CXLMC: Model Checking CXL Shared Memory Programs

Yutong Guo
University of California, Irvine
Irvine, CA, USA
yutong4@uci.edu

Conan Truong
University of California, Irvine
Irvine, CA, USA
cjtruong@uci.edu

Brian Demsky
University of California, Irvine
Irvine, CA, USA
bdemsky@uci.edu

Abstract

Compute Express Link (CXL) shared memory is an emerging industry standard that will allow for cache coherent sharing of remote memory between many machines. Memory devices will contain large amounts of DRAM that can be shared by many machines in a CXL cluster. This will enable software running on clusters of computers to use shared memory to communicate more efficiently and to share important data between these machines.

As CXL clusters grow larger, machine failures will become a significant risk. Software will need to tolerate machine failures. A key challenge is that CXL uses caching of remote memory to hide latency. If a machine fails before it has flushed dirty cache lines back to the CXL shared memory device, the latest stores to those cache lines will be lost. Data structures have been developed that combine crash-consistent designs with flush and fence instructions to ensure that the data structures remain consistent even in the presence of failures.

However, developing such crash-consistent data structures is error prone. It is easy to make a design or implementation error. Such crash consistency errors are hard to detect with testing. We propose CXLMC, a model checker that systematically explores crashing executions for the x86-CXL shared memory platform. We have evaluated CXLMC and found 24 bugs in 8 applications including 7 new bugs.

1 Introduction

Compute Express Link (CXL) is an open standard for cache-coherent interconnects [1, 22]. CXL enables multiple machines to access memory shared by a CXL network-attached memory device in a cache-coherent fashion. CXL has the potential to enable new software that leverages multiple machines to efficiently compute on shared data. It also enables multiple machines to efficiently share one copy of a large dataset. CXL has started to attract the interest of the systems research community and several researchers have proposed using clusters of CXL connected machines, or CXL Pods [87], to implement more efficient database systems [36, 39, 40, 51] and key value stores [67].

CXL 3.2 is designed to support cache coherent sharing between up to 4,095 memory entities. With larger clusters, machine failures will inevitably occur. Clusters will contain both memory nodes and compute nodes. We expect that there will be fewer memory nodes and thus they will have a lower risk of failure. Compute nodes, on the other hand, will be more common, and thus there will be a higher risk that one fails.

Software applications will need to tolerate the failure of one or more compute nodes. The first challenge is that a compute node failure stops the execution of the program on that node, and software must tolerate this. However, caches bring additional challenges for tolerating failures. If a failed compute node has written to a cache line, it is possible that the only copy of those stores resided in the cache of the failed compute node. While CXL has a mechanism, global persistent flush (GPF), for flushing the machine's cache, GPF is primarily targeted towards power outages. GPF does not address a wide range of other potential failures such as a power supply, motherboard, or CXL transceiver failure.

Persistent memory (PM) systems have similar challenges with failures. However, there are significant differences between failures in CXL shared memory and PM. First, we expect partial failures in which one compute node fails to be the primary concern for CXL shared memory systems, whereas PM systems have a complete failure model. Second, CXL systems know which cache lines were lost due to the cache coherence directory structure. Recovery procedures can potentially leverage this information. The partial failure mode can complicate recovery. Operations on persistent memory structures can assume that any failures have occurred before the operation starts. For CXL memory, failures can occur concurrently with running data structure operations and those operations may need recovery code to handle concurrent failures. For example, a machine may fail while another machine is running recovery code for a previous failure, a scenario that is not possible for PM systems.

Developing crash-consistent code can be difficult. A key lesson from earlier work on building crash-consistent persistent memory applications is that developing correct crash-consistent code is very difficult. Early work on crash-consistent persistent memory data structures was largely done without tool support. Later, researchers developed several tools [2, 24, 27, 31, 33, 47, 54, 55, 61] for finding crash-consistency bugs in persistent memory programs. Developing crash-consistent software without tool support has proven to be bug prone. Between these tools, they have found

183 crash-consistency bugs in a relatively small number of benchmark applications.

The CXL shared memory standard is new, and the hardware is not commercially available and thus has not yet been used in real systems. Existing information suggests that the amount of shared memory CXL systems support may be limited [39–41]. Early work suggests that CXL shared memory may find use in data structures used for coordination between machines and core index structures. There are ongoing efforts from both industry and academia to implement key shared data structures using CXL shared memory, including database indices [36, 39, 40, 51], message buffers for cross-node communication [8, 56, 57], and shared object stores [63, 72]. Note that a single machine crash could corrupt these core data structures, stopping an entire cluster of machines.

In this work, we present a model checker that enables efficiently checking for crash consistency of x86-CXL shared memory programs using a constraint-based approach; the constraint-based approach was originally developed by prior work on model checking persistent memory programs [33]. Differences between the CXL and PM failure and consistency models mean that this approach is not directly applicable. We have extended the constraint-based approach to handle the CXL memory failure and consistency models. In particular, our model checker is able to emulate x86-CXL shared memory programs running across multiple compute nodes, as well as the partial failure of any subset of compute nodes. Supporting these requirements efficiently requires innovations in both algorithm design and implementation.

