To translate flows between coherence domains, we make two enabling observations. First, dynamic message information (e.g., *address, tag, senderID, destinationID*) is protocol-agnostic and can be passed by value. Second, message translations between domains are deterministic and can be statically pre-computed.

Based on these observations, C$^3$ uses translation tables that map incoming messages and the current compound state to the corresponding cross-domain access and resulting nested coherence flow. Tab. II shows for the situation in Fig. 6b a fragment of these translations for incoming CXL directory messages. For example, when receiving a `BISnpInv` in state $(M, M)$, C$^3$ interprets this as a conceptual *store* operation that needs to be propagated to the host cache hierarchy. This triggers a `Fwd-GetM` message to the host caches and transitions C$^3$ to the transient state $(MI^A, M)$ (wait for local acknowledgment). Conversely, the same message in state $(I, M)$ requires no host involvement and can be directly satisfied with a writeback to the CXL directory.

We describe in Sec. V how these translation tables are generated automatically during synthesis. These pre-computed translation rules are then embedded directly into the C$^3$-logic, introducing no runtime overhead while ensuring correctness through strictly enforcing the two design rules outlined earlier.

[5]Local protocols with self-invalidation, such as RCC, may let host caches temporarily hold stale data, which appears to violate C$^3$ 's inclusion. This occurs because CXL invalidations (from the CXL directory or self-evictions) do not update host caches. However, RCC restores inclusion at each *release* / *acquire* via self-invalidation, which is the intended behaviour and consistent with the assumptions of RCC that programmers perform explicit synchronization to avoid stale data.

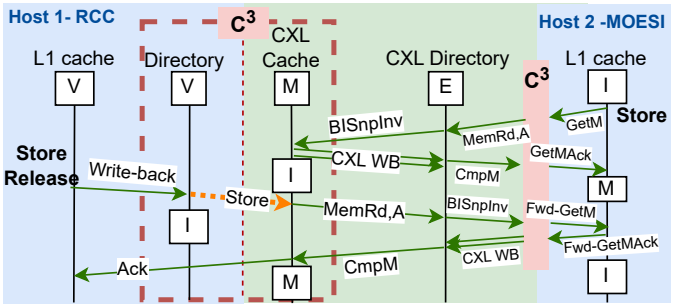



Fig. 8: Example operations with RCC-CXL Bridge. The C$^3$ instance for host 2 is simplified for brevity.

### D. Discussion

*1) CXL Cache Evictions:* To ensure that the CXL cache stays inclusive, C$^3$ must also handle evictions as cross-domain communications. Fig. 7 shows the self-eviction of a cache line where the bridge is in the state $(M, M)$ meaning that one of the host private caches still holds a dirty copy of the data which must be reclaimed before evicting the block to memory. To force the invalidation, C$^3$ mimics a store access in the hosts coherence domain triggering the host cache writeback. Upon receiving the data C$^3$ can proceed with normal eviction by going through the CXL writeback sequence (as shown in the first flow of Fig. 2) with the CXL directory. Once the eviction completes, and a free slot is available in the CXL cache, C$^3$ will proceed with the initial request that caused the eviction.

*2) RCC:* Relaxed consistency protocols like RCC represent an interesting case where the CXL cache is not kept strictly inclusive with host caches. In RCC, C$^3$ can directly respond to invalidations from the CXL directory without host cache involvement; it is the responsibility of host caches to self-invalidate to synchronize with C$^3$'s CXL cache. C$^3$ only forwards requests from the host to CXL when the host accesses uncached data or executes explicit synchronization instructions like *store-release* or *load-acquire*. Nevertheless, C$^3$ still ensures that the CXL cache stays coherent with remote memory, such that during synchronization events, the host never operates with stale data and properly synchronizes globally with CXL as well.

Fig. 8 illustrates a store-release operation in RCC where a CXL cache is in an invalid state $(I)$ due to a previous store operation from another host. Thus, before responding to the host's store-release request, C$^3$ must first acquire the most recent data and write permissions from remote memory.

*3) Memory Barriers:* Memory barriers primarily affect the CPU by enforcing ordering constraints within the core pipeline. C$^3$ does not directly handle barriers, but indirectly through the coherence messages and events that the core generates to implement a barrier. For instance, with SWMR protocols such as MESI, barriers translate into regular loads and stores where the core awaits for completion – which C$^3$ handles similarly to regular coherence messages. In weaker protocols like RCC, where barriers may need to propagate, cores either translate them into existing cache maintenance

events (flush, invalidate) or specific coherence messages (like RCC's load-acquire or store-release messages), which C$^3$ translates and forwards to CXL, as described in Sec. IV-D2.

*4) Hardware Complexity and Integration:* C$^3$ 's hardware consists of two main components: the CXL cache and the C$^3$-logic, as shown in Fig. 5. The primary area overhead originates from the CXL cache, which must be inclusive of all CXL data cached by a host. While the above sections describe it as a dedicated cache for clarity, in practice, it can be integrated with the LLC. For instance, existing CXL-enabled platforms from Intel (SPR/EMR processors) already couple LLC slices with the CXL *Caching and Home Agent* (CHA) [26], [50].

The C$^3$-logic implements the fused host–CXL cache coherence protocols as a finite-state machine. Although the synthesis process involves generating correct request translations, stable and transients states, and protocol transitions; the resulting hardware is purely combinational and sequential logic, incurring minimal area and power overhead. The translation tables described in Sec. IV-C are purely conceptual, used only by the generator to produce the final state transitions. Thus, the synthesized hardware does not require table lookup, as all translations are embedded directly into the generated FSM. Even though C$^3$ enables heterogeneous translations of requests, the complexity of its logic and its controller latency are comparable to other conventional hierarchical coherence controllers, such as those used in Arm's CHI protocol [14] or those local coherence directories in multi-socket platforms.

C$^3$ is fully backward compatible with earlier CXL standards and can be integrated into existing CXL implementations with minimal effort. In Intel SMR/EMR platforms, only the controller logic in the CXL CHA needs to be extended with C$^3$'s stateful coherence logic to support multi-host coherence. In hybrid memory configurations, C$^3$ handles remote CXL coherence traffic while local traffic routes to existing controllers without additional modification.

## V. Methodology

**Implementation.** We implement C$^3$ as a generic model for gem5, a cycle-approximate simulator widely adopted in computer architecture research [20], [29], [54]. Its detailed memory subsystem, Ruby [53], provides an ideal platform for evaluating various CC protocols and their interactions with CXL memory devices. The gem5 simulator employs SLICC [28], a domain-specific language for modeling cache coherence protocols at an abstract level. While manually implementing specific protocol combinations in SLICC is possible, our goal is to create a generic solution that can accommodate various host and device protocols.

