vCXLGen: Automated Synthesis and Verification of CXL Bridges for Heterogeneous Architectures

Anatole Lefort*
Technical University of Munich
Munich, Germany

Julian Pritzi*
Technical University of Munich
Munich, Germany

Nicolò Carpentieri
Technical University of Munich
Munich, Germany

David Schall
Technical University of Munich
Munich, Germany

Simon Dittrich
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

Compute Express Link (CXL) offers byte-addressable, cache-coherent remote memory accesses across multiple hosts. Unfortunately, the CXL specification lacks mechanisms to ensure safe interoperability between *heterogeneous host architectures* with diverse cache coherence (CC) protocols and memory consistency models (MCMs). This semantic gap poses fundamental challenges and a significant barrier to adopting CXL in modern heterogeneous data centers.

We propose CXL bridges, an abstraction that interposes between hosts and CXL to reconcile differences in CC protocols and MCMs. We present vCXLGen, the first system that automatically synthesizes and verifies these CXL bridges. We make two core contributions: (1) a fully automated approach to synthesize CXL bridges from machine-readable CC protocol specifications, and (2) a compositional formal verification approach for scalable model-checking of liveness properties.

Our evaluation shows that vCXLGen is general, i.e., it supports diverse protocols (both SWMR and relaxed consistency) and is easily *extensible* when integrating new protocols, such as `CXL.mem`. Our *performance* evaluations indicate that synthesised bridges achieve comparable results to manually designed homogeneous protocols. Finally, for *correctness*, our formal verification rigorously proves the safety and liveness of synthesized bridges, all while achieving significant scalability in liveness verification of complex heterogeneous systems.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Hardware** → **Emerging technologies**.

*Authors contributed equally.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790245

Keywords: Cache, coherence; CXL; Disaggregated, systems

ACM Reference Format:
Anatole Lefort, Julian Pritzi, Nicolò Carpentieri, David Schall, Simon Dittrich, Soham Chakraborty, Nicolai Oswald, and Pramod Bhatotia. 2026. 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, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3779212.3790245

1 Introduction

Modern data centers are increasingly heterogeneous, employing diverse CPUs (x86, Arm, RISC-V) and accelerators (GPUs, FPGAs, TPUs) for performance, power efficiency, domain-specific computing, and license requirements [9, 40, 60]. This architectural diversity necessitates a standardized interconnect for sharing data across architectures.

Compute Express Link (CXL) [31, 34, 43] has emerged as a promising industry standard, offering efficient, low-latency, byte-addressable memory access. The latest iteration, CXL v3.0 [31, 46], introduces a pivotal advancement: hardware-enforced cache coherence across multiple hosts, which paves the way for high-performance distributed shared memory (DSM) by eliminating the complexities of software-based solutions. A boon for developing distributed systems [33, 53, 63].

Unfortunately, the CXL v3.0 specification lacks inherent mechanisms to guarantee safe interoperability across heterogeneous host architectures, creating a significant obstacle for hardware-coherent DSM in heterogeneous environments: the semantic gap between varied, vendor-specific cache coherence (CC) protocols and memory consistency models (MCMs) [2, 13, 45, 62] that prevents combining diverse architectures and impedes broader CXL adoption.

This semantic gap manifests in two key dimensions. Firstly, it stems from the inherent differences between diverse vendor-specific CC protocols and the CXL-defined protocol (`CXL.mem`).

To enable multi-host coherence, hardware vendors must integrate `CXL.mem` with their architecture-specific CC protocols—deeply ingrained in their processor design, optimized for their specific memory subsystem, and not initially intended for external interoperability. Secondly, vendors hold varying perspectives on MCMs: some enforce Single-Writer Multiple-Reader (SWMR) [10, 13, 45] while others support relaxed memory consistency [6, 7, 62]. These heterogeneous models introduce programmability challenges when diverse hosts access shared memory regions via CXL [38].

To systematically address these challenges, we introduce the concept of a *CXL bridge*: an abstraction interposing between a host and the CXL interconnect. This bridge orchestrates the existing host and CXL fabric CC protocols transparently, propagating requests and responses bidirectionally between them without requiring any modifications to either protocol. Our vision is a drop-in component that enables seamless CXL integration for any host system, encapsulating the complexity required for correct interoperation while preserving each host's native CC protocols and MCMs. From the CXL perspective, each bridged host appears as a standard CXL device, while the host operating system views the CXL memory as a conventional NUMA node or DAX device [55].

Realizing an effective CXL bridge entails addressing five key requirements: (1) *Generality* to support diverse host architectures (CCs, MCMs) without requiring disruptive changes. (2) *Automated synthesis* to eliminate the error-prone manual process of translating protocol states and transitions. (3) Well-defined memory consistency for the integrated system, providing predictable *programmability* semantics despite underlying host MCM variations. (4) *Correctness* through formal verification, to ensure progress (liveness) and adherence to the defined MCMs (safety). (5) *Performance* to fulfill CXL's promise of low-latency and transparent remote memory access.

To meet these requirements, we present vCXLGen, the first system for the automated and correct-by-construction synthesis of CXL bridges. vCXLGen takes as input the protocol specifications for both the local (host-specific) and global (CXL) CC protocols, expressed in the ProtoGen DSL [66]. It then generates comprehensive specifications for the merged system-wide coherence protocol. Notably, vCXLGen tackles three key challenges: (1) Protocol combination complexity by automating the intricate process of identifying propagation requirements, semantic translation, and request linearization across heterogeneous protocols. (2) Verification scalability by employing a novel compositional approach that leverages the modularity of multi-host systems to enable rigorous verification of complex real-world scenarios, overcoming the state-space explosion [67, 68]. (3) Performance evaluation by providing a gem5 simulator backend, creating the first testbed for evaluating CXL bridges in heterogeneous multi-host settings.

Our evaluation demonstrates the effectiveness of vCXLGen across several key dimensions. Regarding *generality*, vCXLGen successfully synthesizes CXL bridges for various cache coherence protocols, including those adhering to SWMR and relaxed consistency models. We validate their functionality by running 35 parallel applications from three benchmark suites (PARSEC [1], Phoenix [71], and SPLASH [39]) and a distributed KVS using YSCB workloads [32]. Regarding *extensibility*, the modular design of our system allows for the straightforward addition of new protocols, as evidenced by relatively concise specifications for the new `CXL.mem` protocol [31]. For *performance*, we show that the automatically generated bridges achieve comparable performance to optimized homogeneous protocols. Finally, regarding *verification*, we show safety using model-checking and present a novel compositional approach that verifies liveness properties while achieving significant scalability in complex system models.

Overall, our paper makes two core contributions:

(1) Automated synthesis of CXL bridges: We present a generic and automated approach for combining coherence protocols with CXL. This is realized through vCXLGen, a generator synthesizing CXL bridges directly from protocol specifications. Leveraging vCXLGen, we generate a diverse set of CXL bridges tailored for various CC protocols and architectures. Furthermore, we develop a custom gem5 backend, establishing the first simulation testbed capable of evaluating CXL multi-host coherence with our generated bridges.

(2) Automated verification of CXL bridges: We formally verify that generated CXL bridges preserve local MCMs and guarantee progress. With vCXLGen, we generate system models for the Murφ and *Rumur* model checkers. We use 216 litmus tests to characterize the MCM of various CXL multi-host systems integrating our bridges, and verify that each thread always adheres to its local MCMs axioms. Last, we introduce compositional liveness checks: a novel automated verification technique that scales to complex multi-host systems.

2 Background

CXL systems. Compute Express Link (CXL) [30, 31] is an open interconnect standard designed for high-speed, efficient communications between host processors and devices like accelerators, GPUs or memory expanders [43]. CXL defines two cache coherence protocols: `CXL.cache` and `CXL.mem`. The `CXL.mem` protocol provides hosts with transparent and coherent access to device-attached memory [43], presented as a physical memory mapping [79]. With CXL version 3.0, the specification defines how multiple hosts can coherently access the same shared CXL memory, through device-to-host invalidation flows `BISnp{Inv,Data}`. The hardware implementing `CXL.mem` protocol must, in turn, ensure that host caches are kept consistent. Namely, the specification mandates that the *Home Agent* (HA), a component responsible for enforcing cache coherence on the host [31], is extended to allow `CXL.mem` to affect host caches. This extension of the HA effectively bridges the host's CC protocol and `CXL.mem`.

Description	MESI msg	CXL msg
Get copy with exclusive ownership	`GetM`	`MemRd-A`
Get sharable copy	`GetS`	`MemRd-S`
Writeback, do not keep cachable copy	`WB+Put`	`MemWr-I`
Writeback, retain cachable copy	`WB`	`MemWr-S`
Device requests sharable copy from host	`Fwd-GetS`	`BISnpData`
Device requests exclusive cachable copy	`Fwd-GetM`	`BISnpInv`

Table 1. Most notable `CXL.mem` coherence messages with meta values and their equivalents in the MESI protocol [62].

Cache coherence (CC) protocols. CC protocols ensure data consistency across cores in a multi-core system. They define caches' actions in response to memory requests — invalidating or updating cache lines, exchanging messages, or retrieving data from other caches or memory. CC protocols can be classified by whether they enforce Single-Writer-Multiple-Reader (SWMR) or rely on write-backs and self-invalidation.

The MESI family is a collection of SWMR CC protocols described based on their possible cache line states: **I**nvalid, **M**odified, **S**hared, **E**xclusive and **O**wned. MESI variants are widely adopted by CPU vendors such as Intel, AMD and Arm [10, 13, 45]. In contrast, GPUs commonly implement write-backs and self-invalidation like the RCC protocol [62].

`CXL.mem` is a MESI variant with subtle differences, notably, the use of a handshake (`BIConflict`) to resolve potential ordering conflicts detected at the hosts. Tab. 1 summarizes other relevant `CXL.mem` messages with their MESI equivalents.

Memory Consistency Models (MCMs). CC protocols are integral to a system's memory consistency model (MCM). MCMs defines the order in which memory operations appear to be executed by a processor, establishing crucial rules for writing correct and efficient parallel programs.

Sequential consistency (SC)[49] is a strong, intuitive model where memory operations follow a global order, exactly as written in the program, but it is costly to implement. Modern CPUs instead adopt relaxed MCMs: x86 uses Total Store Order (TSO), preserving store ordering only[72]. Meanwhile, ARM, RISC-V, and GPUs implement weaker models, placing more synchronization responsibility on the programmer [2, 11, 62].

Recently proposed *Compound MCMs* [38, 62, 68] explore the memory consistency resulting from combining threads with heterogeneous MCMs. They show within an abstract compositional operational model, that by propagating each thread's native ordering constraints globally, each thread can still perceive its original MCM, regardless of other (heterogeneous) system threads. This conveniently keeps existing software correct, without modification or re-compilation.

3 Motivation

The CXL standard intends to enable interoperability of any device that adheres to the specifications — CPUs (x86, Arm, RISC-V), GPUs, FPGAs, accelerators, and memory devices. By this means, Fig. 1 depicts a heterogeneous example where two compute nodes (Intel x86 and Arm) share and concurrently access a common memory region through CXL. The x86 and

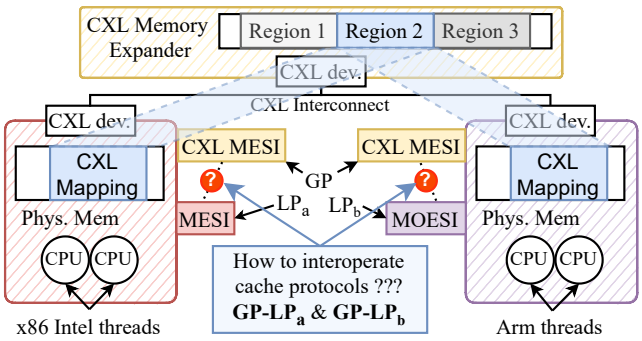


Figure 1. Heterogeneous system (Intel-Arm) that maps and shares a multi-host CC CXL memory region. Each host uses its own *local* protocol $LP_{\{a,b\}}$ (MESI, MOESI). The *global* protocol *GP* (`CXL.mem`) propagates coherence effects across the interconnect to the two compute nodes and the memory node.

Arm threads observe those concurrent memory operations, as they are propagated by the combination of 3 variables: (1) hosts' MCMs (TSO, relaxed), (2) local CC protocols (MESI, MOESI), and (3) the global CC protocol (`CXL.mem`). Crucially, we identify that the logic extensions required to tie together local protocols (LPs) with the global protocol (GP) are inherently complex due to protocol disparities, and need to be purpose-built for each host (MCM and LP specific).