We introduce CXLMC, the first model checker for crash-consistency bugs for CXL shared memory. While CXL is intended to be cross-platform, our work focuses on CXL shared memory using x86 machines. This paper makes the following contributions:

- **Adapting constraint-based model checking to CXL shared memory:** Constraint-based model checking approaches must be modified because CXL shared memory has a partial failure model, in which individual machines can fail during the execution while others continue. CXL shared memory also has a different consistency model—remote reads ensure that cache lines have been written to the backing storage.
- **Multi-process support:** Model checking CXL shared memory requires novel implementation techniques as CXL shared memory programs will have multiple processes that all have access to a common shared memory region. Prior model checkers do not support this multi-process model.
- **Implementation:** We develop CXLMC to implement our model checking algorithm. CXLMC uses the LLVM compiler to instrument C and C++ CXL programs.
- **Evaluation:** We have evaluated CXLMC on a set of benchmarks. These benchmarks consist of the RECIPE [49] persistent memory benchmarks that have been adapted for use with CXL shared memory as well as programs from CXL-SHM [85].

2 Background

In this section, we provide an overview of the relevant CXL protocols and the TSO memory model.

2.1 Compute Express Link

Compute Express Link (CXL) features a suite of interconnect protocols for low-latency remote memory accesses between host and device machines. The CXL.mem protocol [66] gives a host the ability to access the attached device memory at cache line granularity with cache coherence guarantees. Additionally, in the CXL 3.0 standard, multiple hosts share a single CXL memory pool by mapping it into their address space and accessing the memory concurrently and coherently, enabling applications to use CXL memory as distributed shared memory [36, 85]. Although there is currently no implementation of CXL 3.0 in hardware, emulation is possible through other means, such as having multiple NUMA nodes share a CXL-enabled device [85].

2.2 Memory Model

Our tool implements the x86-TSO memory model with the addition of cache line flush instructions. We based the properties of the memory instructions on prior formalizations of the x86 persistent memory semantics [18, 43, 64], and in particular, the $Px86_{sim}$ model [64]. Currently, there is no definite standard on the CXL memory model beyond cache coherence, and we chose this design based on discussions with Intel engineers who confirmed that hardware cache-coherent CXL memory is intended to support the TSO model. Although the CXL standard is intended to be cross platform, the exact memory semantics will depend on the underlying machines. We chose to focus our efforts on x86 as many of the available benchmarks [49, 85] assume the x86 architecture. We note that although CXL enables machines with different memory models to coherently share memory, we do not consider such scenarios, as the memory model for such cases remains unclear. In contrast, CXL0 [10], an abstract model for the CXL protocol layer, includes operations like a remote store that currently have no x86 mapping, and thus is not supported by CXLMC. If subtle differences are discovered between the eventual implementations and the expected failure semantics and memory model, we expect that our approach is general enough to support these variations with small changes. Figure 1 shows a representation of the target memory model of our tool, namely multiple machines with x86-TSO memory model sharing CXL memory.

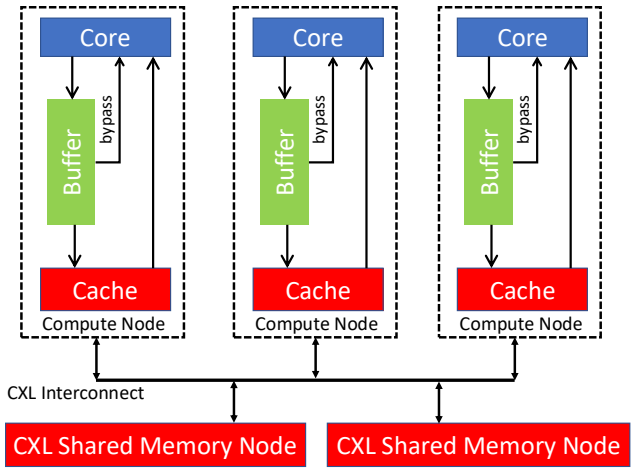

Figure 1. Three Machines with x86-TSO Memory Model Sharing Two CXL Memory Devices

Earlier in Program Order \ Later in Program Order	Re	Wr	RMW	mf	sf	clflushopt	clflush
Read	✓	✓	✓	✓	✓	✓	✓
Write	X	✓	✓	✓	✓	CL	✓
RMW	✓	✓	✓	✓	✓	✓	✓
mfence	✓	✓	✓	✓	✓	✓	✓
sfence	X	✓	✓	✓	✓	✓	✓
clflushopt	X	X	✓	✓	✓	X	CL
clflush	X	✓	✓	✓	✓	CL	✓

Table 1. Summary of ordering constraints in the $Px86_{sim}$ model from Raad et al. [64] A ✓ indicates that the order between the two instructions is preserved, a X indicates that the two instructions can be reordered, and a CL indicates that the order is preserved only if they both operate on the same cache line.

In the x86 total store order (TSO) memory model, each thread has a store buffer that buffers its stores to the cache to hide the memory access latency. Loads from a thread can read the most recent store in its local store buffer through bypassing, as well as stores that have been committed to the cache, but they cannot read from stores in other thread's store buffers. In short, stores may be visible to the local thread before other threads due to store buffering. Buffered stores are committed to the cache in program order, but at a nondeterministic rate. Fence instructions such as `mfence` and `sfence` as well as `RMW` instructions have the effect of emptying the store buffer immediately, causing all local stores to become visible globally.

The x86 architecture provides three flush instructions — `clflush`, `clflushopt`, and `clwb`. The `clflush` instruction is the most expensive and has stronger ordering constraints. It is ordered with respect to later writes and `clflushopt`/`clwb` instructions. The other two instructions `clflushopt` and `clwb` have the same weaker ordering constraints and are often combined with fence instructions to flush multiple cache lines in a batch. Their only difference is that `clwb` is not required to invalidate the cache line written back to the memory, leading to better performance in some cases. As far as our system is concerned, they can be treated identically. A summary of the ordering constraints between the relevant instructions is shown in Table 1.