To enable generality and to be able to support arbitrary input protocols, we developed a generator tool [47] that takes machine-readable stable state protocol (SSP) specifications [66] for both host and CXL CC protocols as input, merges them, and outputs SLICC code for C$^3$. The tool generates the C$^3$-logic connecting the host's cache and directory controller with the CXL directory controller.

| | |
|---|---|
| Cores | 8-30 cores[6], 2 GHz, x86/Arm, 8-wide OoO, 192 ROB |
| L1 cache | 128 KiB, 8-way, private, LRU, 1 cycle latency |
| LLC | 4 MB, 8-way, shared, inclusive, LRU |
| Intra-cluster Interconnect | point-to-point topology, static routing, 72B per flit, 1 cycle router latency, 10 cycle link latency |
| Cross-cluster Interconnect | star topology, static routing, 256B per flit, 1 cycle router latency, 70 ns link latency |
| CXL Memory | DDR5, 4400 Mhz, 1-channel, 10 ns latency |

TABLE III: Simulated system parameters.

The generator's front-end is based on Protogen [66], which parses SSP specifications into an intermediate representation (IR) and generates concurrent FSMs with all stable states for each input protocol. Next, the tool generates translation tables by systematically traversing the FSMs of both local and global protocols, identifying the specific coherence actions required by C$^3$ for each input message and state combination. When Rule I requires a cross-domain access, the corresponding nested flow is identified by simulating the core access that would trigger an equivalent action in the target domain. Using these translation tables, the tool merges the two FSMs into a single compound FSM, which is then analyzed to remove all forbidden states as specified by Rule II. Finally, the tool generates the complete SLICC code for C$^3$.

**Simulation Environment.** We use gem5 version 23.1 [20], [29], [54] in syscall emulation mode (SE) and our previously described tool to generate SLICC implementations of C$^3$ for various protocol combinations in gem5's Ruby system. Our tool has one current limitation: it does not support separate instruction and data caches[7]. Therefore, we simulate a common private cache for both instructions and data per core. To make simulations tractable in a reasonable timeframe, we use small input sizes and scale the cache sizes and number of cores for each workload to achieve a similar number of misses per kilo-instructions (MPKI) as observed in real hardware experiments on an Intel Sapphire Rapids server [92].

We model a two-node heterogeneous system, mimicking the one depicted in Fig. 1, by splitting the cores into two clusters, each with its own shared last-level cache (LLC). C$^3$ replaces the LLC controller in each cluster and communicates with a CXL directory at a remote memory controller (Device coherency engine (DCOH)) through a high-latency link[8]. We rely on gem5's Garnet network model [19] to simulate communication between hosts and CXL memory, rather than using dedicated PCIe-based CXL simulation models [90]. Although Garnet was originally designed as an on-chip network and real CXL systems communicate over a PCIe fabric, Garnet is tailored for coherence protocols which aligns with our focus on protocol bridging. We use its flexible network configuration (link latency, bandwidth, flit size) to align with CXL topolo-

[6]The number of cores is calibrated for each workload to match approximately the same MPKI as observed on real hardware.

[7]Adding support for separate instruction and data caches is not a methodological problem but merely a matter of engineering effort.

[8]The link latency was determined empirically to match the CXL memory access latency of 400ns as reported by prior work [57]

gies. It lets us isolate performance effects stemming from protocol logic and C$^3$ from the PCIe transport overheads.

We deliberately evaluate a worst-case scenario with all data in remote CXL memory to maximize coherence traffic and stress-test C$^3$, while noting that a hybrid configuration, where only part of the data is remote, might be more practical.

To simulate different MCMs, we use gem5's `needsTSO` flag of the out-of-order core as an alternative to simulating different ISAs for different clusters. When the flag is enabled, the default for x86 CPUs, it enforces the TSO MCM. For Arm cores, the flag is disabled, allowing gem5 to model a weak MCM. This approach allows us to isolate performance differences attributable to the MCM, from those tied to ISA-specific implementation differences in the gem5 models.

Tab. III lists the complete details of the simulated system parameters. The implementation of our tool, the CXL controllers in gem5's SLICC code, and our experimental setup are publicly available (see the artifact appendix).

**Workloads.** We evaluate C$^3$ using 33 highly parallel applications from three benchmark suites: *Splash-4* [32], *PARSEC* [2] and *Phoenix* [73], which are widely used to evaluate concurrent workloads in multi-core scenarios.

## VI. Evaluation

We show C$^3$ to be correct, generic, and non-intrusive (in terms of changes to existing hardware and performance) while maintaining compatibility with each host's MCM.

### A. Correctness

**Formal Verification.** We verify the correctness of C$^3$'s FSMs and SLICC controllers. To verify C$^3$'s FSMs, we extend our generator tool with a backend that follows the same Mur$\varphi$-based formal verification methodology introduced in HeteroGen [68]: We use the herd7 [6] tool to generate litmus tests for sequential and relaxed consistency threads, including common checks like *IRIW, MP, 2+2W, CoRR1, CoRR2, LB, R, RWC, S, SB, WRC, WRW+2W, and WWC*. To map these to our heterogeneous setups, we consider all possible assignments of threads to sequential and relaxed consistency clusters. The litmus tests for the weaker MCM are refined by using ArMOR [55] to remove fences that are no longer required when combining with the stronger MCM, as proposed in [68]. Using the Mur$\varphi$ model checker, the FSMs were verified to never reach any forbidden outcome for a mix of cores from sequential and relaxed consistency clusters. This ensures that C$^3$ retains the local MCM of each host.

**Litmus Tests.** To increase confidence in the SLICC controllers, which realize the FSMs, we empirically evaluated litmus tests in our gem5 simulation environment. For this, we configured a 2-cluster setup with 8 ARM O3 cores per cluster. We distributed litmus test threads equally across two MESI clusters, connected via C$^3$ to CXL at the global level. Our evaluation included seven common litmus tests: *MP, IRIW, 2_2W, R, S, SB* and *LB*, generated using the herd7 tool [6] and partially based on prior work [31].

| Test | MESI-CXL-MESI | | | MESI-CXL-MOESI | | |
|---|---|---|---|---|---|---|
| | Arm-Arm | TSO-Arm | TSO-TSO | Arm-Arm | TSO-Arm | TSO-TSO |
| 2_2W-sys | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| IRIW-sys | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| LB-sys | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| MP-sys | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| R-sys | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| S-sys | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SB-sys | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

TABLE IV: Litmus test results for different protocol and MCM combination. The ✓symbol indicates no forbidden outcomes.