3.1 The Case for a CXL Bridge

In Fig. 1, the extensions required to integrate each LP with `CXL.mem` are consequential for the global consistency of memory operations. However, CXL does not prescribe how to safely integrate, i.e., guarantee correct and consistent behavior, particularly across heterogeneous hosts and devices.

A central challenge arises from significant disparities between CC protocols, creating a *semantic gap*, even when they use the same stable states. For example, textbook MESI relies on causality to resolve request races, whereas CXL MESI adds a `BIConflict` handshake to disambiguate behavior. Upon receiving an invalidation, a textbook MESI cache controller (cc) always invalidates, while a CXL cc may initiate a handshake depending on its internal state. More heterogeneity amplifies those disparities: e.g., a RCC cc may request a shared or exclusive copy for a `store`, whether it follows a *release* event.

The above examples demonstrate that tying LPs with GP always requires complex logic, with *context-sensitive* semantic translation of requests. Extending host architectures with ad-hoc solutions is brittle and likely to introduce memory consistency bugs, which are notoriously difficult to detect and debug. Thus, we advocate for an automated and verifiable approach that introduces minimal changes to host architectures.

Our proposal: We propose the notion of *CXL bridges*—a logical component that sits at the frontier between LP and GP CC domains. Internally, it contains the logic extensions to perform the *context-sensitive* and *semantic* translation of requests when propagating them across domains. Because this logic

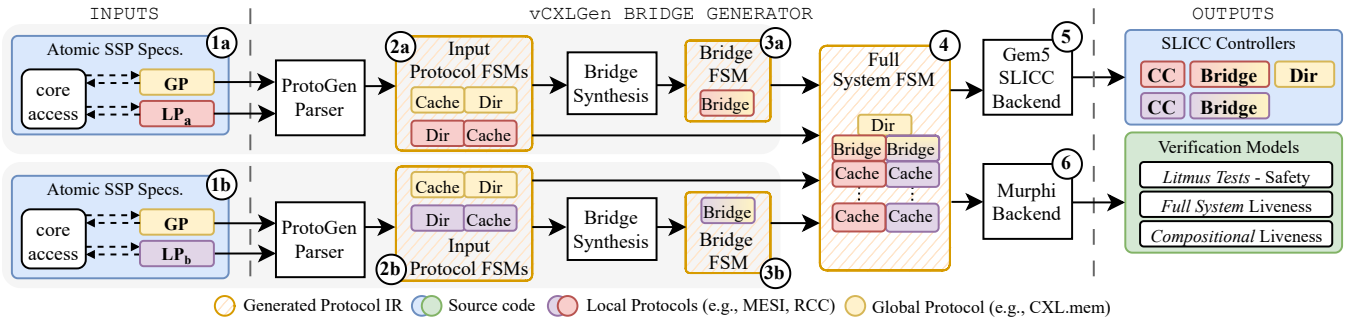


Figure 2. Overview of vCXLGen architecture. Bridge synthesis pipelines run in parallel for each input local protocol *LP*. It generates protocol controller *intermediate representations (IRs)* (②,③,④) from specifications (①), and exports them to controller implementations for simulations (⑤) and verification models (⑥).

is host-specific, critical, and with a complexity exacerbated in heterogeneous systems, we ask the following:

Research question: *How can we automatically synthesize and verify a CXL bridge for heterogeneous multi-host CC?*

3.2 Design Goals

We identify five key design goals for the CXL bridge.

(1) *Generality:* The bridge must support a wide range of host architectures with distinct CC protocols and consistency models without requiring invasive system changes. (2) *Automated synthesis:* To replace the complex and error-prone manual design process, a fully automated, end-to-end methodology is needed that generates CXL bridges from input specifications. (3) *Programmability:* The bridge must adhere to a MCM that does not violate MCM semantics of individual hosts, to maintain compatibility with existing applications, and ease adoption across heterogeneous systems. (4) *Correctness:* As bridges become an integral part of the memory subsystem and can impact MCM semantics, a rigorous verification methodology is needed to ensure the bridge is deadlock-free (liveness) and correctly enforces hosts' MCMs (safety). (5) *Performance:* To maintain CXL's performance and low-latency promises, a comprehensive performance evaluation framework is needed to ensure the bridge does not introduce any bottlenecks.

3.3 Research gap

Although the diversity of heterogeneous MCMs and CC protocols is currently limited; rapid evolutions of protocols require frequent bridge redesign. For instance, CXL has undergone six revisions in five years, and new protocols like NVLink Fusion [64] continue to emerge. Automating bridge synthesis and verification avoids repeating this complex and error-prone process for every CXL revision.

Prior works on automatic CC protocol combination and heterogeneous MCM compounding all fail to cater to CXL systems. Hieragen [67] combines protocols hierarchically, but is limited to protocols of the MESI-family as it relies on ad-hoc translation rules between protocols. Thus, it lacks the generality required for heterogeneous CXL systems. Likewise, Heterogen [68] combines arbitrary CC protocols by fusing their directory controllers—at the memory device. Thus, it requires a priori knowledge of all participating protocols, which suits SoCs but not dynamic CXL network topologies. Furthermore, the verification methodology used in [66–68] sees a state-space explosion when modeling more than 4 cache controllers. This lack of scalability makes it inapplicable to multi-host CXL systems. Compound MCMs [38] provides a theoretical framework to globally order memory operations with heterogeneous MCM combinations. However, as they abstract memory operations as propagating atomically between two threads, their propagation rules overlook the practical challenges of request concurrency and semantic translation in real-world distributed coherence protocols.

We present a concrete synthesis algorithm that realizes Compound MCMs [38] principles with arbitrary input protocols and `CXL.mem`. It properly handles request concurrency and heterogeneity, synthesizes bridges that address protocol hierarchies without restricting original protocol concurrency; whereas HeteroGen [68] synthesizes an order-enforcing directory controller. Our novel compositional verification scales to complex systems with multiple clusters and CXL bridges.

4 vCXLGen Overview

We introduce vCXLGen, an end-to-end generator that automatically synthesizes verified and efficient state machines for *CXL bridges*. Our generated CXL bridges systematically and correctly assemble a system-wide *compound coherence protocol*, which is the fusion of one or multiple *host-specific* LP with `CXL.mem` (GP) through CXL bridges. We stress that vCXLGen automates the entire process.

System workflow. Fig. 2 depicts the entire workflow of vCXLGen, and how it turns machine-readable specifications of LP and GP into concrete CXL system representations, namely: (a) verifiable models for the model checkers, and (b) functional implementations for the *gem5* simulator.

Our generator takes as input *atomic stable state protocol (SSP) specifications* (①). It parses them individually and initializes the IR of the base protocols (②). We rely on *ProtoGen* [66] and its domain-specific language (DSL) in the front end to

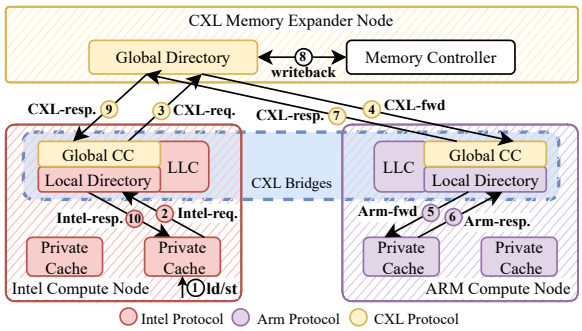


Figure 3. CXL bridges operating principle. Local flows (②, ⑩) are propagated across coherence domains with nested global transactions (③, ⑨). Global flows (④, ⑦) are translated locally (⑤, ⑥) to achieve a coherent state across hosts.

specify protocols concisely within only a few hundred code lines and make their finite state machines (FSMs) concurrent. The generator then proceeds with our bridge synthesis algorithm to fuse LP-dir and GP-cache FSMs and build the *Bridge IR* (③). Two backends then turn the IR of the full system into: ⑦ models for verification with the model checkers, and ⑧ an implementation of the resulting compound protocol in SLICC – the cache coherence DSL of the *gem5* simulator. For verification, we generate 3 different types of models: (1) *litmus test* models to characterize the full system's memory consistency (safety), (2) *full system* liveness models, and (3) our novel *compositional* liveness models and GP-equivalence checks.

5 CXL Bridge Architecture

In this section, we present our bridge abstraction and system model for the new coherence interface in CXL's *Host-managed Device Memory with Device-initiated Invalidation* (HDM-DB). It prompts us to define a new hierarchical organization of concerns between host/device (LP) and `CXL.mem` (GP) protocol domains. We summarize in Fig. 3 our system model and operating principle for CXL bridges, which we detail in what follows.

5.1 System Model

CXL shared memory systems. The multi-host system from Fig. 1 is a prime example of CXL's HDM-DB, where two heterogeneous hosts (Intel and Arm) access a single remote pool of hardware coherent shared memory. We model the CC architecture of this system in Fig. 3. The hosts (Intel and Arm) rely on LP internally and emit GP requests when memory operations can not be satisfied locally. The memory device tracks cache line states across hosts, with its coherence engine (DCOH) implementing a global *directory*.

Compound protocol hierarchies. CXL's HDM-DB settings resolve coherence globally through a two-layered stack of heterogeneous protocols. The GP domain, e.g., `CXL.mem`, resolves coherence between all hosts. The LP domain, e.g., MESI, RCC, resolves coherence between caches within a single host.

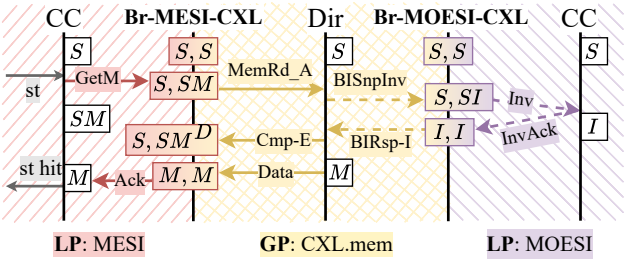


Figure 4. Example of transaction flow: *store miss from a shared state*. CXL bridges maintain coherence across multiple hosts.

Global coherence states across all three nodes are enabled by *CXL bridges*, which realize the coupling of LP and GP domains.

5.2 Operating Principles

The CXL bridge abstraction. We devise *CXL bridges* to incarnate simultaneously a LP directory and a GP coherence client, to effectively keep coherence state consistent across domains. Conceptually, CXL bridges implement a *compound state machine* of LP and GP, that dictates when *local* memory operations propagate *globally*, and vice versa. This logic also performs internally the *semantic translation* of requests, and enforces proper *global ordering* of requests through design rules.

Bridging rules. The bridging logic operates through *nesting of coherence transactions*. As seen in Fig. 3, a bridge processes either local requests (②) or global forwarded requests (④), that it translates and propagates with their counterparts in the other domain (③, ⑤). The correctness of the bridge then hinges on 3 key rules to propagate transactions:

- ***Delegation*:** native transaction flows are always used to propagate *globally visible* memory operations across domains.
- ***Nesting atomicity*:** nested transactions appear to complete atomically from the origin domain.
- ***Selective stalling*:** GP and LP requests are (un)blocked according to the global serialization order.

Example. We illustrate these three key principles with a concrete bridging example based on Fig. 4. (1) *Delegation*: When the bridge observes requests from LP or GP origin domains (`GetM` ②), it propagates them by nesting native flows from the remote domain (`MemRd-A` ③) (2) *Nesting atomicity*: Coherence effects become visible in the origin domain only after the nested transaction completes in the remote domain, meaning, the local response (`GetM-Ack` ⑩) is only sent after receiving the nested remote response (`Cmp-E` ⑨). (3) *Selective stalling*: Only the GP directory can globally order racing requests in GP. As such, GP *fwd* messages (`BISnpInv` ④) must not be indefinitely blocked by another GP request sent earlier by the bridge (`MemRd-A` ③). However, GP fwds may be safely stalled by the bridge until a LP transaction flow completes (`InvAck` ⑥).

6 Automated Synthesis of CXL Bridges

In this section, we present our generalized method to construct bridges between *local* and *global* protocols, and its application inside the automated synthesis pipeline of vCXLGen.