Shared cache-coherent CXL memory differs from persistent memory in a few key ways. One key difference is that cache coherence in shared CXL memory implies that when one host loads a cache line owned by another host, the cache line will be written back to the CXL memory device. Another key difference is that the directory structures on a CXL memory node mean that in the event of a compute node failure, CXL knows which cache lines were lost. This enables the CXL protocol to define a memory poisoning mechanism that returns a poison value when a host reads from a cache line owned by another failed host [37]. Our tool supports memory poisoning as an option, but we do not use it for our evaluation as it is a drastically different failure model than the familiar one from PM, and currently there are no applications designed to work with memory poisoning enabled.

Like for persistent memory, stores to CXL memory are volatile when in the cache, and may be lost if the writing machine fails in a way not handled by global persistent flush. The stores must be written back to memory to remain visible to other processors after the writing processor crashes. A crucial difference from persistent memory is that the failure of individual machines running a CXL memory program only causes data loss in their own caches, while the caches of other machines are unaffected. Write backs from the cache to memory happen when the cache needs more space for later stores and do not follow any specific order. Therefore, explicit flush instructions are often needed to ensure crash consistency.

Global Persistent Flush: In the case of a host failure, CXL provides a global persistent flush (GPF) [66] mechanism that flushes the contents of a host's cache to shared memory. GPF primarily addresses data losses due to power outages; GPF requires a power store such as a battery or power supply capacitor to keep a system powered long enough to flush its caches. GPF relies on many components of a machine to function correctly. If a machine fails due to a component GPF requires for its function such as a broken motherboard, failed CPU, power supplies, CXL transceiver, CXL cable, etc., GPF may not successfully flush the caches. It is also not clear how universal support for GPF will be in CXL systems due to the energy store requirement. This work focuses specifically on failure scenarios that cannot be handled by GPF, and therefore we assume that failed machines do not write their cache back.

3 System Overview

CXLMC is designed to find crash consistency bugs in CXL shared memory programs written by exhaustively exploring different partial failure scenarios. Programs are first instrumented by an LLVM [26] pass that replaces all memory accesses, fences, and flushes with calls into CXLMC. CXLMC then launches the instrumented programs using separate processes to simulate different machines. Each process has an independent address spaces and emulates CXL shared memory. Each CXLMC process can have multiple threads, simulating multiple threads running on each CXL machine. CXLMC also emulates other aspects of the memory model such as store buffers and machine local caching, and intercepts methods from the pthread library to have control over multithreading.

3.1 System Configuration

CXLMC assumes a CXL system configuration consisting of multiple x86 compute nodes sharing a single memory device. It is conceptually straightforward to extend this model to allow sharing multiple memory devices and/or to allow the compute nodes to share their local memory. The key difference with a system in which a compute node shares its local memory is that a failure of that compute node would result in the complete loss of access to that memory region. Although our core algorithms could easily support such systems, we did not explore this scenario because the benchmark programs we could find could not tolerate the complete loss of a region of memory. We do not currently consider the case of any memory device being persistent as this may require new x86 instructions to flush a remote cache.

3.2 Correctness, Soundness, and Completeness

CXLMC exposes critical bugs that lead to unintended crashes or assertion failures, and provides the soundness guarantee that any execution it finds is feasible under the memory and failure model. This implies that all bugs it finds are actual bugs. Note that some bugs might not be realizable for specific processor models, due to factors such as eviction policies that are not guaranteed by the architectural model and thus not modeled by CXLMC. These are still bugs that should be fixed as processor upgrades or other small system changes may make them realizable. CXLMC also does not check a specific correctness condition beyond whether the program crashes or an existing assertion fails, though CXLMC is designed to support extensions to check properties such as robustness [31]. In terms of completeness, CXLMC is guaranteed to enumerate at least one execution from every equivalence class under reads-from equivalence [15] for a fixed schedule. CXLMC only model checks non-determinism from crashes and does not model check non-determinism from concurrency in order to scale to longer executions. CXLMC can generate multiple different schedules using different random seeds. Performance characteristics are not considered, as it is only intended as a tool for checking correctness.

Naively enumerating all post-crash states will result in an exponential blow-up of the state space. This approach was taken by the Yat [47] tool for model checking persistent memory crashes. For CXL programs with partial failures, the eager enumeration of post-failure executions would be even more intractable than for persistent memory programs, as at every step of the execution all subsets of currently running processes are subject to failures. Jaaru [33] developed a solution to this problem for PM programs using the technique of cache line constraint refinement that constructs post-failure executions lazily based on constraints on the possible time interval of pre-failure cache line write backs. Alternatively, it could be understood as a form of dynamic partial order reduction (DPOR) [25] that only explores executions with differences observable through post-failure loads. It efficiently verified programs that use the commit store pattern, where a commit store makes a group of prior stores observable. This pattern is valid for CXL programs, and cache line refinement ensures only two executions are explored for each commit store pattern—one with a failure before the commit store is written back, and one that does not fail. We adapt this technique in CXLMC, which requires a novel formulation to address the challenges of partial failures.

3.3 Constraint Refinement

The post-failure state of a cache line is determined by the time it was last written back to DRAM before the failure—all stores to the cache line ordered before the last write back are persisted, and all stores to the cache line ordered after are lost. Explicit flush instructions such as `clflush`, and for CXL shared memory, remote loads from the same cache line establish a lower bound on the time of the last write back. However, this information is not sufficient. Suppose that when a machine A crashes each of its m cache lines has n potentially unpersisted stores, *i.e.*, stores that come after the lower bound for its last flush computed based on pre-crash information, the possible number of post-crash states of its cache lines would be $O((n+1)^m)$, since the last flush for each cache line could be at any point within the sequences of n stores. Recall that a given CXL machine may have multiple threads; CXLMC implements this by running multiple threads within each process. Note that a local load from another thread on the same machine that performed a store does not force the store to be written back and thus does not refine the cache line constraint.