We executed each litmus test one hundred thousand times in gem5 for each of the following configurations: *(a)* same CC protocols and same MCMs in both clusters, *(b)* different CC protocols, same MCMs, *(c)* same CC protocols but different MCMs, *(d)* different protocols, different MCMs. In configurations *(a)* and *(b)*, across all tests, we encountered no forbidden outcomes under the host's local MCM, confirming that C$^3$ maintains consistency by enforcing appropriate orderings. To guarantee that the litmus tests will detect forbidden outcomes and that C$^3$ does not introduce stronger memory ordering guarantees than the compound memory model defines, we intentionally removed all synchronization primitives from the litmus tests. As expected, these modified tests produced at least one forbidden outcome in each case, serving as a control to verify that our tests don't invariably pass unconditionally.

In configurations *(c)* and *(d)* with heterogeneous MCMs, we run the same litmus tests with the `needsTSO` flag enabled on cores of one of the two clusters, thereby enforcing a stronger TSO MCM for these cores only. As expected, having a stronger MCM in one cluster does not change the litmus test results—all tests pass without exhibiting forbidden outcomes. Next, we want to validate C$^3$'s ability to reconcile heterogeneous MCMs, i.e., C$^3$ strictly propagates local memory guarantees to the global CXL memory. To this end, we run again the same litmus tests for *(c)* and *(d)*, while selectively removing memory fences on the threads mapped to the TSO cores. Since we know that TSO cores naturally enforce *store-store* memory order, litmus tests should not exhibit any forbidden outcomes when TSO cores execute threads without explicit *store-store* fences. In these runs, we observe no forbidden outcomes, as expected. For instance, in the *MP* litmus test, $thread\ \#1$ executes a series of stores that $thread\ \#2$ reads in reverse order. With memory fences disabled on $thread\ \#1$ on a TSO core, we observe no forbidden outcomes.

For our validation to be complete, we must also check whether forbidden outcomes can be observed when other types of fences are removed (e.g., *load-load* or *load-store*). As expected, when removing more fences from the TSO cores, we start observing forbidden outcomes. Similarly, when we disable synchronization primitives on the ARM cores, we also observe forbidden outcomes. For instance, in the *MP* test, if we disable the *acquire* event from $thread\ \#2$ on an ARM core, although a TSO $thread\ \#1$ provides strongly ordered stores, we can still observe reads out-of-order from the ARM core. Overall, this validates that C$^3$ strictly propagates guarantees

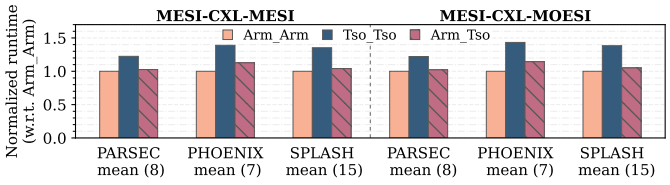



Fig. 9: Performance comparison of heterogeneous MCMs in two scenarios: homogeneous and heterogeneous CC protocols.

from each individual memory model, and does not violate or strengthen the global memory consistency defined by the compound MCM.

In Tab. IV, we summarize all combinations of CC protocols and MCMs tested for each litmus test. The first and last elements of each protocol combination represent the local coherence protocols of the two clusters, with CXL serving as the global protocol connecting them. A ✓symbol indicates all tests passed—meaning all allowed outcomes were observed without any forbidden outcomes. The consistently correct results across all protocol combinations and MCMs demonstrate that C$^3$ always propagates each cluster's memory consistency guarantees into the global memory, for every underlying local coherence protocol and memory model.

These empirical gem5 results, combined with HeteroGen's Murphi-based verification, confirm that C$^3$ always correctly bridges heterogeneous coherence protocols with CXL and successfully reconciliates heterogeneous hosts' local MCMs.

### B. Generality

Having validated that C$^3$ maintains the intended memory consistency guarantees, we evaluate now its general usability by running 33 parallel benchmarks from three different suites across diverse MCM and CC protocol combinations.

We evaluate C$^3$ across two dimensions. First, we test three different MCM combinations while keeping the CC protocol fixed to MESI-CXL-MESI: all cores implementing the ARM MCM, all cores implementing TSO, and a heterogeneous setup with the ARM MCM in the first cluster and TSO in the second cluster. Second, we change the CC protocol in the second cluster from MESI to MOESI to evaluate C$^3$ with mixed MCMs and mixed CC protocols simultaneously.

The left part of Fig. 9 shows the mean performance of each benchmark suite for the homogeneous MESI-CXL-MESI combination, normalized to the ARM-MCM setup. Switching from the weak ARM-MCM to the stronger TSO-MCM results in a 22-39% performance degradation. This degradation is expected and aligns with prior work on binary translation [34], [75], [77], where enforcing TSO on Arm architectures can impact performance by up to 75% (avg. 48%) for similar workloads [34][9].

In the mixed setup where only the second cluster implements TSO, the performance degradation is only 2.6-12.7%, demonstrating that C$^3$ efficiently bridges heterogeneous MCMs hindering performance of the weaker memory model.

The same trends hold when CC protocols also differ between clusters. In the right part of Fig. 9, using the strong TSO MCM in the heterogeneous MESI-CXL-MOESI protocol setup results in a degradation of 22-43% compared to the weak ARM MCM. With different MCMs and CC protocols in each cluster (ARM/TSO) results in only a 2.2-14.4% slowdown.

The key takeaway is that C$^3$ successfully and efficiently bridges arbitrary combinations of CC protocols and MCMs, even when both differ simultaneously between clusters.

### C. Performance

We evaluate C$^3$'s performance in more depth using the same 33 parallel benchmarks as in the previous section. For this experiment, we are running on x86 gem5 out-of-order models. Our primary focus in this section is on performance differences between CC protocol combinations, so we keep the MCM fixed. For fair comparison, we modify only the local and global protocols. Parameters for link latency, topology, and cluster configuration remain the same across all experiments.

Our baseline system (`MESI-MESI-MESI`) uses a homogeneous but hierarchical setup with MESI as both local and global protocols. In this configuration, C$^3$ functions as a passive device, simply forwarding inter-cluster coherence requests one-to-one between the local and global coherence domains.