Intuition. We synthesize bridges by fusing the finite state machines (FSMs) of the *local directory* and *global cache controller*. The end goal is to produce a bridge FSM with transition sequences that propagate complete flows as the one seen in Fig. 4. For short, our synthesis algorithm identifies and composes transition sequences from the input LP and GP FSMs, in a specific way that realizes our bridging principles.

To understand the algorithm, let's first decompose the transition sequence from the previous example. In Fig. 6, we show its associated sub-graph in the synthesized FSM of the MESI-CXL bridge. This sequence is the composition of the LP: $(S) \xrightarrow{GetM} (M) \rightarrow$, and GP: $(S) \xrightarrow{store} (SM) \xrightarrow{Cmp-E} (SM^D) \xrightarrow{Data} (M) \rightarrow$ original transition sub-graphs. Operationally, it denotes that from a bridge state (S,S), if it observes the LP dir transition: S+GetM, it resolves it by initiating the GP cc transition: S+store. We then detail how this transition sequence is synthesized to satisfy our 3 bridging rules.

Our *delegation* rule follows from our method to identify candidate origin and remote graphs. In this example, from the bridge state: (S,S), the GetM request must propagate as it will make subsequent updates to the cache line globally visible in LP. Thus, we retain all origin graphs in LP that match the prefix: $(S) \xrightarrow{GetM}$. To find the matching remote graphs, intuitively, we *repeat* the access of the requestor cache to the other domain. Since this transaction was initiated by a store cache access in LP, which equivalently maps here to a store for GP cc, we retain the sub-graph of GP cc with the prefix: $(S) \xrightarrow{store}$.

In the previous composition, we fully encapsulate the remote graph (GP) in the origin graph (LP), to satisfy our *nesting atomicity* rule. This guarantees that the remote flow appears to complete atomically from the origin domain, i.e., the coherence effects (global visibility of a store) are completely propagated to GP before continuing the LP coherence transaction.

In Fig. 6, notice that the remote nested sub-graph (GP) has multiple paths to its end state M + store-hit. Those paths must be preserved in the composition, to satisfy our *selective stalling* rule. Otherwise, the bridge may stall other GP *fwds* from the dir, for instance BISnpInv, leading to *deadlocks*.

In the remainder of this section, we present the generic synthesis pipeline and refine this intuition of bridge synthesis (Alg. 1). In particular, we answer the following questions: How to, systematically: *establish the translation rules between protocols that preserve original semantics* (§6.2), *identify which coherence transaction should be propagated across domains* (§6.4), and *correctly compose remote flows inside them* (§6.5).

Algorithm 1: vCXLGen's bridge synthesis algorithm

```
1  begin
2      Bridge.states ← LP.dir.stable_states × GP.cache.stable_states;
3      forall bstate as (lstate, gstate) in Bridge.states do
           /* l: local, g: global */
4          bTransL2G ← genTrans(lGraph, gGraph, lstate, gstate);
5          bTransG2L ← genTrans(gGraph, lGraph, gstate, lstate);
6          Bridge.FSMgraph.add(bTransL2G, bTransG2L);
7      pruneUnreachableStates (Bridge.FSMgraph);
8  end
   /* o: origin, r: remote */
9  def genTrans(oGraph, rGraph, oState, rState):
       /* All transition trees starting from oState */
10     forall oTransT in oGraph.sub_trees[oState] do
           /* Search for propagation rule in TLT */
11         oAccess ← TLT.get(oState,oTransT[0].label);
           /* No propagation rule: local-only flow */
12         if oAccess is undefined then yield oTransT; continue ;
           /* Propagation: translate to remote flow */
13         rAccess ← equivalentAccess (LP,GP,oAccess);
14         forall rTransT in rGraph.sub_trees[rState]
15          with rTransT[0].label == rAccess do
16             newTree ←
                nestGraph(oTransT[0],rTransT,oTransT[1:]);
17             yield newTree;
```

6.1 Synthesis Pipeline Overview

In Fig. 5a, we present the automated synthesis pipeline, which is organized in two passes. The first pass identifies flow translation rules (①) between domains. For that, it statically analyzes input protocols in isolation to detect which flow should be propagated, and infer semantic translation rules for them. The second pass then constructs the bridge FSM following 3 steps: ② generating the compound stable state space, ③ detecting transactions that must be propagated and translated when it populates *LP* and *GP* transitions in the bridge FSM graph, and ④ for every such transaction, perform nesting of the remote flow graphs that we match with our inferred translation rules.

6.2 Semantic Translation of Transactions

In a first pass, we statically analyze input protocols to establish semantic rules that indicate which transitions in the bridge, initiated by *local cache requests* or *global forwarded requests*, should be detected and propagated between domains, and to which universal access type they semantically map to.

Universal semantics. Our general approach to establish a semantic mapping relies on cache *access types* (load, store) to denote the effects of requests that should propagate between domains. We make the key observation that, for every propagated request, the original access type performed by the original requestor cache can be simulated, i.e., re-emitted in the other domain, to produce semantically matching effects.

For instance, the example in Fig. 5c shows that with a local MESI protocol, receiving a GetM request in the S local dir state originates from a store. In Fig. 6, for the same example, the store access is *repeated* or *simulated* by the global cache from its current state (S) to find all *remote transition graphs* that must be used to propagate the initial local GetM request.

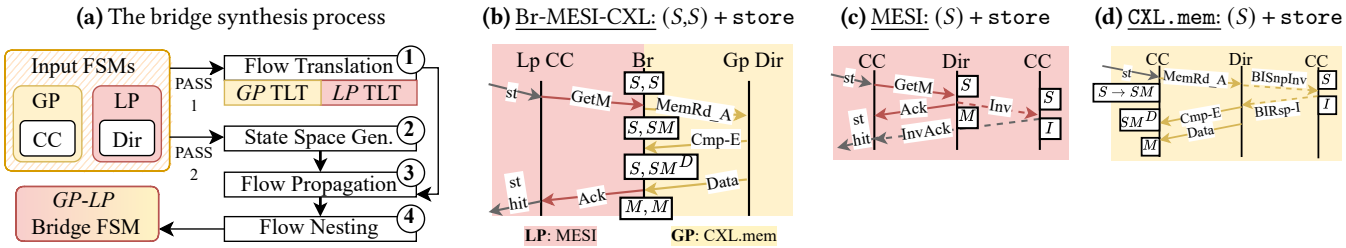


Figure 5. (a) The bridge synthesis process. Step ①: detection of translation rules. Steps ②-④: fusion of FSM graphs. **(b)** Sample synthesized transaction flow for a *MESI-CXL* bridge: *store miss from a shared state.* **(c), (d)** Original LP and GP transaction flows.

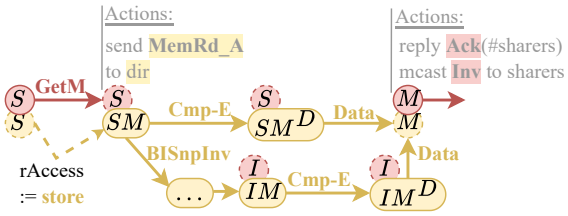


Figure 6. MESI-CXL bridge transition graph for (*S*,*S*)+`GetM`.

In cases where input protocols use different core accesses, e.g., weak protocols such as RCC also have *load-acquire* (`ldar`) and *store-release* (`stlr`) accesses; we rely on the ArMOR framework [57] to build an additional translation layer between them. For instance, in this framework, a release consistency (RC) `stlr` maps to a SC `store`. To query these mappings between heterogeneous core accesses, we define the function `equivalentAccess()` (line 13 in Alg. 1).

Flow translation tables (TLTs). We capture in TLTs all transitions that must be detected and propagated, along with the access requested by the initial requestor cache. Those transitions are denoted from their initial state and transition label. In our previous example (Fig. 5), the corresponding LP TLT entry is: S+`GetM`$\mapsto$`store`. Note that the transition labels are commonly cache requests for the LP TLT, and forwarded requests for the GP TLT. Translation tables are automatically constructed by our initial *static flow analysis* pass (§6.6).

6.3 Compound State Space Generation

In the second pass, we start by populating the bridge FSM with its stable states. Bridge states are a pair of LP directory and GP cache states, such that the bridge retains a *compound view* of cache line states for both protocols simultaneously. We initially generate the set of stable compound states as the Cartesian product of LP and GP state sets. (line 1 in Alg. 1) Transient states are created along the generation of bridge transitions in the next steps . At the end of the process, we prune all unreachable stable states in the bridge FSM graph.

In the example given in Fig. 5b, the state (*S*,*S*) denotes that the cache block is present in the bridge and that local sharers exist, while in (*I*,*S*) there is no local sharers. The transient state (S,SM^D) indicates that a global flow to upgrade to `M` is waiting on data. Non-inclusive states like (*S*,*I*)—which indicate local sharers without global read permissions—are pruned automatically with the `pruneUnreachableStates()` function (line 7 Alg. 1), as they remain unreachable because our translation rules enforce an inclusive hierarchy by construction.

6.4 Transaction Propagation

In the third synthesis step, we populate the bridge FSM with transitions. For every compound stable state, we iterate over the original transition graphs in LP then GP FSMs, that start from their individual stable states. We identify as *entire flows*, all graph sub-trees with a stable final state. If no transition labels in those sub-trees match with the TLTs, the entire flow does not propagate across domains and can be directly inserted in the bridge FSM. (line 12 in Alg. 1) Conversely, we *propagate* the transitions whose labels match the TLT rules.

Propagated transitions identify the point in a transaction from which the bridge must delegate to a transaction flow into the other domain. We find which remote graphs should be nested inside the origin transaction graph, by collecting all remote sub-trees that start from the current remote stable state with the access that mapped to it in the TLT.

In the example from Fig. 6, we populate transitions for the compound state (S,S). For that, we iterate over all flow sub-trees in LP from state `S`, and GP from state `S`. We detect from our translation rules, that from `S`, transitions in LP with the label `GetM` must be intercepted and translated to a `store`. For those, we then search in GP for matching sub-trees that begin in state `S` with the label `store`, and then proceed to nest those sub-trees in the initial `GetM` transition sub-graph.

6.5 Transaction Nesting

In the fourth and last synthesis step, we finally nest inside *propagated flows*, all remote sub-trees that mapped to it in our TLT.

We do not insert the *first transition* of the origin graph in the bridge FSM, but instead, all remote sub-trees that match the translation rules. We *nest* or *compose* those remote sub-trees by updating their first and final transitions. Specifically, we update in the first transition the initial state and label to correspond to those of the origin transition, and in the final transition, we update the terminal state and action to correspond to those of the origin transition.

In the example from Fig. 6, the *propagated* transition has the initial state *S*, the label *GetM*, the final state *M* and the action

GetM_AckD. The remote sub-tree is then updated, such that it begins with a transition from the start state (*S*,*S*) and label *GetM* instead of *store*, and that its final transition ends with state (*M*,*M*) and with the *GetM_AckD* action.

6.6 Static Flow Analysis

To construct our flow TLTs, we statically analyze each input protocol in isolation. We generate all communication traces between cache and directory controllers, and collect complete *flows*, as distinct FSM transactions forming a complete transition sequence from a stable state and core access (label) to final states where the cache obtains requested permissions. For upstream translation (LP), we detect flows producing globally visible memory operations, i.e., containing a memory read/write . For downstream translation (GP), we detect flows that downgrade cache permissions. For all detected flows, we create a TLT entry that maps the stable start state and first message name (label) in the transaction's trace, to the core access of the original requestor cache, such as S+`GetM`$\mapsto$`store` (Fig. 5). Each entry identifies a transaction that must be propagated. When propagating transactions, we select the correct translated label by simulating the mapped core access to the other domain, intuitively *"repeating"* the requestor's initial access.

6.7 Transient States

Transient states are protocol-specific intermediate states in a coherence transaction that define controller behavior in the face of concurrent requests, capturing vendor-specific optimizations. While input FSMs from parsed PCC specifications include transient states for their respective domains, CXL bridges also propagate transactions across domains, requiring our synthesis algorithm to define new transient states for bridges to manage concurrency at the LP-GP interface.

Specifically, for each propagated transaction, we synthesize a new transient compound state that logically stalls other requests from the origin domain to preserve *atomicity*. While the nested transaction is in-flight, it also stalls concurrent requests from the nested domain only if they must propagate further to the origin domain and the nested domain is LP, but we allow propagation when the nested domain is GP, necessary to satisfy our *selective stalling* rule and avoid deadlocks.