The classic DPOR formulation pioneered the approach of identifying backtracking points as the execution progresses [25]. The same idea is applicable for enumerating the possible time of cache line write backs, which gives rise to cache line constraint refinement. In cache line constraint refinement, we keep track of a possible time interval for the last flush of each cache line, which we call cache line

constraints, and refine it throughout the execution. The set of time intervals encodes an equivalence class of states if a crash were to occur, rather than a single state. A post-crash load from the cache line is then able to lazily determine its value based on the cache line constraints.

To illustrate, consider a program with two processes A and B that share two CXL memory locations `x` and `y` on the same cache line. As shown in Figure 2, Machine A stores `y=1`, `x=2`, executes a `clflush`, and then stores `y=3`, `x=4`, `y=5`, and `x=6` before crashing due to a machine failure. Suppose that the store `y=1` is at timestamp 1, then we can infer that the cache line constraint is $[3, \infty)$ from the position of the only `clflush` instruction on the timeline.

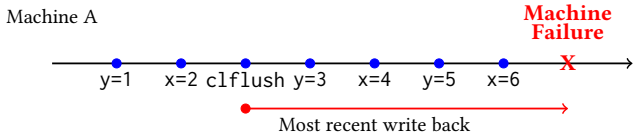

Figure 2. Execution of a CXL shared memory program on machine A. Assume `x` and `y` share the same cache line and do not overlap.

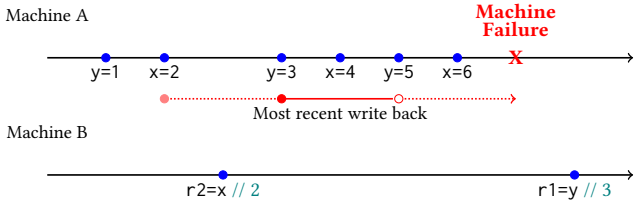

Figure 3. Execution of a CXL shared memory program on machines A and B. Assume `x` and `y` share the same cache line and do not overlap.

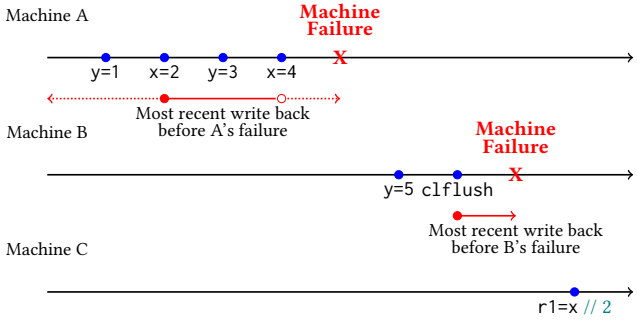

Figure 4. Execution of a CXL shared memory program on machines A, B, and C. Assume `x` and `y` share the same cache line.

When multiple machines communicate through CXL memory, loads from a machine can cause stores to be persisted. In Figure 3, Machine B loads the value 2 from `x` that was written by machine A before it failed. This writes back the cache line due to CXL's cache coherence and updates the constraint to $[2, \infty)$ as shown by the dotted red interval. When machine B loads from `y` following machine A's failure, the loaded value could be any value that `y` holds during the current cache line constraint interval—in this case, `y=1`, `y=3` or `y=5`, and we can further refine the cache line constraint based on the load result. Suppose that the result is 3 as shown in the figure, then the most recent cache line write back must have occurred after `y=3` but before `y=5`, and the constraint is updated accordingly to $[4, 7)$, shown as the solid red interval. This constraint further limits the results of future loads from the cache line. If machine B were to load from `y` again at this point, the only possible value would be `y=3`, which ensures the consistency of consecutive loads from the same location. If it were to load from `x`, the possible values are `x=2` and `x=4`.

The same procedure extends to the case of multiple partial failures, with extra attention needed to handle the constraints for each failed machine separately. For example, consider a program with machines A, B, and C sharing memory locations `x` and `y` as shown in Figure 4. Machine A executes the stores `y=1`, `x=2`, `y=3`, and `x=4` before failing, and then machine B executes `y=5` and `clflush` before failing. At this point, the cache line constraint for A is the default $[0, \infty)$, and the constraint for B is $[7, \infty)$ as machine B executes a `clflush` at timestamp 7. When machine C reads from `x` after both A and B have failed, it consults the cache line constraint for A as `x`'s most recent stores `x=2` and `x=4` are from A, and whether they are persisted depends on the cache line write backs before A's failure. When the value turns out to be 2, the cache line constraint for A is updated to $[2, 4)$. If machine C were to load from `y` instead, the cache line constraint for B would be used, producing the only possible value `y=5`. In the general case with k machine failures, the system needs to keep track of at most k cache line constraints per cache line, one for each failed machine. The number could be less than k because a cache line might not be written back between two failures, and the failed machines would then share the same constraint.