We compare this all-MESI baseline against three alternative protocol combinations. In the first combination (`MESI-CXL-MESI`), we replace the global MESI protocol with CXL, representing a system where two homogeneous host clusters share remote memory via CXL. In this setup, C$^3$ must perform active protocol translation to communicate with the CXL directory. In the second and third combinations (`MESI-CXL-MOESI`, `MESI-CXL-MESIF`), the MESI protocol in the second cluster is replaced with MOESI and MESIF, respectively, creating truly heterogeneous systems with two different local CC protocols communicating via CXL at the global level.

Fig. 10 shows the execution time of all 33 parallel applications, normalized to the `MESI-MESI-MESI` baseline. Most benchmarks demonstrate limited sensitivity to different protocol combinations. However, switching the global protocol to CXL results in consistent slowdowns across all three heterogeneous protocols compared to the baseline, as seen in the `Mean` section of Fig. 10. In detail, the performance degradation for the configurations `MESI-CXL-MESI`, `MESI-CXL-MOESI`, and `MESI-CXL-MESIF` are respectively 4.0-26.6% (avg. 5.5%), 3.9-28.6% (avg. 5.7%), and 4.0-29.4% (avg. 5.5%). The *F* and *O* states provide intra-cluster optimizations, whose effect are dwarfed by the longer cross-cluster CXL latencies.

*1) Performance Analysis of CXL Slowdowns:* In Fig. 11, we show the cache miss latency breakdown by instruction types, grouped in 3 miss latency ranges, comparing `MESI-MESI-MESI` and `MESI-CXL-MESI` in 3 of the most impacted workloads (*histogram*, *barnes*, *lu-ncont*) with 19-25% more miss cycles, and one of the least impacted workloads (*vips*) with 2.2% more

[9]While binary translation enforces TSO through software fences and our approach through hardware, both approaches limit the same memory ordering optimizations that contribute to ARM's performance advantages.

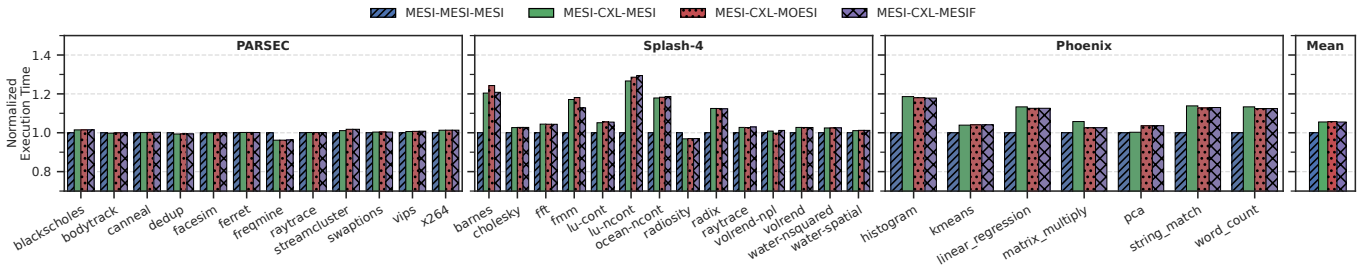



Fig. 10: Performance comparison of heterogeneous CC protocol combinations normalized to MESI-MESI-MESI baseline.

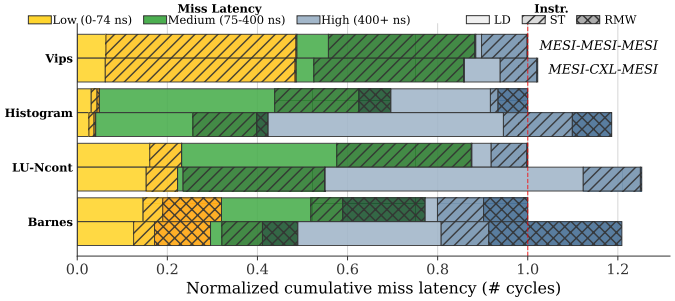



Fig. 11: Breakdown of total miss cycles by request latency and instruction. Selected workloads show surging high-latency accesses with CXL, while *vips* exhibits minimal sensitivity.

miss cycles. We see that the increase in miss latency directly correlates with the slowdown in Fig. 10. To understand the cause, we break the misses into three groups: *low* ($<$ 75ns), *medium* (75-400ns), and *high* ($>$ 400ns) latency misses. Since 400ns is the typical round-trip latency of memory requests, the three categories broadly map to: intra-cluster coherence transactions (L2 or LLC misses), CXL memory access, and cross-cluster coherence transactions. Affected workloads see an increase only in the *high*-range (remote-cluster) access by 2.9×, for stores or RMWs (read-modify-writes) and loads. Since the number of misses is unchanged for MESI and CXL, it indicates that cross-cluster coherence transactions are costlier with CXL, for both *read* and *write* requests.

For *write requests* (store, RMW) with remote cluster invalidation, CXL's overhead stems from a more complex transaction flow. MESI handles them in 3 remote message delays: `GetM` (cache to dir) → `Fwd_GetM` (dir to owner) → `GetM_Ack` (owner to cache). Additionally, the MESI dir can pipeline requests to the same address without waiting for any response. Conversely, CXL requires 6 remote message delays when the owner is dirty (4 when clean) with 2 blocking transient states at the directory (cf. Fig. 3), preventing pipelining and doubling message complexity compared to MESI.

For *read requests* (loads) with remote owner invalidation, CXL also has higher complexity (4 vs 3 message delays), but it mainly suffers from *convoy effect* from the blocking transient states of its directory (from both *loads* and *stores*). We confirmed it with an additional analysis of address access frequency at the memory controller, where we detected some cache lines are *hot-spots* for both read and write across the two clusters, in CXL-sensitive applications. From the miss latency distribution in Fig. 11, we see that loads with *medium*-range latency are further delayed with CXL into the *high*-range.

In summary, CXL slowdowns are inherent to its protocol design and independent of C$^3$. Note that CXL uses more directory handshaking and lacks peer-to-peer responses between hosts, because CXL is designed to cope with network message re-orderings and dynamically changing hosts in *sharer* lists.

## VII. Related Work

**CXL systems.** CXL is emerging as a promising technology with widespread support from industry [25], [40], [61], [70]. Most related work aims to use CXL memory for memory tiering (layering) [46], [48], [57], [79], [82], [83], [94], [98] and memory pooling (sharing) [17], [18], [36], [80], [81]. Beyond these applications, numerous studies have investigated CXL to improve a wide range of distributed systems and applications [4], [9], [16], [37], [49], [56], [85], [91], [97].