To understand the reason, consider the example in Fig. 6. After propagating the LP request `GetM` to GP, the bridge cannot stall GP snoops (e.g., `BISnpInv`) while it waits for completion, as the GP directory may be serializing another transaction first. In Fig. 6, from the transient state (S,SM), the bridge can observe completion (`Cmp-E`) or a snoop (`BISnpInv`). In the original GP FSM, the snoop transitions the cc from SM to IM transient states, removing its read permissions. Meaning, this snoop must propagate first to invalidate the entire LP domain and transition the bridge to (I,IM), before the bridge can receive completion (`Cmp-E`) for its nested transaction.

In practice, we achieve this by running our nesting algorithm again inside a nested transaction from existing GP cc transient states, which are captured in TLTs in entries such as $SM + BISnpInv \mapsto store$, that we create when we statically analyze GP FSM and detect a downgrade of GP cc permissions.

6.8 Discussions

Non-SWMR global protocol. A GP cannot be weaker than a host's LP without violating the host's coherence assumptions and breaking its original MCM. Since SWMR protocols are common in CPUs, vCXLGen supports SWMR-enforcing GP protocols, like `CXL.mem` or other MESI-family protocols.

In systems with only RC devices, using RCC as GP is possible, and we see no conceptual barrier to vCXLGen synthesizing an RCC-RCC bridge. The key is to ensure that the `acquire` fence also propagates from LP to GP. For that, we only need the LP-RCC protocol specifications to differentiate read-requests on load-acquire (`ldar`), e.g., with a distinct `GetA` request, which textbook RCC lacks. After updating the PCC file accordingly, vCXLGen will automatically detect it and handle the LP-`ldar` to GP-`ldar` translation. Other RC devices (e.g., GPU) may also implement `acquire`/`release` fences with cache maintenance messages, like `invalidation` or `flush`, that can be similarly specified in PCC. However, performing an GP-acquire is inefficient since it invalidates the entire cluster's shared cache. While scoped memory accesses (cluster and system scope) typically solve this by restricting access and fence propagation, CPUs generally lack this support.

Vendor-specific optimizations. Vendors can freely specify relaxations and optimizations within their LP domain via PCC specifications, which vCXLGen preserves, as original LP and GP protocols are never modified. Our compositional verification guarantees GP-equivalence between the bridged LP cluster and a CXL caching client, intuitively showing the bridged cluster externally behaves like a normal CXL cache client, regardless of vendor-specific internal optimizations. For example, vCXLGen bridges both strong protocols like MESI and relaxed protocols like RCC. A MESI-CXL bridge enforces SWMR system-wide, while an RCC-CXL bridge only globally orders stores marked as release. Thus, LP relaxations are preserved and vCXLGen propagates only necessary requests, avoiding creating overly strong compound protocols.

Non-atomicity. vCXLGen's protocol composition enforces atomicity at the LP-GP interface through synchronous nesting of propagated transactions. This design enables `CXL.mem` integration without changing native LP behaviors or redesigning internal host FSM states, and does not restrict existing concurrency in LP and GP domains. Our logical blocking of requests is a standard practice for hierarchical cache controllers, as in ARM's CHI protocol [12], local coherence directories in multi-socket platforms, or gem5's MOESI LLC, which all await a response from the next hierarchical level.

Non-atomic propagation might seem optimal—e.g., in Fig. 6, sending the translated GP request (`MemRd_A`) and local `Inv` simultaneously could save 1 local message delay. However,

T_1 (TSO)	T_2 (RC)
st_1: `st` #1 X;	ld_1: `ldar` r_1 Y;
st_2: `st` #1 Y;	ld_2: `ld` r_2 X;

(a) TSO producer, RC consumer

T_1 (RC)	T_2 (TSO)
st_1: `st` #1 X;	ld_1: `ld` r_1 Y;
st_2: `stlr` #1 Y;	ld_2: `ld` r_2 X;

(b) RC producer, TSO consumer

Figure 7. Heterogeneous MP litmus test. In both, $st_1 \longrightarrow st_2$ & $ld_1 \longrightarrow ld_2$, thus, the outcome $\{r_1 = 0, r_2 = 1\}$ is disallowed.

in this example, the requestor could then begin accumulating `Inv_Acks` from local sharers before the global directory serializes its request. If a GP snoop (`BISnpInv`) arrives next, the local requestor will not respond to the translated local `Inv`, causing a deadlock. Conceptually, the LP PCC could avoid it by being GP-aware and integrate ad-hoc mechanisms for each propagated transaction to ensure correct LP-GP concurrency; though these changes to the LP FSM are intrusive and GP-specific. A general mechanism to prevent message races in CC protocols is handshaking, which is conceptually similar to our synchronous transaction nesting.

7 CXL Bridge Memory Consistency Model

In this section, we detail the memory consistency of vCXLGen, resulting from the composition of host's local protocol with CXL through generated CXL bridges.

7.1 Heterogeneous Memory Semantics

We design vCXLGen to generate CXL bridges that ensure each host observes its MCM regardless of concurrent accesses from other hosts. Meaning that by construction, CXL bridges always realize a *compound memory model* – as rigorously defined in [38, 68] – when combining heterogeneous MCMs.

Informally, for any two memory operations (o, o'), when a host's MCM mandates an ordering constraint $(o \longrightarrow o')$ within its threads, our CXL bridges also *atomically propagate* this ordering constraint to *all system threads*, and *in the same order* (o *propagates before* o'); because o' is stalled until o completely propagates. Overall, this ensures that propagation and visibility of operations follow their local dependency order constraints. That is, memory operations propagate and become *globally visible* system-wide following their local ordering, such that each host can view the CXL memory as adhering to its own local MCM axioms, irrespective of other hosts.

In examples from Fig. 7, programmers expect re-ordering constraints at each individual MCMs through the use of host-specific instructions. For instance, the constraint $c_1 : st_1 \longrightarrow st_2$ follows from TSO's `st` or RC's `stlr`, and the constraint $c_2 : ld_1 \longrightarrow ld_2$ follows from TSO's `ld` or RC's `ldar`. Each of those constraints c_1 and c_2 are propagated individually by their emitter threads T_1 and T_2 through their respective CXL bridges; overall guaranteeing that every local thread propagates their MCM ordering constraints globally. This means memory operations become globally visible in the same order to all system threads, and not only to host-local threads.

7.2 An Axiomatic MCM for SC-vCXLGen-RC

We now detail the axiomatic semantics of a heterogeneous MCM composition, showing how CXL bridges realize compound memory consistency. We define the system-wide memory consistency model where the two coherence domains follow SC and RC individually and are interconnected via vCXLGen's bridges and a CXL domain. Our axiomatic formal semantics follow the standard definitions in the 'cat' language such as program-order (po), read-from (rf), from-read (fr), coherence-order (co), atomic read-modify-write (rmw), po on same location ($\text{po}|_{\text{loc}}$) [5].

Axiomatic semantics of SC+RC. We define the axioms of the SC-vCXLGen-RC consistency model by combining the SC and RC models with vCXLGen.

Coherence: Coherence axiom denotes that the same-memory events are in a total order in an execution. The total order is enforced by the same-location relations $\text{po}|_{\text{loc}}$, rf, co, fr.

$$(\text{po}|_{\text{loc}} \cup \text{rf} \cup \text{co} \cup \text{fr}) \text{ is acyclic} \qquad \text{(Coherence)}$$

Coherence is ensured in both SC and RC, respectively, and the protocols enforced by vCXLGen ensure that coherence is satisfied system-wide.

Atomicity: Atomicity axiom enforces that a pair of rmw-related events has no intermediate event on the same-location.

$$\text{rmw} \cap (\text{fr};\text{co}) \text{ is empty} \qquad \text{(Atomicity)}$$

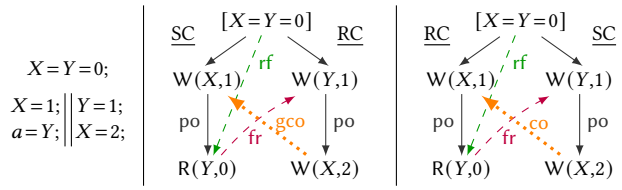
The (Atomicity) axiom ensures that if $\text{rmw}(r, w)$ holds in an execution then the execution has no other write w' such that $\text{fr}(r, w')$ and $\text{co}(w', w)$ hold. (Atomicity) axiom is ensured in SC and RC individually. The SWMR GP protocol and vCXLGen propagation ensure that the (Atomicity) axiom is preserved in an SC+RC execution globally.

Global ordering: The SC-vCXLGen-RC model enforces a global ordering by *globally-happens-before* (ghb) relation from the protocols generated by vCXLGen. The ghb relation in SC-vCXLGen-RC combines the *happens-before* (hb) relations in SC and RC, hb_{SC} and hb_{RC} respectively. Note that the SC and RC models vary in the respective global orderings. For example, in SC model, hb_{SC} relation is defined by combining po, rf, co, and fr relations. Finally, the SC model enforces that hb_{SC} is acyclic. In RC model, hb_{RC} definition is more subtle and may vary based on the specific events and fences. In RC model, $\text{hb}_{\text{RC}} \cup \text{co} \cup \text{fr}$ is acyclic. vCXLGen enforces that if a read event reads-from a write event of another domain then it establishes synchronization. We denote the cross-domain rf-relation by grf and define the global ordering ghb. We also write gco that is a co relation from RC to SC domain. In an execution co and fr accesses follow the $\text{ghb}|_{\text{loc}}$ (ghb on same-location events) order. Finally, SC-vCXLGen-RC model enforces the global ordering axiom (Ord) as follows.

$$(\text{ghb}|_{\text{loc}} \cup \text{co} \cup \text{fr}) \text{ is acyclic} \qquad \text{(Ord)}$$
$$\text{where } \text{ghb} \triangleq (\text{hb}_{\text{SC}} \cup \text{grf} \cup \text{hb}_{\text{RC}} \cup \text{gco})^{+}$$
$$\text{and both } \text{hb}_{\text{SC}} \text{ and } \text{hb}_{\text{RC}} \text{ are acyclic}$$

Thus, an execution is consistent under SC-vCXLGen-RC execution if it satisfies all these axioms. vCXLGen ensures that the protocols preserve the consistency globally if SC and RC domains individually ensure consistency.

Example. Consider the program below and its execution $a=0, X=1$ in SC, RC, and SC-vCXLGen-RC. The execution is forbidden in SC as it creates a hb_{SC} cycle. The execution is allowed in RC as it does not violate any axiom. The execution is forbidden in SC+RC when the first thread is in SC and the second thread is in RC as shown in the first execution. In this case, $W(Y,1) \xrightarrow{\text{po}} \xrightarrow{\text{gco}} \xrightarrow{\text{po}} R(Y,0)$ results in ghb and consequently creates a ghb∪fr cycle which is forbidden. In the second execution, there is ghb ordering across the threads and the execution $a=0, X=1$ is allowed.



8 Automatic Verification of CXL Bridges

We next detail our model checking approach to verify the functional correctness of the generated CXL bridges. First, we review the criteria for a correct bridge (§8.1) and our automated verification methodology (§8.2). Then, we present our automated compositional verification contribution (§8.3).

8.1 Verification Properties

We define a correct CXL bridge as one that satisfies the following properties. For every reachable system state:

- **Safety:** No state transitions sequence violates any of the LP MCM litmus tests.
- **Deadlock-freedom:** There is always a different state to transition to.
- **Extended liveness:** For all cache controllers *cc*, addresses *a*, and access permissions *p* there is a path from this state to a state in which *cc* has permissions *p* for address *a*.

Rationale. The safety property verifies that our bridges do not violate the MCMs of all the LP clusters, as characterized by a set of litmus tests. Deadlock freedom alone is insufficient to guarantee progress. A system where only one cluster or cc changes its internal state is considered deadlock-free, even if all other clusters and components are stuck forever. Therefore, we define and verify the extended liveness (EL) properties as well to guarantee that all memory operations can terminate so all cores can make progress.

System models. We do not analyze bridges in isolation, we instead model the *full system* that integrates them. For our verification, we consider multi-cluster systems, with a CXL cluster and two heterogeneous sub-clusters. The sub-clusters contain ccs and a bridge that connects them to CXL. The CXL cluster only contains a directory and the bridges to the sub-clusters.