4 Algorithm

In this section, we present the model checking algorithm used by CXLMC. Differences from Jaaru's algorithm are indicated by shaded boxes. We first establish the following notations to be used in the presentation:

- σ is a sequence number uniquely assigned to each store, `clflushopt`, and `sfence` based on the order they take effect on the cache.
- σ_{curr} is the global sequence number to be assigned to the next instruction that takes effect on the cache. It also serves as a timestamp in the model checker.
- $\tau \in$ T denotes a thread. It is assumed to be the current thread executing the code unless explicitly defined elsewhere.
- $\mu \in$ M denotes a machine.
- μ_{curr} denotes the current machine.

- Φ_{curr} denotes the current set of failed machines. We will also refer to this as the current failure set.
- μ.cacheline maps a cache line address to the constraint for the most recent write back of the cache line before μ's failure. If μ is a live machine, then it maps to the constraint for the current most recent write back of the cache line.
- queue maps an address *addr* to a sequence of triplets of the form $\langle val, \sigma, \mu \rangle$. Each triplet represents a store that contains its value, its sequence number, and the machine that issued it. The stores in each sequence are ordered from earliest to latest by sequence numbers.
- S_τ denotes a sequence for each thread τ that contains its store, sfence, and clflush instructions that have not taken effect in the cache.
- F_τ denotes a sequence for each thread τ that contains its clflushopt instructions that have not taken effect in the cache.
- t_τ denotes the time of the last sfence committed by thread τ
- $t_{\tau,\text{CacheID}(curr)}$ denotes the time of the last store or clflush committed by τ.

4.1 Memory Model

CXLMC adopts Jaaru's approach of emulating the target memory model using a store buffer S_τ, a per thread τ flush buffer F_τ to reorder instructions, and a map queue to hold globally visible stores. As machines have independent failure domains, cache line constraints are associated with each machine μ, rather than an execution epoch as required for persistent memory, while the other components remain unchanged.

The x86 memory model maintains the order of store, sfence, and clflush instructions executed by a thread. A clflushopt instruction has weaker ordering properties, for example, it may be reordered across later clflushopt, clflush, and store instructions, but not across later sfence instructions. To implement these ordering properties, CXLMC inserts these instructions into a per-thread store buffer S_τ, and these operations do not take effect on the cache until they are removed from the store buffer in a FIFO fashion. To implement the weaker ordering property of the clflushopt instruction, it is inserted into the flush buffer F_τ after being removed from the store buffer S_τ, so that it may be reordered with later stores and clflushopt instructions. Reorderings with earlier clflushopt instructions and stores on different cache lines are implemented by updating the beginning of the cache line constraint upon exit from F_τ. Algorithm 1 presents pseudocode for the execution of instructions.

The first four functions simply enqueue their instructions into the store buffer S_τ along with other relevant data to simulate reordering with later instructions, while mfence takes effect immediately in Exec_MFENCE. It empties F_τ and S_τ, serializing all memory and cache instructions issued by the current thread. The enqueued instructions then wait to be processed by the commit functions in Algorithm 2.

Algorithm 1 Algorithm for executing instructions

```
1:  function Exec_Store(addr, val)
2:      Enqueue ⟨store, addr, val⟩ into S_τ.
3:  function Exec_CLFLUSH(addr)
4:      Enqueue ⟨clflush, addr⟩ into S_τ.
5:  function Exec_CLFLUSHOPT(addr)
6:      Enqueue ⟨clflushopt, addr, σ_curr⟩ into S_τ.
7:  function Exec_SFENCE
8:      Enqueue ⟨sfence⟩ into S_τ.
9:  function Exec_MFENCE
10:     Commit all entries in S_τ.
11:     Flush F_τ.
```

Algorithm 2 Algorithm for committing instructions

```
1:  function Commit_SB(store, addr, val)
2:      σ_curr := σ_curr + 1
3:      Enqueue ⟨val, σ_curr, μ_curr⟩ into queue(addr).
4:      t_{τ,CacheID(addr)} := σ_curr
5:  function Commit_SB(clflush, addr)
6:      σ_curr := σ_curr + 1
7:      cl := μ_curr.cacheline(addr)
8:      cl.begin := σ_curr
9:      t_{τ,CacheID(addr)} := σ_curr
10: function Commit_SB(clflushopt, addr, σ)
11:     Enqueue ⟨addr, max(σ, t_{τ,CacheID(addr)}, t_τ)⟩ into F_τ.
12: function Commit_SB(sfence)
13:     σ_curr := σ_curr + 1
14:     Flush F_τ
15:     t_τ := σ_curr
16: function Commit_FB(addr, σ)
17:     cl := μ_curr.cacheline(addr)
18:     cl.begin := max(cl.begin, σ)
```

We next discuss the functions that commit operations from the store buffer that appear in Algorithm 2. We use shaded blocks to indicate how CXLMC's algorithm differs from Jaaru's algorithm. The commit function for stores, Commit_SB(store, *addr*, *val*), increments the current sequence number and inserts the store with the sequence number into queue(*addr*). At this point, the store is considered to be in the cache and is visible globally. It also updates $t_{\tau,\text{CacheID}(addr)}$ that will be used to prevent the reordering of later clflushopt to the same cache line.

The commit function for clflush, Commit_SB(clflush, *addr*), increments the sequence number and sets it as the lower bound of the current cache line constraint. This signifies that the cache line is written back to memory at this timestamp. It also updates $t_{\tau,\text{CacheID}(addr)}$ to the current sequence number.

As mentioned previously, clflushopt's weaker ordering constraints require special treatment in

CommitSB(clflushopt, $addr, \sigma$). A clflushopt can reorder with earlier stores and clflush instructions to different cache lines, we determine the earliest timestamp that clflushopt could take effect by reordering with earlier stores and clflush committed from S_τ as the maximum of the following timestamps: (1) σ, the timestamp when clflushopt was executed, (2) $t_{\tau,\texttt{CacheID}(addr)}$, the last store or clflush instruction committed by τ to the same cache line, and (3) t_τ the last sfence committed by τ. This timestamp is inserted into the flush buffer F_τ along with the flush address, which simulates the clflushopt's reordering with later instructions.