Evaluations of real CXL hardware have primarily focused on its use to realize memory expanders [51], [52], [86]. Multi-host coherence, as defined by CXL 3.0, was not studied so far on real hardware due to the lack of commercially available hardware supporting this feature. While different CXL-based systems have been assessed through both emulation and simulation [9], [12], [35], [44], [48], [57], [72], [89], [93], [95], no prior work, to the best of our knowledge, has examined CXL within a heterogeneous multi-host coherence setup. To address this gap, we present C$^3$ to guarantee predictable and consistent behavior across heterogeneous multi-host CXL systems.

**Heterogeneous cache coherence.** Various industry standards [14], [22], [24], [27] aim to enable heterogeneous cache coherence in multicore architectures. Prior work has proposed both manual [8], [45], [65] and automated [66]–[68] techniques to combine protocols that bridge the semantics of heterogeneous architectures. These methods suffer from assumptions on a static architecture and/or a lack of generality. They either use merged directories [8], [68], which prevents dynamically connecting new hosts or they realize custom interfaces [65], [67] that only support SWMR protocols.

Notably, neither of these provides a comprehensive method for how arbitrary heterogeneous protocols can be combined

within a hierarchical framework. To address this limitation, we develop design rules for heterogeneous CXL systems.

**Memory consistency models.** A clear MCM is often missing in past heterogeneous cache coherence work. Dedicated works such as Memglue [21] propose modifying coherence controllers to orchestrate coherence protocols, thereby realizing a MCM that aligns with the C memory model. In contrast, compound memory models [31], [63], [68] define a more lightweight approach that preserves the local MCM semantics of all hosts. However, these and similar approaches [27], [41] are specific to their target systems and do not apply CXL. Formalization for CXL has been presented recently [84] but currently omits `CXL.mem` and multi-host coherence. C$^3$'s design and rules produce a compound MCM for CXL coherence.

## VIII. Conclusion

This work presents C$^3$, a CXL coherence controller that overcomes interoperability challenges arising from the mismatch of coherence protocols and memory consistency models in heterogeneous CXL-connected systems. C$^3$ reconciles disparate cache coherence protocols by synthesizing protocol bridges, which we validate through formal verification and performance evaluation across diverse workloads. Our results demonstrate that C$^3$ preserves host memory consistency semantics while incurring minimal performance overhead.


## Acknowledgment

We thank the anonymous reviewers as well as the members of the Systems Research Group at the TU Munich for their valuable feedback on this work. This work was supported in part by an ERC Starting Grant (ID: 101077577), the DFG Priority Program "Disruptive Memory Technologies" (SPP 2377), the Chips Joint Undertaking (JU), the Intel Trustworthy Data Center of the Future (TDCoF), Google Research Grants, and the Alexander von Humboldt Foundation. The authors acknowledge the financial support by the Federal Ministry of Research, Technology and Space of Germany in the programme of "Souverän. Digital. Vernetzt.". Joint project 6G-life, project identification number: 16KISK002


## Artifact Appendix

### A. Abstract

This artifact provides a gem5-based implementation of C$^3$, along with instructions to reproduce the evaluation results of this work. The repository includes detailed guidance on using the models, installing dependencies, and compiling the benchmarks from scratch.

To facilitate the use of the artifact, we also provide a Docker container with its `Dockerfile` to set up the environment with all required dependencies.

Additionally, to save compilation time for the workloads and all 6 gem5 model variants of C$^3$ (approx 30min-1h30 on 128-core server), we offer a prebuilt Docker image that includes all compiled binaries for C$^3$ (models and workloads), ready to run the simulations.

### B. Artifact check-list (meta-information)

- **Compilation:** GCC 11.4.0, SCons 4.0+, Python 3.10+, and all gem5 [29] v23.1.0.0 dependencies
- **Data set:** PARSEC 3.0, SPLASH-4, Phoenix-2.0; Litmus tests: *IRIW, 2_2W, LB, MP, R, S, SB* (generated with Herd7 [6])
- **Run-time environment:** Ubuntu 22.04 LTS or 24.04 LTS (native or Docker container)
- **Metrics:** Execution time.
- **Experiments:** Use the provided scripts to evaluate the execution times of different cache coherence protocol combinations.
- **How much disk space is required? (approx):** Total: $\sim$30 GiB. Breakdown: gem5 builds: $\sim$23 GiB, PARSEC: $\sim$5 GiB, other benchmarks: $<$1 GiB, experiment outputs: $<$1 GiB.
- **How much time is needed to prepare workflow? (approx):** $\sim$1–3 hours to compile all gem5 variants; $\sim$30 minutes to compile the workloads.
- **How much time is needed to complete experiments? (approx):** $\sim$4–12 hours for the full experiment suite (Figure 9, 10, 11, and litmus tests) on a 32-core server.
- **Publicly available:** Yes
- **Code licenses:** MIT
- **Archived (DOI):** https://doi.org/10.5281/zenodo.17828238

### C. Description

*1) How to access:* The source code is publicly available on GitHub (https://github.com/TUM-DSE/C3) or Zenodo (https://doi.org/10.5281/zenodo.17828238).

*2) Hardware dependencies:* The artifact can be evaluated on any general-purpose CPU with at least 30 GiB free disk space to build and run gem5. We recommend running the experiments on a server with at least 32 cores to speed up compilation and simulation.

*3) Software dependencies:* We recommend running all compilation and experiments inside a docker container with the Dockerfile provided. Thus, the only software dependency is a working Docker environment. Alternatively, a Ubuntu 22.04 or 24.04 installation can be used (native, container or VM). To run the artifacts natively, refer to the artifact repository for additional instructions to setup the environment.

*4) Data sets:* The artifact includes the source code and instructions to build the workloads from source. We also provide the compiled binaries as a Docker image on DockerHub (https://hub.docker.com/r/gingerbreadz/c3-artifact-prebuilt/). To build from source, use the `Build` scripts as noted in the instruction `README`.

### D. Installation

The artifact repository contains all necessary components, including the gem5 simulator, benchmark suites, and experiment scripts. For each component, we provide detailed build instructions in its respective `README` section.

To build gem5 with all protocol variants (MESI-MESI-MESI, MESI-CXL-MESI, MESI-CXL-MOESI, MESI-CXL-MESIF) for both X86 and ARM architectures, please follow the instructions in `Build gem5` section of the artifact `README`.

To build the benchmarks, please follow the instructions in `Build Benchmarks` section of the artifact `README`.

To run the experiments, you need first to run the workload configurations script (`Generate Workload`

`Configurations`) and then run the experiments. Detailed instructions are available in `Run Experiments`.

### E. Evaluation and expected results

Once all experiments have been completed, the run script will create plots for Fig. 9, Fig. 10, Fig. 11, and Tab. IV and place them into the `data` folder. The figures should match the ones in the paper.