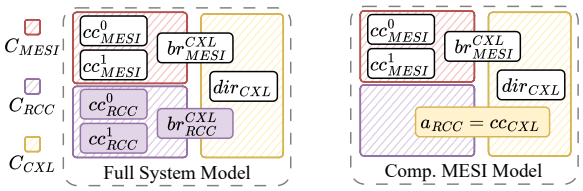


Figure 8. Example full system model (left) has the same GP (C_{CXL}) behavior as the compositional MESI model (right).

T_0 (SC)	T_1 (SC)	T_2 (RC)	T_3 (RC)
`ld` r_1 X;	`ld` r_1 X;	`stlr` #1 X;	`stlr` #2 X;
`ld` r_2 X;	`ld` r_2 X;		

Figure 9. Heterogeneous CoRR2 litmus test.

8.2 Automated Verification

We automate verification by creating a vCXLGen back-end to generate system models for model checkers. All of these models contain FSMs of all system components, connected through a network to exchange messages.

Safety verification. We verify adherence to LP MCMs using litmus tests. Litmus tests are small, multi-threaded programs. They specify which outcomes are allowed under some MCM.

We map each thread of the litmus tests to ccs, generating a separate model for each possible assignment of threads to clusters. The ccs issue memory operations according to the litmus test instructions. The model explores all possible orders in which the memory system could handle these operations. It ensures that no forbidden outcome of the litmus test can be reached. We allow any set of litmus tests as input to vCXLGen. In our evaluation, we obtain litmus tests for a heterogeneous architecture from HeteroGen [68]: These use litmus tests from the weaker MCM as the basis and remove fences that are unnecessary when combined with the stronger MCMs.

Consider the *full system* model depicted in Fig. 8 (left). It consists of a MESI (C_{MESI}) and RCC (C_{RCC}) sub-cluster, containing cache controllers (cc^i_{id}), bridges (br^{CXL}_{id}) and a directory (dir_{CXL}). Suppose we want to generate a model for the heterogeneous CoRR2 litmus test, depicted in Fig. 9. The threads are mapped according to their MCM: $T_0 \rightarrow cc^0_{MESI}, T_1 \rightarrow cc^1_{MESI}$ and $T_2 \rightarrow cc^0_{RCC}, T_3 \rightarrow cc^1_{RCC}$. Modeled are two load requests for each cc^0_{MESI} and cc^1_{MESI} and one store-release request for each cc^0_{RCC} and cc^1_{RCC}. Using exhaustive exploration, the model checker ensures that the forbidden outcomes of the litmus test $T_0\{r_1 = 1, r_2 = 2\}$ $T_1\{r_1 = 2, r_2 = 1\}$ and $T_0\{r_1 = 2, r_2 = 1\}$ $T_1\{r_1 = 1, r_2 = 2\}$ are impossible to reach.

Liveness verification. We verify liveness in an unrestricted *full system* model by defining atomic prepositions for each cache and access permission; the preposition is true in all states where the cache has the specified access permission. The model checker ensures, for each preposition, that all reachable states have a path to some state where it holds. This allows us to verify that the memory operations of caches are eventually served. Deadlock-freedom is checked implicitly, ensuring all reachable states can transition to a different state.

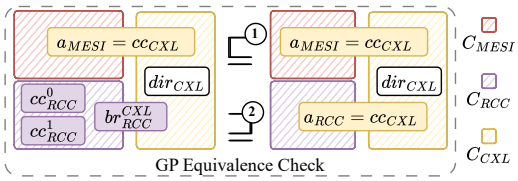


Figure 10. GP-Eq check verifies all GP behaviors of C_{RCC} are reproducible by a_{RCC} (① $C_{RCC} \sqsubseteq cc_{CXL}$) and all GP behaviors of a_{RCC} are reproducible by C_{RCC} (② $cc_{CXL} \sqsubseteq C_{RCC}$).

Challenges. The complexity of our models causes a state-space explosion, with an equal explosion in memory consumption and verification time—an observation consistent with prior work [67, 68]. We were able to run the safety models on our 1.8TB server by optimizing the model state representation. The liveness models, however, are even more resource-hungry due to exploring all possible sequences of core accesses , leading to out-of-memory failures. Additionally, extended liveness properties significantly increase verification time, making conventional liveness verification of multi-cluster systems practically infeasible. As a solution, we next propose a novel compositional approach that scales better and makes liveness verification tractable.

8.3 Scalable Compositional Verification

Compositional verification breaks the *full system* model into multiple smaller *compositional* models (see Fig. 8), one for each LP cluster. These models replace the remaining LP clusters with *cluster abstractions*. These abstractions only match the GP behavior of the full clusters they replace.

Intuitively, LP clusters can only affect each other by proxy, through the GP requests their bridge emits and receives. As such, the progress of one specific LP_x cluster depends only on the messages from its own LP network and the GP network. In our compositional models, we then verify that: the progress in GP of any other cluster in the system does not hinder AND is not hindered by the progress of any ccs in the LP_x cluster.

This way, verifying deadlock-freedom and extended liveness properties in all compositional models guarantees that these properties hold for the full system.

Cluster abstractions. We use a cc directly connected to the CXL network as the abstraction for each LP cluster. The GP ccs' FSMs are reused from the bridge synthesis step. We propose an automated GP equivalence check (GP-Eq) to prove that these FSMs have the same GP behavior as the full clusters.

Equivalence. We ensure GP equivalence (GP-eq) by comparing the GP communication of the full cluster and its abstraction. This check is *fully automated* using two symmetric models: one verifies that the abstraction can reproduce all behaviors of the full cluster, and the other ensures the full cluster can reproduce those of the abstraction. If both hold, we consider the abstraction GP-equivalent to the full cluster.

To have models that verify the reproducibility of behavior, we explore the full cluster and its abstraction simultaneously. Whenever one of their FSMs produces a state change visible to the other clusters (i.e., a GP message), the other is verified to be able to reproduce the same state change. Additional logic ensures that the same holds for blocking behavior, where the full cluster/abstraction enters a state from which no visible state changes are possible.

Example. We illustrate the compositional approach on the full system depicted on the left in Fig. 8, with the end goal of decomposing it into two compositional models: (1) One for C_{MESI} where an abstraction a_{RCC} replaces C_{RCC} (right of Fig. 8). (2) A second one for C_{RCC} where abstraction a_{MESI} replaces C_{MESI}. This leaves us with one model for each bridge in the system: one for br^{CXL}_{MESI} and one for br^{CXL}_{RCC}.

We use a CXL cache controller $cc_{CXL} = a_{RCC} = a_{MESI}$ as the cluster abstraction. To ensure that cluster abstractions can safely replace their respective LP models, we perform two partial GP-Eq checks for each abstraction–cluster pair: One ensures that a_{id} can reproduce all the behaviors of C_{id}, and the other verifies the opposite direction.

If all GP-Eq checks succeed for both C_{MESI} and C_{RCC}, then the cluster abstraction $a_{RCC} = a_{MESI} = cc_{CXL}$ is considered valid and can safely replace their corresponding LP clusters. We then verify liveness properties for each MESI cache controller cc^i_{MESI} using a model with br^{CXL}_{MESI} and a_{RCC}, and likewise for each cc^i_{RCC} using a model with br^{CXL}_{RCC} and a_{MESI}.

Scalability. Instead of exploring the *full system* model (left of Fig. 8), this approach explores all the compositional models (e.g., right of Fig. 8) and GP-Eq checks (e.g., Fig. 10) to show liveness. These models are expected to have a smaller state space individually. This is because they use abstractions for other clusters, and the GP-Eq checks exploit high symmetry.

9 Evaluation

We next evaluate vCXLGen across five dimensions: *generality* (§9.1), *extensibility* (§9.2), *correctness* (§9.3), *scalability* of compositional verification (§9.4), *performance* using multi-threaded benchmarks (§9.5) and CXL-based KVS (§9.6).

Hardware and systems. The test machine is a dual-Intel SPR 6438Y+ hyperthreaded 128-core server with 2TB of main memory. It runs linux 6.0 with gcc 13.3 and glibc 2.40. We use gem5 v24.1, CMurphi v5.4.9, and *Rumur* v2024.7.14. vCXLGen generator implemented in Python has about 18,000 SLOC.

9.1 Generality

We first evaluate the generality of vCXLGen by testing its ability to create CXL bridges across various compound protocols. We cover two protocol categories: those with the SWMR invariant (common in CPUs) and those targeting RC models (common in GPUs). For SWMR, we use MSI, MESI, MOESI, and CXL; for RC, we use RCC and RCC-O. Our generator successfully produces correct *compound protocols* from any combination of these inputs. While all SWMR and RC protocols function as LP, only SWMR protocols can serve as the

Input Protocols LP	Input Protocols GP	MCM	States/ Transitions LP_Cache	States/ Transitions GP_Dir	States/ Transitions Bridge
MESI	MESI	SC	13/41	6/45	36/195
MSI	MESI	SC	11/32	6/45	34/184
MESI	MSI	SC	13/41	4/26	36/195
MESI	MOESI	SC	13/41	5/69	36/203
MOESI	MSI	SC	17/57	4/26	43/298
RCC	MSI	RC	20/37	4/26	16/55
RCC-O	MESI	RC	14/30	6/45	19/72
MSI	CXL	SC	11/32	12/74	44/220
MESI	CXL	SC	13/41	12/74	48/263
RCC	CXL	RC	20/37	12/74	23/83
RCC-O	CXL	RC	14/30	12/74	26/101

Table 2. List of input protocol combinations, with the resulting MCM, states, and transitions of the generated FSMs.

GP. Tab. 2 summarizes the input protocols, their memory consistency models, and the resulting states and transitions in the generated compound protocol components. We run these protocol combinations on a wide range of applications in §9.5.

9.2 Extensibility

The modular design of vCXLGen allows for easy extension to support new protocols in the future. The `CXL.mem` protocol specifications are only about 650 SLOC of Protogen [66] DSL. MSI-family specifications are about 350 SLOC, and 200 SLOC for RC. vCXLGen automatically detects translation rules between input protocols; thus, writing a specification file is the only required step to support new protocols.

9.3 Correctness

We validate the correctness of vCXLGen's generated bridges and use its verification back-end to produce Murφ-compatible system models. We check liveness and adherence to our MCM.

Liveness. For the liveness analysis, we generate Murφ and *Rumur* models that verify *deadlock-freedom* (DF) and extended-liveness (EL). Tab. 3 shows methods we use to verify different systems; one method suffices to verify the property.

Safety. We generate models for 216 litmus tests using the herd7 [5] tool. We extend those to account for heterogeneous MCMs [68] (see §8.2). We cover common checks like *IRIW, MP, 2+2W, CoRR1, CoRR2, LB, R, RWC, S, SB, WRC, WRW+2W, and WWC* [70], with variants for SC and RC threads. We generate one model for each combination of: litmus test, distribution of threads across the two LP clusters, and initial state of the caches. We allow bridges to self-evict between any two instructions, as this produces globally visible behavior. Exhaustive exploration of these models determines all possible operation re-orderings and ensures none break our MCM. We show in Tab. 4 that for all setups, the results of the litmus tests are consistent with the behaviors allowed by our MCM — each cluster retains its original MCM when merged with CXL.

Takeaways. We find no violations of our MCM in our 216 litmus test models, and no deadlocks or liveness issues using multiple verification methods—confirming the correctness of our generated protocols.

Compound Protocols LP_1	Compound Protocols GP	Compound Protocols LP_2	2 Caches per LP DF	2 Caches per LP EL	3 Caches per LP DF	3 Caches per LP EL
MESI	MESI	MESI	M,R,C	R,C	M,R,C	C
MESI	CXL	MESI	M,R,C	R,C	M,R,C	C
MESI	CXL	RCC	M,R,C	R,C	C	C

Table 3. Verification strategy for showing deadlock-freedom (DF) and extended liveness (EL), with full-system Murφ (M), full-system *Rumur* (R), and compositional models (C).

Compound Protocols LP_1	Compound Protocols GP	Compound Protocols LP_2	Litmus Test Setup C1	Litmus Test Setup C2	Succeeding Models
MESI	MESI	MESI	SC	SC	216 / 216
MESI	CXL	MESI	SC	SC	216 / 216
MESI	CXL	RCC	SC	RC	216 / 216
RCC	CXL	RCC	RC	RC	216 / 216

Table 4. The number of succeeding litmus test models that are consistent with our MCM for varying protocols[1].