The commit function for sfence, CommitSB(sfence), empties F_τ to serialize previous clflushopt instructions executed by τ, and updates t_τ with an incremented current sequence number to prevent reordering with later clflushopt instructions.

Finally, when a clflushopt is committed from F_τ, it takes effect when the function CommitFB($addr, \sigma$) updates the lower bound of the cache line constraint with the timestamp calculated in CommitSB(clflushopt, $addr, \sigma$).

4.2 Read-From Set Construction

Here, we discuss the implementation of loads in CXLMC separately, as the lazy approach shifts most of the work to load operations. When an instrumented load calls into the model checker, the set of all possible load results at that point is constructed by the function BuildMayReadFrom($addr$). Algorithm 3 presents pseudocode for the BuildMayReadFrom function.

Algorithm 3 CXLMC's read-set construction algorithm

```
1:  function ScanStores(addr, Φ, σ_start)
2:      return {⟨val, σ, μ, Φ⟩ | σ ≤ σ_start ∧
3:          queue(addr) = m_1.⟨val, σ, μ⟩.m_2 ∧
4:          (∄⟨val', σ', μ'⟩ ∈ m_2.μ' ∈ Φ ⟹
5:              σ' ≤ μ'.cacheline(addr).begin) ∧
6:          (μ ∈ Φ ⟹ σ < μ.cacheline(addr).end)}
7:  function BuildMayReadFrom(addr)
8:      if ∃val.S_τ = b_1.⟨addr, val⟩.b_2 ∧ ∀val'.
9:          ⟨addr, val'⟩ ∉ b_2 then
10:         return {⟨μ_curr, 0, val, Φ_curr⟩}
11:     r := ScanStores(addr, Φ_curr, σ_curr)
12:     Φ_new := Φ_curr
13:     while ∃⟨_, σ, μ, _⟩ ∈ r. μ ∉ Φ_new ∧ μ ≠ μ_curr ∧
14:         σ > μ.cacheline(addr).begin do
15:         Φ_new := Φ_new ∪ {μ}
16:         r := r ∪ ScanStores(addr, Φ_new, σ − 1)
17:     return r
```

BuildMayReadFrom serves the same purpose as the function of the same name from Jaaru [33]. We briefly summarize how Jaaru constructs the read-from set for persistent memory programs, then describe how CXLMC does the construction for CXL shared memory.

For persistent memory, full-system failures divide the program into separate executions. If there are no failures, the program can only read from the latest store to the address. Otherwise, the previous execution before the failure is searched for a set of stores that falls in the range of the cache line constraint of the execution. If a store in the execution occurs before or on the beginning of its cache line constraint, it must have been persisted and overwrites previous stores, and the search can terminate. Otherwise, the search continues to earlier executions until some store that overwrites previous stores is found.

ScanStores. Generalizing the set construction to CXL memory with per-machine failure domains, there is no longer a notion of executions separated by failures, and the set of failed machines, along with a timestamp for the last failure become parameters of the search. Therefore, we present a helper procedure ScanStores($addr, \Phi, \sigma_{start}$) that constructs a read-from set given a fixed failure set and timestamp.

With multiple machines, stores from any live machine overwrite previous stores, and the same is true for stores from failed machines that must have been persisted. Therefore, the set ScanStores($addr, \Phi, \sigma_{start}$) constructs only includes the latest store that belongs to either of these two categories, and ignores all previous stores. A store must have been persisted if it is from a failed machine, and its sequence number is on or before the start of the latest cache line write back interval before the failure, and therefore lines 4-5 capture the condition for stores that overwrite previous stores. Additionally, stores from failed machines may have been persisted only if their timestamps are earlier than the latest possible write back time before the failure, which is checked by line 6.

In other words, ScanStores($addr, \Phi, \sigma_{start}$) searches inside queue($addr$) starting from the store at σ_{start} and moving earlier in time. The search does not terminate until it finds either a store from a live machine or a store that must have been persisted, and adds the other stores it encounters to the set if they satisfy the cache line end condition.

BuildMayReadFrom. We next discuss CXLMC's BuildMayReadFrom procedure. For a load from $addr$, lines 8-10 first implement local bypassing in the TSO model by searching the local store buffer for the latest store with a matching address. The placeholder sequence number 0 is used as these stores that have not reached the cache and thus are not yet assigned sequence numbers. The function then constructs the read-from set r by calling the function ScanStores in line 11 with the current set of failed machines Φ_{curr} and the current sequence number σ_{curr}. Using σ_{curr} in this call lets the search in ScanStores start from the latest store in queue($addr$).

Next, the function initializes a failure set Φ_{new} with Φ_{curr}, to be updated in the loop from lines 13-16. The loop continues as long as the read-from set r contains a store that may not

have been written back if the issuing node were to fail, *i.e.*, its sequence number σ is later than the start of the node's cache line constraint. Such a store must be unique if found, since r contains at most one store from a live machine. Inside the loop, the function then tries to inject a failure of the machine μ by adding μ to Φ_{new} in line 15, and expands the set of candidate stores by searching through stores before σ with a call to ScanStores($\Phi_{new}, \sigma - 1$) in line 16.