### F. Methodology

Submission, reviewing, and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

## REFERENCES


[1] "Nvidia unveils nvlink fusion for industry to build semi-custom ai infrastructure with nvidia partner ecosystem." [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-nvlink-fusion-semi-custom-ai-infrastructure-partner-ecosystem

[2] "The princeton application repository for sharedmemory computers (parsec)." [Online]. Available: http://parsec.cs.princeton.edu/.

[3] K. M. U. Ahmed, M. H. J. Bollen, and M. Alvarez, "A review of data centers energy consumption and reliability modeling," *IEEE Access*, vol. 9, pp. 152 536–152 563, 2021.

[4] M. Ahn, A. Chang, D. Lee, J. Gim, J. Kim, J. Jung, O. Rebholz, V. Pham, K. Malladi, and Y. S. Ki, "Enabling cxl memory expansion for in-memory database management systems," in *Proceedings of the 18th International Workshop on Data Management on New Hardware*, ser. DaMoN '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3533737.3535090

[5] J. Alglave, W. Deacon, R. Grisenthwaite, A. Hacquard, and L. Maranget, "Armed cats: Formal concurrency modelling at arm," *ACM Trans. Program. Lang. Syst.*, vol. 43, no. 2, 2021.

[6] J. Alglave and L. Maranget, "herd7 consistency model simulator," http://diy.inria.fr/www/, 2022.

[7] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood, "Lazy release consistency for gpus," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–14.

[8] J. Alsop, M. D. Sinclair, and S. V. Adve, "Spandex: a flexible interface for efficient heterogeneous coherence," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 261–274. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00031

[9] C. Alverti, S. Psomadakis, B. Ocalan, S. Jaiswal, T. Xu, and J. Torrellas, "Cxlfork: Fast remote fork over cxl fabrics," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 210–226. [Online]. Available: https://doi.org/10.1145/3676641.3715988

[10] Amazon, "Amazon ec2 instance types." [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[11] AMD, "Amd64 architecture programmer's manual volume 2: System programming," 2012. [Online]. Available: https://web.archive.org/web/20170619232736/http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf#page=217

[12] Y. An, S. Yi, B. Mao, Q. Li, M. Zhang, K. Zhou, N. Xiao, G. Sun, X. Wang, Y. Luo, and J. Zhang, "A novel extensible simulation framework for cxl-enabled systems," 2024. [Online]. Available: https://arxiv.org/abs/2411.08312

[13] ARM, "Arm architecture reference manual armv7-a and armv7-r edition," ARM Limited, Tech. Rep., 2011. [Online]. Available: https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Application-Level-Memory-Model/Memory-types-and-attributes-and-the-memory-order-model/Atomicity-in-the-ARM-architecture

[14] ARM, "Amba chi architecture specification," 2024. [Online]. Available: https://developer.arm.com/Architectures/AMBA

[15] ARM, "Mesi and moesi protocols," 2024. [Online]. Available: https://developer.arm.com/documentation/den0013/d/Multi-core-processors/Cache-coherency/MESI-and-MOESI-protocols

[16] M. Bailleu, D. Stavrakakis, R. Rocha, S. Chakraborty, D. Garg, and P. Bhatotia, "Toast: A heterogeneous memory management system," in *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. New York, NY, USA: Association for Computing Machinery, 2024, p. 53–65. [Online]. Available: https://doi.org/10.1145/3656019.3676944

[17] D. S. Berger, D. Ernst, H. Li, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, L. Hsu, I. Agarwal, M. D. Hill, and R. Bianchini, "Design tradeoffs in cxl-based memory pools for public cloud platforms," *IEEE Micro*, vol. 43, no. 2, pp. 30–38, 2023.

[18] D. S. Berger, Y. Zhong, P. Zardoshti, S. Teng, F. Kazhamiaka, and R. Fonseca, "Octopus: Scalable low-cost cxl memory pooling," 2025. [Online]. Available: https://arxiv.org/abs/2501.09020

[19] S. Bharadwaj, J. Yin, B. Beckmann, and T. Krishna, "Kite: A family of heterogeneous interposer topologies enabled via accurate interconnect modeling," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[20] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[21] R. Cleaveland and C. Trippel, "Memory consistency model-aware cache coherence for heterogeneous hardware," in *2024 Formal Methods in Computer-Aided Design (FMCAD)*, 2024, pp. 1–12.

[22] C. Consortium, "Ccix standard." [Online]. Available: https://www.ccixconsortium.com/

[23] C. E. L. Consortium, "Compute express link (cxl) consortium," 2024. [Online]. Available: https://www.computeexpresslink.org/

[24] C. E. L. Consortium, "Compute express link (cxl) specification 3.2," Compute Express Link Consortium, Tech. Rep., 2024. [Online]. Available: https://computeexpresslink.org/cxl-specification/

[25] CXL Consortium, "Integrators List." [Online]. Available: https://computeexpresslink.org/integrators-list/

[26] D. Das Sharma, R. Blankenship, and D. Berger, "An introduction to the compute express link (cxl) interconnect," *ACM Comput. Surv.*, vol. 56, no. 11, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3669900

[27] H. Foundation, "Heterogeneous system architecture (hsa) foundation," 2024. [Online]. Available: https://hsafoundation.com/

[28] gem5, "gem5 slicc," 2024. [Online]. Available: https://www.gem5.org/documentation/general_docs/ruby/slicc/

[29] gem5 developers. (2023) gem5. [Online]. Available: https://github.com/gem5/gem5/releases/tag/v23.1.0.0

[30] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, p. 15–26, May 1990. [Online]. Available: https://doi.org/10.1145/325096.325102

[31] A. Goens, S. Chakraborty, S. Sarkar, S. Agarwal, N. Oswald, and V. Nagarajan, "Compound memory models," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. [Online]. Available: https://doi.org/10.1145/3591267

[32] E. J. Gómez-Hernández, J. M. Cebrian, S. Kaxiras, and A. Ros, "Splash-4: A modern benchmark suite with lock-free constructs," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*. Austin, TX (USA): IEEE Computer Society, Nov. 2022, pp. 51–64. [Online]. Available: http://webs.um.es/aros/papers/pdfs/ejgomez-iiswc22.pdf

[33] Google, "Google cloud: Machine families resource and comparison guide." [Online]. Available: https://cloud.google.com/compute/docs/machine-resource

[34] R. Gouicem, D. Sprokholt, J. Ruehl, R. C. O. Rocha, T. Spink, S. Chakraborty, and P. Bhatotia, "Risotto: A dynamic binary translator for weak memory model architectures," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2022, p. 107–122. [Online]. Available: https://doi.org/10.1145/3567955.3567962

[35] D. Gouk, M. Kwon, H. Bae, S. Lee, and M. Jung, "Memory pooling with cxl," *IEEE Micro*, vol. 43, no. 2, pp. 48–57, 2023.