9.4 Scalability of Compositional Verification

Next, we evaluate the scalability of our compositional model checking approach compared to that of full-system models with medium and large system complexities;

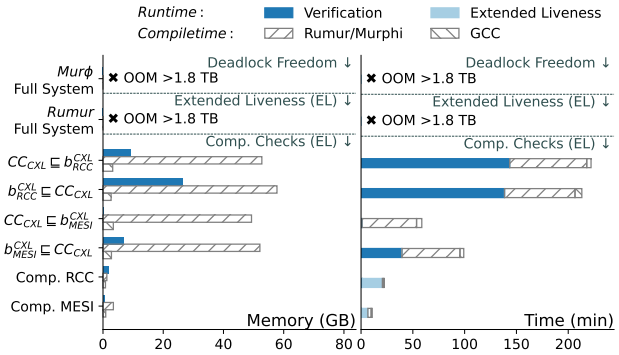
Methodology. We measure execution time and memory consumption. We build on the HeteroGen [68] verification back end and extend it to generate multi-cluster Murφ-compatible models, which cover: deadlock-freedom (DF) only with *full-system* Murφ, or DF and extended liveness (EL) with *full-system Rumur* and our novel compositional checks. We choose two compound protocols, shown in Fig. 11, to evaluate the approach for medium and large system complexities.

Memory usage. Fig. 11 shows compositional checks consistently reduce verification memory footprint compared to the optimized full-system checks. The larger the system, the larger the reduction of the memory footprint: 92% for the medium and over 98% for the large system. However, our compilers— Murφ,*Rumur*, and gcc—also have a significant memory footprint. Factoring these in reduces the medium and large systems improvement to 86% and over 96%, respectively. Note that the large full-system models run out of memory with 1.8TB of RAM; the 98%/96% improvement reported above is relative to this 1.8 TB, the real improvement compared to what these models require to *complete* verification is likely larger.

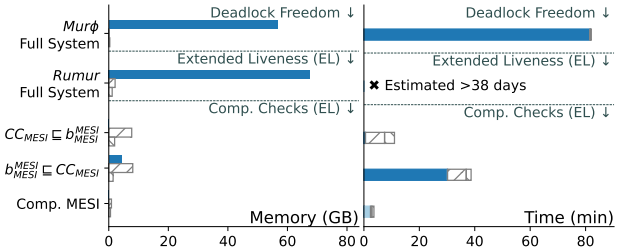
Execution time. Compositional checks also reduce verification time for large systems. For our medium setup Fig. 11 shows a 35% improvement in overall deadlock freedom verification and a 99% improvement in extended liveness properties verification time according to our estimates (from *Rumur* reports of properties verified per minute, after 24h of run time).

Compilation. The compilation of the *Rumur* models dominates the resource consumption of the compositional models. These compilation overheads mainly depend on the complexity of the fused protocols and not the number of ccs, which

[1] Although MESI enforces *SC-per-location* only, we check for SC because our litmus models preserve instruction program order. Should our bridge fail to preserve *SC-per-location*, the litmus test outcomes will also violate SC.



(a) Large Model, [MESI, CXL, RCC], **3** caches per cluster.



(b) Medium Model, [MESI, MESI, MESI], **3** caches per cluster.

Figure 11. Breakdown of memory footprint and time required to verify deadlock-freedom (DF) and extended liveness (EL). Murφ Full System only shows DF, *Rumur* Full System shows DF+EL, all compositional checks *together* show DF+EL.

causes the compositional approach to perform worse than full-system models for small systems (2 ccs or less).

Parallelism and reusability. With large enough system memory, all compositional checks can be run concurrently. Furthermore, it is possible to reuse checks from previous runs or symmetry in the system: For our medium setup, we utilize symmetry to eliminate half of the compositional checks. If the system changes, we only need to verify the new cluster.

Takeaways. Compositional verification enables liveness verification of larger 2×3 systems, using less than 60GB—where the baseline fails even with 1.8 TB. Even with medium complexity systems, only our compositional verification verifies EL properties in a feasible time span.

9.5 Performance

Testbed. We evaluate the performance overheads of generated CXL bridges using *gem5* [20, 56], by modeling a CXL multi-host memory system, similar to Fig. 3, with two clusters connected via a high-latency interconnect. vCXLGen generates compound protocols and emits gem5-compatible SLICC code for caches, bridges, and directory controllers. Bridge controllers replace each cluster's shared last-level cache (LLC) and interact with the CXL directory over a high-latency link. We rely on gem5's Garnet network model [18] to simulate communication between hosts and CXL memory, rather than using dedicated PCIe-based CXL simulation models [84, 85]. Although Garnet was originally designed as an on-chip network and real CXL systems communicate over a PCIe fabric, Garnet is tailored for CC protocols which aligns with our focus on protocol bridging. We use its flexible network configuration (link latency, bandwidth, flit size) to align with CXL topologies. It lets us isolate performance effects stemming from protocol logic and vCXLGen from the PCIe transport overheads

Cluster 1-2	4-16 cores, x86, 8-wide OoO, L1-cache: 128 KB, 8-way, private, LRU, 1 cycle controller latency
LLC	4 MB, 8-way, shared, inclusive, LRU
Interconnect	intra-cluster: point-to-point topology, 10 clk link latency cross-cluster: star topology, 70 ns link latency common: static routing, 72B per flit, 1 clk router latency
Memory	DDR5, 4400 Mhz, 8GB, 10 ns controller latency

Table 5. Evaluation system parameters.

We run all benchmarks in *gem5*'s *syscall emulation* (SE) mode, first, under worst-case conditions: with all memory accesses directed to remote CXL memory. In §9.6, we then use a distributed KV store with a hybrid memory configuration, to also evaluate bridges for real-world applications with both local and remote memory. We calibrate the interconnect latency to reflect that expected for remote CXL memory (450 ns RTT, from cores to memory). We also omit per-core L2 caches for fair comparison with the baseline (unsupported by gem5's MOESI protocol). Detailed parameters are listed in Tab. 5.

Workloads. We use a total of 35 parallel applications from common parallel benchmark suites: *Splash-4* [39], *PARSEC* [1, 19] and *Phoenix* [71]. They cover all common shared-memory communication patterns, including message passing, migratory sharing, and producer-consumer communication [15]. To keep simulation time tractable while preserving representative coherence traffic, we use scaled-down workload inputs and proportionally smaller L1 and LLC sizes, verified to maintain a miss ratio (MPKI) similar to bare-metal runs with larger inputs. With our configuration, each workload generates at least 8 million coherence transactions, which accurately represent real hardware access patterns and CXL memory latency.

Protocols. As a baseline, we use a hypothetical ideal system with a homogeneous protocol across all clusters using modified *MOESI_CMP* from *gem5* with an inclusive LLC (noted **MOESI**). We compare this against two protocol combinations generated by vCXLGen: [*MESI, MESI, MESI*] (**MESI-Br**) and [*MESI, CXL, MESI*] (**CXL-Br**), allowing us to isolate performance impacts of bridging from CXL-specific coherence.

Results. Fig. 12 presents performance trends across our applications, showing means for the three benchmark suites and two representative outliers. MESI-Br performs similarly to MOESI (within ±2% for most workloads, with a max of 10%), confirming minimal bridge logic overhead. For fluidanimate, MESI-Br even outperforms MOESI by 20%. CXL-Br matches MESI-Br in most applications, except in seven (mostly Phoenix), with an average overhead of up to 20.3%.

Analysis. To understand these performance variations, Fig. 13 analyzes LLC hits and misses for *load* and *store* requests, showing accumulated cycles for different LLC transactions

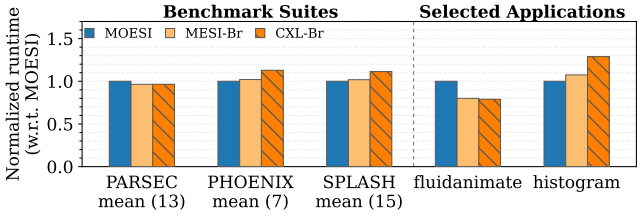


Figure 12. Performance of generated CXL bridges by vCXLGen. Mean over the three benchmark suites (35 applications) and two representative outliers.

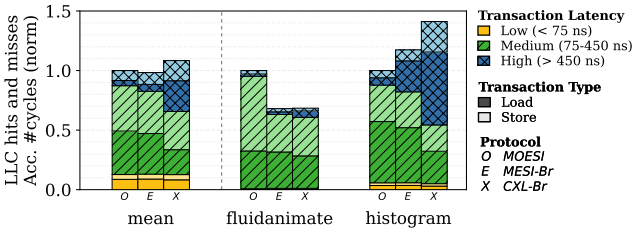


Figure 13. Breakdown of LLC hit/miss stalled cycles (normalized to MOESI). Mean over all applications, and for fluidanimate and histogram where CXL respectively under- and outperforms the baseline.

normalized to the MOESI. We categorize LLC transactions into three latency groups: (i) *low* captures transactions completed under 75 ns which corresponds to the latency of LLC hits or local misses (intra-cluster); (ii) *medium* captures transactions completed between 75 ns and 450 ns which corresponds to CXL remote memory accesses; and (iii) *high* captures transactions completed above 450 ns which corresponds to cross-cluster data exchanges or invalidation. We use 450 ns as the *medium*/*high* cutoff as it is the nominal latency (round-trip time) for memory requests in our simulation testbed.

For brevity, we present the mean across all 35 benchmarks and two applications representative of the two observed performance trends. First, in FLUIDANIMATE, MOESI spends 26% and 24% more cycles on *medium* LLC store misses only, compared to MESI-Br and CXL-Br respectively. As *gem5*'s *MOESI* lacks the E state, *write-after-read*-heavy applications like fluidanimate generate more LLC store misses. In contrast, both MESI-Br and CXL-Br properly manage the E state across local and global domains, allowing exclusive L1 caches to self-upgrade to M state without additional store misses. Second, in HISTOGRAM, MESI-Br spends 17.5% more cycles on *high* LLC load misses only, compared to MOESI. The cause is MESI-Br's synchronous memory write-backs required before forwarding dirty data on sharing requests (e.g., `FwdGet_S`). MOESI's Owned (O) state enables direct sharing of dirty cache lines without memory traffic–the owner cache supplies data directly to the requestor instead of first writing back to memory. This avoids stalling at the directory controller, improving transaction pipelining and overall reducing latency.

A similar effect exists with CXL-Br but is exacerbated because `CXL.mem` uses even more synchronous communications with the memory controller when invalidating the remote cluster, resulting in 20.3% extra total LLC transaction cycles compared to MESI-Br, for both loads and stores in HISTOGRAM.

For write requests with remote invalidation, `CXL.mem` uses 6 remote message delays when the owner is dirty (vs. 3 in MESI-Br), increasing latency. `CXL.mem` also incurs 2 blocking transient states at the CXL directory, further preventing request pipelining; whereas MESI-Br directly transfers data and ownership with `Fwd_GetM` to the requestor's bridge, without blocking the directory, which favors request pipelining.

For read requests with remote invalidation, `CXL.mem` uses only 1 additional message delay (4 vs. 3 in MESI-Br) but also has a blocking transient state at the directory for data write-back to memory. In MESI-Br, the previous owner directly sends data to the requestor on `Fwd_GetS` without waiting for memory write-back. We identified that for both loads and stores, CXL-Br suffers from *convoy effect* because of those directory blocking transient states in `CXL.mem`. Indeed, we observed in workloads like histogram some cache lines are memory hot-spots[2], accessed by both clusters for writes, but unlike MESI-Br, CXL-Br does not avoid synchronous memory write-backs on `Fwd_GetM`, which causes subsequent dependent LLC read-misses-to-memory to be delayed above 450 ns, pushing them from the medium to high-latency group.

Note that CXL-Br uses more directory handshaking and lacks peer-to-peer responses between hosts compared to MESI-Br, because `CXL.mem` is designed to cope with network message re-orderings and dynamically changing hosts in sharer lists, which is independent of vCXLGen.

Takeaways. vCXLGen heterogeneous bridges perform comparably to the *MOESI* homogeneous LLC. Every performance difference stems from protocol-specific behaviors rather than suboptimal bridging logic—*MOESI* lacks silent upgrades without an E state, while *CXL* employs more (synchronous) communications with the directory (handshaking) on inter-cluster data movement compared to textbook *MESI*.

9.6 CXL-based Distributed KVS

Methodology. We evaluate the performance of generated CXL bridges for a complex, real-world application: a distributed key-value store (KVS), with the same testbed as in the previous section. We build an in-memory KVS using a concurrent hash-map and a separate allocator, allowing shared memory objects to reside on remote CXL memory, while all other thread-private data remain on local memory.