The motivation for injecting failures here is that a store from a different live machine overwrites earlier stores, but the overwriting is reverted if the machine fails and the overwriting store is not persisted. On the other hand, if this store is read by a remote load, earlier stores will be permanently overwritten as the cache line must have been written back. To ensure the exhaustiveness of the read-from set, we have to consider the possibility that the writing machine fails before the load, and we repeatedly expand the set r as long as a store overwriting other stores could be reverted. This makes whether a store in r may be read depend on which machines have failed, and therefore we include the failure set along with each candidate store in line 2 and line 10 to tell the model checker which machines to fail when a store is chosen.

When the loop terminates, the set r will contain all possible stores that the load may read from. Lastly, the read-from set r is returned in line 17. This function runs in time linear in the size of S_τ and queue($addr$), as each store in them only needs to be examined at most once.

Side Note. Memory poisoning can be implemented by performing the following checks after local bypassing in lines 8-10. If the latest store in queue to any location on the same cache line as *addr* is later than the cache line end constraint for their machine, the cache line must be poisoned. Otherwise if any such store is between the beginning and the end of the constraint, the model checker examines the latest one and has a choice of either making the cache line poisoned and move the cache line end constraint before the time of the store, or not making the cache line poisoned and move the cache line begin constraint past the time of the store. When a machine reads a poisoned cache line, the model checker may simply trigger a runtime exception.

DoRead. CXLMC uses a back-tracking based search to select a store from the output of BuildMayReadFrom to read from. Once this store is selected, the procedure DoRead in Algorithm 4 updates the state of the model checker to reflect this choice. First, in lines 4-5, any new addition μ' to the updated failure set Φ is added to the current failure set Φ_{curr} by calling the function Fail, and CXLMC halts the execution of μ'. Then, the function picks out the relevant cache line constraint based on *addr* and the machine μ of the store. It considers two cases: (1) If the machine has failed, the cache line constraint is updated to be between the sequence number of the chosen store and the next store in the cache to *addr* in lines 7-10, essentially locking into the chosen store as the only one that may be loaded in the future among the stores from μ to *addr*. (2) If the store comes from a live machine that is different from the current one, the cache coherence of CXL memory requires the cache line to be written back to memory after the store, and the start of the cache line constraint is updated to be at least the sequence number of the store in lines 11-12.

Algorithm 4 Algorithm for DoRead

```
1:  function Fail(μ)
2:      Stop μ' and add it to Φ_curr
3:  function DoRead(addr, val, σ, μ, Φ)
4:      for μ' ∈ Φ\Φ_curr do
5:          Fail(μ')
6:      cl := μ.cacheline(addr)
7:      if μ ∈ Φ_curr then
8:          cl.begin := max(cl.begin, σ)
9:          if ∃⟨val', σ', μ', _⟩ such that queue(addr) =
            m1.⟨val, σ, μ⟩.⟨val', σ', μ'⟩.m2 then
10:             cl.end := min(cl.end, σ')
11:     else if μ ≠ μ_curr then
12:         cl.begin := max(cl.begin, σ)
```

4.3 Exploration

Given the ability to construct read-from sets from the previous section, we present the exploration algorithm in Algorithm 5.

The Explore procedure searches through the space of possible executions for the program. The Explore procedure takes in an execution e and continues recursively as long as the set of enabled threads is not empty. In each iteration, it may choose to commit from a store buffer in line 6 by picking a thread with a non-empty store buffer and commits the head of the store buffer. The commit is processed by one of the commit functions from Algorithm 2. Otherwise, the model picks any enabled thread and executes its next action. For a load action, the algorithm builds the read-from set with BuildMayReadFrom and continues after loading from each possible store in turn with DoRead in lines 10-13, while the other actions are processed by one of the execute functions from Algorithm 1.

A key difference between CXLMC and Jaaru's exploration algorithms lies in the failure injection policy from the shaded block on line 16, to account for the fact that machines now fail independently. The failure injection proceeds as follows: if the next action increments the start of a cache line constraint past a store by a live machine μ, CXLMC injects the failure of μ, as this increment reduces the set of possible post-failure load results from that address in the future. Such an instruction may only be a flush. While loads may also change cache line constraints, they are handled by the shaded block from 10, and the failure injection takes place inside BuildMayReadFrom and DoRead as previously explained.

Algorithm 5 Algorithm for Explore

```
1:  function Explore(e)
2:      if e.enabled is empty then
3:          return
4:      if choose to commit then
5:          pick any thread τ with non-empty store buffer S_τ
6:          commit head h of S_τ
7:          Explore(e)
8:      else
9:          pick any thread τ ∈ e.enabled
10:         if next action a of τ is a load to addr then
11:             rfset := BuildMayReadFrom(addr)
12:             for ⟨val, σ, μ, Φ⟩ ∈ rfset do
13:                 Explore(e.DoRead(val, σ, μ, Φ))
14:         else
15:             Explore(e.executeAction(a))
16:             if a is a flush that increments a cache line constraint
                begin past a store from μ ∉ Φ_curr then
17:                 Explore(e.Fail(μ))
```

4.4 Atomic Sequences

Locked atomic instructions such as compare-and-swap (CAS) and atomic exchange have fence-like semantics in x86. They are equivalent to the sequence `mfence`, load, store, and `mfence`. and CXLMC implements them by executing this sequence atomically.

The problem of mixed sized accesses is handled in a similar manner. In the general case, loads and stores may have multiple sizes, and loads may read from multiple stores that overlap with its range. For example, in `C/C++`, a 64-bit load of a struct with two 32-bit fields may read from two stores, one updating each of the fields. CXLMC executes each multi-byte load as a sequence of single byte loads atomically and returns the concatenated result while carefully ensuring the correct semantics for multi-byte atomics stores.