[36] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access,{High-Performance} memory disaggregation with {DirectCXL}," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 287–294.

[37] Y. Gu, A. Khadem, S. Umesh, N. Liang, X. Servot, O. Mutlu, R. Iyer, and R. Das, "Pim is all you need: A cxl-enabled gpu-free system for large language model inference," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 862–881. [Online]. Available: https://doi.org/10.1145/3676641.3716267

[38] R. Hayden and P. Schell, "Opportunities and challenges for compute express link (cxl)," 2024. [Online]. Available: https://computeexpresslink.org/wp-content/uploads/2024/11/CR-CXL-101_FINAL.pdf

[39] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[40] G. Hilson, "Micron exits 3d xpoint market, eyes cxl opportunities." [Online]. Available: https://www.eetimes.com/micron-exits-3d-xpoint-market-eyes-cxl-opportunities/

[41] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free memory models," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, p. 427–440, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2654822.2541981

[42] Intel, "Intel® 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, part 1," Intel Corporation, Tech. Rep., 2024.

[43] S. Jain, N. Yeleswarapu, H. A. Maruf, and R. Gupta, "Memory sharing with cxl: Hardware and software design approaches," 2024. [Online]. Available: https://arxiv.org/abs/2404.03245

[44] J. Jang, H. Choi, H. Bae, S. Lee, M. Kwon, and M. Jung, "{CXL-ANNS}:{Software-Hardware} collaborative memory disaggregation and computation for {Billion-Scale} approximate nearest neighbor search," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 585–600.

[45] E. Ladan-Mozes and C. E. Leiserson, "A consistency architecture for hierarchical shared caches," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 11–22. [Online]. Available: https://doi.org/10.1145/1378533.1378536

[46] T. Lee, S. K. Monga, C. Min, and Y. I. Eom, "Memtis: Efficient memory tiering with dynamic page classification and page size determination," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 17–34. [Online]. Available: https://doi.org/10.1145/3600006.3613167

[47] A. Lefort, J. Pritzi, N. Carpentieri, D. Schall, S. Dittrich, S. Chakraborty, N. Oswald, and P. Bhatotia, "vcxlgen: Automated synthesis and verification of cxl bridges for heterogeneous architectures," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2026. [Online]. Available: https://doi.org/10.1145/3779212.3790245

[48] 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, 2023, p. 574–587. [Online]. Available: https://doi.org/10.1145/3575693.3578835

[49] S. Li, Y. E. Zhou, H. Ren, and J. Huang, "Bytefs: System support for (cxl-based) memory-semantic solid-state drives," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 116–132. [Online]. Available: https://doi.org/10.1145/3669940.3707250

[50] X. Li, Z. Guo, Y. Bai, M. Ketkar, H. Wilkinson, and M. Liu, "Understanding and profiling cxl.mem using pathfinder," in *Proceedings of the ACM SIGCOMM 2025 Conference*, ser. SIGCOMM '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 758–779. [Online]. Available: https://doi.org/10.1145/3718958.3750479

[51] 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*, 2025, pp. 1203–1217.

[52] J. Liu, H. Hadian, H. Xu, D. S. Berger, and H. Li, "Dissecting cxl memory performance at scale: Analysis, modeling, and optimization," *arXiv preprint arXiv:2409.14317*, 2024.

[53] J. Lowe-Power, "gem5 ruby," 2025. [Online]. Available: https://www.gem5.org/documentation/general_docs/ruby/

[54] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrill'on, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. M"uck, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and 'Eder F. Zulian, "The gem5 simulator: Version 20.0+," *CoRR*, vol. abs/2007.03152, 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[55] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi, "Armor: defending against memory consistency model mismatches in heterogeneous architectures," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 388–400. [Online]. Available: https://doi.org/10.1145/2749469.2750378

[56] T. Ma, Z. Liu, C. Wei, J. Huang, Y. Zhuo, H. Li, N. Zhang, Y. Guan, D. Niu, M. Zhang, and T. Ma, "Hydrarpc: Rpc in the cxl era," in *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC'24. USA: USENIX Association, 2024.

[57] 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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 742–755. [Online]. Available: https://doi.org/10.1145/3582016.3582063

[58] P. McKenney, *Memory Barriers: a Hardware View for Software Hackers*, 2010. [Online]. Available: http://www.rdrop.com/~paulmck/scalability/paper/whymb.2010.07.23a.pdf

[59] P. McKenney, *Is Parallel Programming Hard, And, If So, What Can You Do About It?*, 2024. [Online]. Available: https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

[60] Mincrosoft, "Azure virtual machine series." [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/

[61] T. P. Morgan, "The cxl roadmap opens up the memory hierarchy." [Online]. Available: https://www.nextplatform.com/2021/09/07/the-cxl-roadmap-opens-up-the-memory-hierarchy/

[62] T. P. Morgan, "Cxl and gen-z iron out a coherent interconnect strategy," 2020. [Online]. Available: https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/

[63] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020. [Online]. Available: https://doi.org/10.2200/S00962ED2V01Y201910CAC049

[64] R. Networks, "How cxl technology solves memory problems in data centres," 2024. [Online]. Available: https://www.ruijienetworks.com/support/tech-gallery/how-cxl-technology-solves-memory-problems-in-data-centres-part1

[65] L. E. Olson, M. D. Hill, and D. A. Wood, "Crossing guard: Mediating host-accelerator coherence interactions," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17.

New York, NY, USA: Association for Computing Machinery, 2017, p. 163–176. [Online]. Available: https://doi.org/10.1145/3037697.3037715

[66] N. Oswald, V. Nagarajan, and D. J. Sorin, "Protogen: Automatically generating directory cache coherence protocols from atomic specifications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 247–260.

[67] N. Oswald, V. Nagarajan, and D. J. Sorin, "Hieragen: Automated generation of concurrent, hierarchical cache coherence protocols," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 888–899.

[68] N. Oswald, V. Nagarajan, D. J. Sorin, V. Gavrielatos, T. Olausson, and R. Carr, "Heterogen: Automatic synthesis of heterogeneous cache coherence protocols," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 756–771.

[69] S. Owens, "Reasoning about the implementation of concurrency abstractions on x86-TSO," in *ECOOP*, 2010, pp. 478–503.