Results. Fig. 14 shows the throughput for the same protocol combinations evaluated in the previous section. CXL-Br performs nearly identically to MOESI, with a throughput difference of only 1%. In contrast to previous workloads, MESI-Br outperforms both CXL-Br (by 6-8.8%) and the baseline (by 7%).

[2]We conducted a separate analysis on memory address access frequency at the memory controller

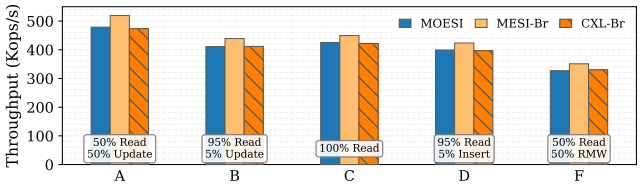


Figure 14. The YCSB benchmark: throughput of compound and baseline protocols, for 8 threads across 2 clusters.

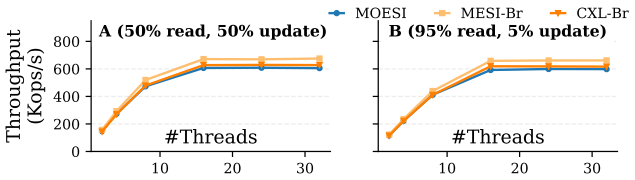


Figure 15. Scalability of compound protocols in YCSB-A (update-heavy) and YCSB-B (read-heavy), for 1 to 32 threads.

Scalability. Fig. 15 shows YCSB-A and YCSB-B throughput as we scale from 1 to 32 threads, adjusting cores, L1s, and LLC banks to eliminate cache hierarchy bottlenecks. All protocols exhibit the same scaling trend, saturating at 16 threads. Throughput differences remain consistent with Fig. 14, with MESI-Br maintaining a peak advantage of 8% over the baseline, and CXL-Br an advantage of 1% over the baseline; both confirming that vCXLGen introduces no scalability overheads compared to the homogeneous baseline.

Takeaway. Our evaluation of a distributed CXL KVS with YCSB confirms that vCXLGen-generated bridges preserve the performance and scalability of real-world applications.

10 Related Work

CXL-based systems. CXL shows promise for memory expansion and shared memory [29, 44, 61, 69]. Prior work focuses on CXL management in memory tiering [14, 50, 52, 59, 73, 78, 80, 87, 90] and its potential in distributed systems and applications [4, 8, 42, 54, 58, 82, 86, 88]. Recent studies explore CXL architectural design [16, 17, 41, 74, 76]. In contrast, to our knowledge, our work uniquely examines CXL in a heterogeneous multi-host coherent setup.

Heterogeneous systems. The industry has proposed various standards to facilitate heterogeneous CC protocols for multicore architectures [12, 28, 31, 37]. Prior work has explored both manual, structured approaches [7, 27, 48, 65] and automated techniques [66–68] for generating and integrating protocols based on their specifications. We specifically build on insights from [51, 67, 68] to devise a protocol synthesis technique suitable for CXL bridges in heterogeneous systems.

Memory consistency models. Prior work has explored the MCM resulting from combining different MCMs [38, 62, 68], also known as *Compound MCM*. They have shown that a properly combined MCM can allow each host to perceive the same MCM as before the combination, requiring no changes to the host's applications. However, they do not address CXL-specific MCM. Recent CXL formalization [81] omits `CXL.mem` and multi-host coherence. Our work combines concrete CC protocols, and ensures similar properties as *compound MCMs*.

Specification-driven automated synthesis. The increasing complexity of modern hardware has driven prior research to focus on automated synthesis techniques. General hardware synthesis has been explored based on atomic specifications [3, 36, 47, 75, 77]. Specialized synthesizers also exist for generating cache coherence protocols [35, 66–68, 83]. However, these methods lack a generalized approach for synthesizing hierarchical protocols for CXL bridges.

Formal verification of CC protocols. Existing coherence verification has limited scalability [67, 68]. Parameterized abstraction [26] struggles with heterogeneous hierarchies. Cluster abstractions [21, 22] lack liveness, heterogeneity, and automated diverse abstraction. CRA [23–25, 89] lacks general automation and scalable heterogeneous model generation. Our compositional method automates abstraction and equivalence verification, showing deadlock freedom and our EL properties.

11 Conclusion

We introduce CXL bridges, a novel abstraction enabling seamless interoperability between heterogeneous hosts with differing CC protocols and MCMs over CXL. We present vCXLGen, the first automated architecture for synthesizing and verifying such bridges. Our evaluation shows that generated bridges preserve performance and that our innovative compositional model checking improves the scalability of liveness verification. We believe vCXLGen is a key step towards realizing CXL's potential in diverse multi-host heterogeneous environments, simplifying the integration of `CXL.mem` coherence protocols with existing and future host architectures.

Acknowledgments

We thank the anonymous reviewers and members of the Systems Research Group at the TU Munich for their feedback. 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), European Union(EU) HORIZON-JU-IA (No. 101140087), 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

A Artifact Appendix

A.1 Abstract

This artifact contains the codebase of vCXLGen, our tool for synthesizing and verifying CXL bridges, as well as a gem5-based implementation of vCXLGen-generated bridges. It also comprises the machine-readable specifications of several CC protocols, including `CXL.mem`. The artifact also includes scripts and instructions for reproducing the evaluation results presented in this work, with detailed steps to use the tool and models, installing dependencies, and compiling the benchmarks from scratch.

To facilitate the use of the artifact, we use `nix` for the verification experiments, and also provide a Docker container with its `Dockerfile` to set up the environment with all required dependencies for the gem5-based experiments.

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

A.2 Artifact check-list (meta-information)

- **Compilation:** GCC/G++ 11.4.0, SCons 4.0+, Python 3.10+, and all gem5 v23.1.0.0 dependencies
- **Data set:** PARSEC 3.0, SPLASH-4, Phoenix-2.0, YCSB; Litmus tests: *IRIW, MP, 2+2W, CoRR1, CoRR2, LB, R, RWC, S, SB, WRC, WRW+2W, and WWC* (generated with Herd7 [5])
- **Run-time environment:** Ubuntu 22.04 LTS and the `nix` package manager (native or Docker container)
- **Metrics:** Execution time, throughput, memory usage.
- **Experiments:** Use the provided scripts to generate CXL bridges and verify safety and liveness properties, and evaluate the performance (execution time, throughput) of different bridges and cache coherence protocol combinations.
- **How much disk space is required? (approx):** Total: ~20 GiB. Breakdown: generated litmus test Murφ models: ~5 GiB, gem5 builds: ~8 GiB, PARSEC: ~2 GiB, other benchmarks: <1 GiB, experiment outputs: <1 GiB.
- **How much time is needed to prepare workflow? (approx):** ~3 minutes to set up the generator and verification tools, ~1–3 hours to compile all gem5 variants; ~30 minutes to compile the workloads.
- **How much time is needed to complete experiments? (approx):** ~8 hours for all litmus test models, ~24 hours for liveness models, ~4–12 hours for the full gem5 experiment suite (Figure 11, 12, 13, 14) on a 128-core server.
- **Publicly available:** Yes
- **Code licenses:** MIT
- **Archived (DOI):** https://doi.org/10.5281/zenodo.17939343

A.3 Description

A.3.1 How to access. The source code is publicly available on GitHub (https://github.com/TUM-DSE/vCXLGen) or Zenodo (https://doi.org/10.5281/zenodo.17939343).

A.3.2 Hardware dependencies. The artifact can be evaluated on any general-purpose server with at least 20 GiB free disk space to build the litmus test models and gem5. Our compositional liveness models require a minimum of 60 GiB available DRAM. 2 TiB DRAM is required to show that the baseline liveness models fail even with a large memory size. Less than 2 GiB DRAM is needed for each gem5 simulation. We recommend running the experiments on a server with at least 32 cores to speed up compilation and parallelize simulations.

A.3.3 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, verification only requires the `nix` package manager, and a Ubuntu 22.04 installation (native, container, or VM) can be used for gem5 simulations. To run the artifacts natively, refer to the artifact repository for additional instructions to set up the environment.

A.3.4 Data sets. The artifact includes the source code and instructions to build the workloads from source. We also provide pre-compiled simulation binaries as a Docker image on DockerHub (https://hub.docker.com/r/gingerbreadz/vcxlgen-artifact-prebuilt/). To build from source, use the script indicated in the instruction `README`.

A.4 Installation

The artifact repository contains all necessary components, including the vCXLGen generator tool, the Murφ and *Rumur* model checkers, the gem5 simulator, the PARSEC, SPLASH, PHOENIX, and YCSB benchmark suites, and experiment scripts. For each component, we provide detailed build instructions in its respective `README` section.

To set up the environment for the vCXLGen tool and the model checkers, please follow the instructions in the `Set up` section of the artifact `README`.

To generate and run the safety and liveness verification models, please follow the instructions in the `Verification` section of the artifact `README`.

To build *gem5* with all protocol variants (MOESI_CMP, MESI-MESI-MESI, MESI-CXL-MESI), 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 first need to run the workload configurations script: (`Generate Workload Conf.`) and then run the experiments. Detailed instructions are available in `Run Experiments`.

A.5 Evaluation and expected results

Once all experiments have been completed, the run script will create plots for Fig. 11-15 and place them into the `OUTPUT` folder. The figures should match the ones in the paper.

A.6 Methodology

Submission, reviewing, and badging methodology:

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

References

[1] 2008. The Princeton application repository for sharedmemory computers (PARSEC). Retrieved Apr. 2025 from http://parsec.cs.princeton.edu/.

[2] 2024. RISC-V. Retrieved Apr. 2025 from https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications

[3] 2025. Bluespec. Retrieved Apr. 2025 from https://bluespec.com/

[4] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *Proceedings of the 18th International Workshop on Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN '22)*. Association for Computing Machinery, New York, NY, USA, Article 8, 5 pages. doi:10.1145/3533737.3535090

[5] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. doi:10.1145/2627752

[6] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. 2016. Lazy release consistency for GPUs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 26, 13 pages.

[7] Johnathan Alsop, Matthew D. Sinclair, and Sarita V. Adve. 2018. Spandex: a flexible interface for efficient heterogeneous coherence. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 261–274. doi:10.1109/ISCA.2018.00031

[8] Chloe Alverti, Stratos Psomadakis, Burak Ocalan, Shashwat Jaiswal, Tianyin Xu, and Josep Torrellas. 2025. 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 210–226. doi:10.1145/3676641.3715988

[9] Amazon. 2025. Amazon EC2 Instance types. Retrieved Apr. 2025 from https://aws.amazon.com/ec2/instance-types/

[10] AMD. 2012. AMD64 Architecture Programmer's Manual Volume 2: System Programming. Retrieved Dec. 2024 from https://web.archive.org/web/20170619232736/http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf#page=217

[11] ARM. 2011. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition.* Technical Report. ARM Limited. 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

[12] ARM. 2024. AMBA CHI Architecture Specification. https://developer.arm.com/Architectures/AMBA

[13] ARM. 2024. MESI and MOESI protocols. Retrieved Dec. 2024 from https://developer.arm.com/documentation/den0013/d/Multi-core-processors/Cache-coherency/MESI-and-MOESI-protocols

[14] Maurice Bailleu, Dimitrios Stavrakakis, Rodrigo Rocha, Soham Chakraborty, Deepak Garg, and Pramod Bhatotia. 2024. Toast: A Heterogeneous Memory Management System. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Long Beach, CA, USA). Association for Computing Machinery, New York, NY, USA, 53–65. doi:10.1145/3656019.3676944

[15] Nick Barrow-Williams, Christian Fensch, and Simon Moore. 2009. A communication characterisation of Splash-2 and Parsec. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, USA, 86–97. doi:10.1109/IISWC.2009.5306792

[16] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. 2023. Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms. *IEEE Micro* 43, 2 (2023), 30–38. doi:10.1109/MM.2023.3241586

[17] Daniel S. Berger, Yuhong Zhong, Pantea Zardoshti, Shuwei Teng, Fiodar Kazhamiaka, and Rodrigo Fonseca. 2025. Octopus: Scalable Low-Cost CXL Memory Pooling. arXiv:2501.09020 [cs.AR] https://arxiv.org/abs/2501.09020

[18] Srikant Bharadwaj, Jieming Yin, Bradford Beckmann, and Tushar Krishna. 2020. Kite: A Family of Heterogeneous Interposer Topologies Enabled via Accurate Interconnect Modeling. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. doi:10.1109/DAC18072.2020.9218539

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

[20] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. doi:10.1145/2024716.2024718

[21] Xiaofang Chen and Ganesh Gopalakrishnan. 2006. *A general compositional approach to verifying hierarchical cache coherence protocols.* Technical Report. Technical Report. UUCS-06-014, School of Computing, University of Utah.