[70] M. Papermaster, "AMD Joins Consortia to Advance CXL, a New High-Speed Interconnect for Breakthrough Performance." [Online]. Available: https://community.amd.com/t5/business/amd-joins-consortia-to-advance-cxl-a-new-high-speed-interconnect/ba-p/418202

[71] M. Pöter and J. L. Träff, "Memory models for C/C++ programmers," *CoRR*, vol. abs/1803.04432, 2018. [Online]. Available: http://arxiv.org/abs/1803.04432

[72] A. Puri, K. Bellamkonda, K. Narreddy, J. Jose, V. Tamarapalli, and V. Narayanan, "Dracksim: Simulating cxl-enabled large-scale disaggregated memory systems," in *Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 3–14. [Online]. Available: https://doi.org/10.1145/3615979.3656059

[73] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," USA, p. 13–24, 2007. [Online]. Available: https://doi.org/10.1109/HPCA.2007.346181

[74] B. Reidys, P. Zardoshti, I. n. Goiri, C. Irvene, D. S. Berger, H. Ma, K. Arya, E. Cortez, T. Stark, E. Bak, M. Iyigun, S. Novakovic, L. Hsu, K. Trueba, A. Pan, C. Bansal, S. Rajmohan, J. Huang, and R. Bianchini, "Coach: Exploiting temporal patterns for all-resource oversubscription in cloud platforms," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 164–181. [Online]. Available: https://doi.org/10.1145/3669940.3707226

[75] S. Reimers, D. Sprokholt, M. Fink, T. Augoustis, S. Kammermeier, R. C. O. Rocha, T. Spink, R. Gouicem, S. Chakraborty, and P. Bhatotia, "Arancini: A hybrid binary translator for weak memory model architectures," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2026.

[76] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, "Hmg: Extending cache coherence protocols across modern hierarchical multi-gpu systems," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 582–595.

[77] R. C. O. Rocha, D. Sprokholt, M. Fink, R. Gouicem, T. Spink, S. Chakraborty, and P. Bhatotia, "Lasagne: A static binary translator for weak memory model architectures," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2022. [Online]. Available: https://doi.org/10.1145/3519939.3523719

[78] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-tso: a rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010. [Online]. Available: https://doi.org/10.1145/1785414.1785443

[79] S. Sha, C. Li, Y. Luo, X. Wang, and Z. Wang, "vtmm: Tiered memory management for virtual machines," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 283–297. [Online]. Available: https://doi.org/10.1145/3552326.3587449

[80] D. D. Sharma, "Compute express link (cxl): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy," *IEEE Micro*, vol. 43, no. 2, pp. 99–109, 2023.

[81] J. Sim, S. Ahn, T. Ahn, S. Lee, M. Rhee, J. Kim, K. Shin, D. Moon, E. Kim, and K. Park, "Computational cxl-memory solution for accelerating memory-intensive applications," *IEEE Computer Architecture Letters*, vol. 22, no. 1, pp. 5–8, 2023.

[82] Y. Sun, J. Kim, Z. Yu, J. Zhang, S. Chai, M. J. Kim, H. Nam, J. Park, E. Na, Y. Yuan, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim, "M5: Mastering page migration and memory management for cxl-based tiered memory systems," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 604–621. [Online]. Available: https://doi.org/10.1145/3676641.3711999

[83] B. Tabatabai, J. Sorenson, and M. M. Swift, "Fbmm: making memory management extensible with filesystems," in *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC'24. USA: USENIX Association, 2024.

[84] C. Tan, A. F. Donaldson, and J. Wickerson, "Formalising cxl cache coherence," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 437–450. [Online]. Available: https://doi.org/10.1145/3676641.3715999

[85] Y. Tang, S.-s. Lee, A. Bhattacharjee, and A. Khandelwal, "pulse: Accelerating distributed pointer-traversals on disaggregated memory," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 858–875. [Online]. Available: https://doi.org/10.1145/3669940.3707253

[86] Y. Tang, P. Zhou, W. Zhang, H. Hu, Q. Yang, H. Xiang, T. Liu, J. Shan, R. Huang, C. Zhao, C. Chen, H. Zhang, F. Liu, S. Zhang, X. Ding, and J. Chen, "Exploring performance and cost optimization with asic-based cxl memory," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 818–833.

[87] A. Terekhov, "C/c++11 mappings to processors," 2008. [Online]. Available: https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

[88] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: the next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387517

[89] Y. Wang, L. Wu, W. Hong, Y. Ou, Z. Wang, S. Gao, J. Zhang, S. Ma, D. Dong, X. Qi, M. Lai, and N. Xiao, "A comprehensive simulation framework for cxl disaggregated memory," 2025. [Online]. Available: https://arxiv.org/abs/2411.02282

[90] Y. Wang, L. Wu, W. Hong, Y. Ou, Z. Wang, S. Gao, J. Zhang, S. Ma, D. Dong, X. Qi, M. Lai, and N. Xiao, "Cxl-dmsim: A full-system cxl disaggregated memory simulator with comprehensive silicon validation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 1–1, 2025. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2025.3607145

[91] Z. Wang, Y. Chen, C. Li, Y. Guan, D. Niu, T. Guan, Z. Du, X. Wei, and G. Sun, "Ctxnl: A software-hardware co-designed solution for efficient cxl-based transaction processing," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 192–209. [Online]. Available: https://doi.org/10.1145/3676641.3716244

[92] WikiChip. (2024) Golden cove - microarchitectures - intel. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/golden_cove

[93] J. Wu, J. Liu, G. Kestor, R. Gioiosa, D. Li, and A. Marquez, "Performance study of cxl memory topology," in *Proceedings of the International Symposium on Memory Systems*, 2024, pp. 172–177.

[94] 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 *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'24. USA: USENIX Association, 2024.

[95] Y. Yang, P. Safayenikoo, J. Ma, T. A. Khan, and A. Quinn, "Cxlmemsim: A pure software simulated cxl.mem for performance characterization," 2023. [Online]. Available: https://arxiv.org/abs/2303.06153

[96] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the intel last-level cache," Cryptology ePrint Archive, Paper 2015/905, 2015. [Online]. Available: https://eprint.iacr.org/2015/905

[97] M. Zhang, T. Ma, J. Hua, Z. Liu, K. Chen, N. Ding, F. Du, J. Jiang, T. Ma, and Y. Wu, "Partial failure resilient memory management

system for (cxl-based) distributed shared memory," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 658–674. [Online]. Available: https://doi.org/10.1145/3600006.3613135

[98] 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 *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'24. USA: USENIX Association, 2024.