[22] Xiaofang Chen, Yu Yang, M. Delisi, G. Gopalakrishnan, and Ching-Tsun Chou. 2007. Hierarchical cache coherence protocol verification one level at a time through assume guarantee. In *2007 IEEE International High Level Design Validation and Test Workshop*. 107–114. doi:10.1109/HLDVT.2007.4392796

[23] Shing Chi Cheung, Dimitra Giannakopoulou, and Jeff Kramer. 1997. Verification of liveness properties using compositional reachability analysis. *ACM SIGSOFT Software Engineering Notes* 22, 6 (1997), 227–243.

[24] Shing-Chi Cheung, Dimitra Giannakopoulouy, and Je Kramery. 1996. *Incorporating Verification of Liveness Properties in Compositional Reachability Analysis.* Technical Report. Citeseer.

[25] Shing Chi Cheung and Jeff Kramer. 1999. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 8, 1 (1999), 49–78.

[26] Ching-Tsun Chou, Phanindra K Mannava, and Seungjoon Park. 2004. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings 5*. Springer, 382–398.

[27] Rachel Cleaveland and Caroline Trippel. 2024. Memory Consistency Model-Aware Cache Coherence for Heterogeneous Hardware. In *2024 Formal Methods in Computer-Aided Design (FMCAD)*. 1–12. doi:10.34727/2024/isbn.978-3-85448-065-5_22

[28] CCIX Consortium. 2025. CCIX Standard. Retrieved Apr. 2025 from https://www.ccixconsortium.com/

[29] CXL Consortium. 2025. Integrators List. Retrieved May 1st, 2025 from https://computeexpresslink.org/integrators-list/

[30] Compute Express Link Consortium. 2024. Compute Express Link (CXL) Consortium. Retrieved Dec. 2024 from https://www.computeexpresslink.org/

[31] Compute Express Link Consortium. 2024. *Compute Express Link (CXL) Specification 3.2.* Technical Report. Compute Express Link Consortium.

[32] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152

[33] AL Cox, Sandhya Dwarkadas, and P Keleher. 1994. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems*.

[34] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.* 56, 11, Article 290 (July 2024), 37 pages. doi:10.1145/3669900

[35] Nirav Dave, Man Cheuk Ng, et al. 2005. Automatic synthesis of cache-coherence protocol processors using bluespec. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE'05.* IEEE, 25–34.

[36] Nirav Dave, Michael Pellauer, et al. 2007. Scheduling as rule composition. In *2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*. IEEE, 51–60.

[37] HSA Foundation. 2024. Heterogeneous System Architecture (HSA) Foundation. Retrieved Dec. 2024 from https://hsafoundation.com/

[38] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound Memory Models. *Proc. ACM Program. Lang.* 7, PLDI, Article 153 (June 2023), 24 pages. doi:10.1145/3591267

[39] Eduardo José Gómez-Hernández, Juan Manuel Cebrian, Stefanos Kaxiras, and Alberto Ros. 2022. Splash-4: A Modern Benchmark Suite with Lock-Free Constructs. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, Austin, TX (USA), 51–64. doi:10.1109/IISWC55918.2022.00015

[40] Google. 2025. Google Cloud: Machine families resource and comparison guide. Retrieved Apr. 2025 from https://cloud.google.com/compute/docs/machine-resource

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

[42] Yufeng Gu, Alireza Khadem, Sumanth Umesh, Ning Liang, Xavier Servot, Onur Mutlu, Ravi Iyer, and Reetuparna Das. 2025. 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 862–881. doi:10.1145/3676641.3716267

[43] Reece Hayden and Paul Schell. 2024. Opportunities and challenges for Compute Express Link (CXL). Retrieved Apr. 2025 from https://computeexpresslink.org/wp-content/uploads/2024/11/CR-CXL-101_FINAL.pdf

[44] Gary Hilson. 2025. Micron Exits 3D XPoint Market, Eyes CXL Opportunities. Retrieved Apr. 2025 from https://www.eetimes.com/micron-exits-3d-xpoint-market-eyes-cxl-opportunities/

[45] Intel. 2024. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1.* Technical Report. Intel Corporation.

[46] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. 2024. Memory Sharing with CXL: Hardware and Software Design Approaches. arXiv:2404.03245 [cs.ET] https://arxiv.org/abs/2404.03245

[47] Michal Karczmarek et al. 2008. Synthesis from multi-cycle atomic actions as a solution to the timing closure problem. In *2008 IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 24–31.

[48] Edya Ladan-Mozes and Charles E. Leiserson. 2008. A consistency architecture for hierarchical shared caches. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures* (Munich, Germany) *(SPAA '08)*. Association for Computing Machinery, New York, NY, USA, 11–22. doi:10.1145/1378533.1378536

[49] Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691. doi:10.1109/TC.1979.1675439

[50] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 17–34. doi:10.1145/3600006.3613167

[51] Anatole Lefort, David Schall, Nicolò Carpentieri, Julian Pritzi, Soham Chakraborty, Nicolai Oswald, and Pramod Bhatotia. 2026. C^3: CXL Coherence Controllers for Heterogeneous Architectures. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[52] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835

[53] Kai Li. 1988. IVY: A Shared Virtual Memory System for Parallel Computing. *ICPP (2)* 88 (1988), 94.

[54] Shaobo Li, Yirui (Eric) Zhou, Hao Ren, and Jian Huang. 2025. 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 116–132. doi:10.1145/3669940.3707250

[55] Linux. 2025. Direct Access for files. Retrieved Apr. 2025 from https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[56] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs.AR] https://arxiv.org/abs/2007.03152

[57] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. 2015. ArMOR: defending against memory consistency model mismatches in heterogeneous architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 388–400. doi:10.1145/2749469.2750378

[58] Teng Ma, Zheng Liu, Chengkun Wei, Jialiang Huang, Youwei Zhuo, Haoyu Li, Ning Zhang, Yijin Guan, Dimin Niu, Mingxing Zhang, and Tao Ma. 2024. HydraRPC: RPC in the CXL Era. In *Proceedings of the*

2024 USENIX Conference on Usenix Annual Technical Conference (Santa Clara, CA, USA) *(USENIX ATC'24)*. USENIX Association, USA, Article 24, 9 pages.

[59] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. doi:10.1145/3582016.3582063

[60] Microsoft. 2025. Azure Virtual Machine series. Retrieved Apr. 2025 from https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/

[61] Timothy Prickett Morgan. 2025. The CXL Roadmap Opens Up The Memory Hierarchy. Retrieved Apr. 2025 from https://www.nextplatform.com/2021/09/07/the-cxl-roadmap-opens-up-the-memory-hierarchy/

[62] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence, Second Edition.* Morgan & Claypool Publishers. doi:10.2200/S00962ED2V01Y201910CAC049

[63] Bill Nitzberg and Virginia Lo. 1991. Distributed shared memory: A survey of issues and algorithms. *Computer* 24, 8 (1991), 52–60.

[64] NVIDIA. [n. d.]. NVIDIA Unveils NVLink Fusion for Industry to Build Semi-Custom AI Infrastructure With NVIDIA Partner Ecosystem. https://nvidianews.nvidia.com/news/nvidia-nvlink-fusion-semi-custom-ai-infrastructure-partner-ecosystem

[65] Lena E. Olson, Mark D. Hill, and David A. Wood. 2017. Crossing Guard: Mediating Host-Accelerator Coherence Interactions. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 163–176. doi:10.1145/3037697.3037715

[66] Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. 2018. ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 247–260. doi:10.1109/ISCA.2018.00030

[67] Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. 2020. HieraGen: Automated Generation of Concurrent, Hierarchical Cache Coherence Protocols. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 888–899. doi:10.1109/ISCA45697.2020.00077

[68] Nicolai Oswald, Vijay Nagarajan, Daniel J. Sorin, Vasilis Gavrielatos, Theo Olausson, and Reece Carr. 2022. HeteroGen: Automatic Synthesis of Heterogeneous Cache Coherence Protocols. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 756–771. doi:10.1109/HPCA53966.2022.00061

[69] Mark Papermaster. 2025. AMD Joins Consortia to Advance CXL, a New High-Speed Interconnect for Breakthrough Performance. Retrieved Apr. 2025 from https://community.amd.com/t5/business/amd-joins-consortia-to-advance-cxl-a-new-high-speed-interconnect/ba-p/418202

[70] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2017), 29 pages. doi:10.1145/3158107

[71] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. 12 pages. doi:10.1109/HPCA.2007.346181

[72] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. doi:10.1145/1785414.1785443

[73] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2023. vTMM: Tiered Memory Management for Virtual Machines. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 283–297. doi:10.1145/3552326.3587449

[74] Debendra Das Sharma. 2023. Compute Express Link (CXL): Enabling Heterogeneous Data-Centric Computing With Heterogeneous Memory Hierarchy. *IEEE Micro* 43, 2 (2023), 99–109. doi:10.1109/MM.2022.3228561

[75] Xiaowei Shen et al. 1999. Using term rewriting systems to design and verify processors. *IEEE Micro* 19, 3 (1999), 36–46.

[76] Joonseop Sim, Soohong Ahn, Taeyoung Ahn, Seungyong Lee, Myunghyun Rhee, Jooyoung Kim, Kwangsik Shin, Donguk Moon, Euiseok Kim, and Kyoung Park. 2023. Computational CXL-Memory Solution for Accelerating Memory-Intensive Applications. *IEEE Computer Architecture Letters* 22, 1 (2023), 5–8. doi:10.1109/LCA.2022.3226482

[77] Jørgen Staunstrup and Mark R. Greenstreet. 1988. From high-level descriptions to VLSI circuits. *BIT Numerical Mathematics* 28 (1988), 620–638.

[78] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2025. 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 604–621. doi:10.1145/3676641.3711999

[79] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 105–121. doi:10.1145/3613424.3614256

[80] Bijan Tabatabai, James Sorenson, and Michael M. Swift. 2024. FBMM: making memory management extensible with filesystems. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC'24)*. USENIX Association, USA, Article 48, 14 pages.

[81] Chengsong Tan, Alastair F. Donaldson, and John Wickerson. 2025. Formalising CXL Cache Coherence. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 437–450. doi:10.1145/3676641.3715999

[82] Yupeng Tang, Seung-seob Lee, Abhishek Bhattacharjee, and Anurag Khandelwal. 2025. 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 858–875. doi:10.1145/3669940.3707253

[83] Dana Vantrease, Mikko H Lipasti, and Nathan Binkert. 2011. Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 132–143.

[84] Yanjing Wang, Lizhou Wu, Sunfeng Gao, Yibo Tang, Junhui Luo, Zicong Wang, Yang Ou, Dezun Dong, Nong Xiao, and Mingche Lai. 2026. Cohet: A CXL-Driven Coherent Heterogeneous Computing Framework with Hardware-Calibrated Full-System Simulation. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[85] Yanjing Wang, Lizhou Wu, Wentao Hong, Yang Ou, Zicong Wang, Sunfeng Gao, Jie Zhang, Sheng Ma, Dezun Dong, Xingyun Qi, Mingche Lai, and Nong Xiao. 2025. CXL-DMSim: A Full-System CXL Disaggregated Memory Simulator With Comprehensive Silicon Validation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025), 1–1. doi:10.1109/tcad.2025.3607145

[86] Zhao Wang, Yiqi Chen, Cong Li, Yijin Guan, Dimin Niu, Tianchan Guan, Zhaoyang Du, Xingda Wei, and Guangyu Sun. 2025. 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 192–209. doi:10.1145/3676641.3716244

[87] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. NOMAD: non-exclusive memory tiering via transactional page migration. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 2, 17 pages.

[88] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 658–674. doi:10.1145/3600006.3613135

[89] Hao Zheng. 2010. Compositional Reachability Analysis for Efficient Modular Verification of Asynchronous Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29, 3 (2010), 329–340. doi:10.1109/TCAD.2009.2035544

[90] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing memory tiers with CXL in virtualized environments. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 3, 20 pages.