4.5 Optimization

As programs perform loads frequently, we observed a significant overhead from constructing the entire read-from set on every load. Even for small read-from sets, maintaining a container data structure incurs overhead, especially when each element in the sets contains a separate machine failure set. Therefore, we take advantage of our decision point mechanism, which will be explained in Section 5, to lazily search for elements in the read-from set.

Each time a candidate store is found by the search, we insert a binary decision point that decides whether to choose this store immediately and return, or to ignore it and continue searching. This decision point ensures that a different store will be chosen in the next execution, turning the n-ary choice among elements of the read-from set into a series of binary choices, and eventually every store in a read-from set will be chosen once. As we now only search for one candidate store for each load, there is no longer a need for a container to hold the read-from set.

Additionally, this frees us from having to maintain multiple versions of the failure set, eliminating many copy and allocation operations. Now the algorithm only needs to update a single machine failure set corresponding to the store to be returned.

This is an alternative implementation of set construction that was not used by Jaaru. It might also improve the performance of Jaaru, if retrofitted, but the improvements will likely be less significant as failure sets are not needed for full-system failures.

4.6 Novelty

CXLMC's model checking algorithm presented above generalizes Jaaru's [33] algorithm from a full-system failure model to a partial failure model. A full-system failure model divides a single run of a model checker into multiple failed executions preceding the current execution with no overlap. A partial failure model is more complicated; there is a single execution with individual machine failures at various times, which means: (1) Different machines may impose constraints on the same cache line, requiring generalizing the constraint representation. (2) A load at any point may encounter stores from live machines interleaved with those from failed machines, which must be handled by our read-from set construction. (3) When processing a load under CXL, CXLMC must consider whether to inject a machine failure to allow reading from additional stores. (4) Due to the coherence mechanism, a remote load from a store effectively persists that store, requiring updates to the algorithm. These do not have an equivalent in the PM setting. Another major difference with prior work on persistent memory model checking is that CXLMC adds support for multi-process execution and synchronization between processes. This requires significant changes to the model checker's design.

As such, CXLMC's algorithm achieves the same goal as Jaaru's, but in the setting of a distributed system with partial failures. Compared to tools that inject failures in distributed systems [23, 76, 81], CXLMC differs in that it simulates CXL shared memory's failure model, and employs an exploration strategy suitable for this purpose. It performs a form of DPOR [25] that identifies non-commutative cache line write backs and failure injection sites based on cache line constraints, and explores all equivalent classes of executions of a CXL shared memory program under all partial failure scenarios and a fixed thread interleaving.

The algorithm can also explore alternative interleavings for fuzzing purposes by varying the thread selection policy. Efficient exploration of all possible failure scenarios and thread interleavings remains an open problem, even for the more limited full-system failure model due to the very large number of executions.

5 Implementation

This section describes the implementation of CXLMC. CXL program are compiled using our LLVM compiler pass to instrument all memory instructions, both atomic and non-atomic, along with fence and flush instructions. These programs are then compiled into dynamic libraries to be loaded by CXLMC. The shared memory is first allocated and initialized, and then any number of user programs can be loaded and forked, which ensures the shared memory space is mapped to the same address in all forked processes. The shared memory space holds all of the model checker data, scheduler and thread data, along with the simulated CXL memory.

In order to deterministically replay the program each time CXLMC is run, a scheduler controls which thread is running. Multithreading within a process is supported by context switching using the ucontext library when the process is currently scheduled to run. Processes that are not scheduled to run busy wait in a loop. If the number of processes exceeds the number of cores, CXLMC also supports synchronization through futexes. At the end of each model checker run, the last process to complete performs cleanup of model checker data, and signals all other processes to re-execute if there are more executions to explore.

One problem that may arise from emulating threads with contexts is TLS (thread local storage). When the user program creates a thread, a real thread is created and the pointer to its TLS space is taken from the %fs register which is restored when a context switch occurs.

In the case of programs that use mutexes from the pthread library, we have modified their behavior so that if the process that acquired the mutex terminates, the mutex will be automatically released, which is one of the assumptions made by the authors of RECIPE [49]. The mutex also has an API call that returns whether a mutex was released by its owner or because its owner failed. This allows applications to handle cases where the mutex and its protected data are affected by partial failures of remote nodes that occurred after recovery has begun.

We store each decision made in a node stack that represents a depth-first search of a decision tree. Once an execution completes, the next execution explores a different path using the node stack to determine which decisions have already been made. Each possible failure injection point uses a binary decision point to determine whether the crash actually occurs in that execution.

6 Evaluation

In this section, we evaluate CXLMC's ability to find bugs in benchmarks CXL programs, along with its performance. We report our system configuration in Table 2.

CPU	8-core 3.70GHz Intel® Xeon® CPU E3-1245 v6
Memory	32 GB DDR4 2133 MHz
OS	Ubuntu 20.04.6 LTS
Compiler	clang version 20.0.0

Table 2. System configuration.

There are limited applications available for CXL shared memory at this point, as CXL shared memory is not yet commercially available. To evaluate CXLMC, we found benchmarks from two sources. First, we used the benchmarks from the RECIPE [49] benchmark suite of persistent memory index structures, making modifications as necessary to support shared memory. This set of benchmarks was commonly used in the evaluation of most persistent memory bug-finding tools. Second, we used benchmarks from the CXL-SHM tool [85].

The RECIPE benchmarks are a set of concurrent and crash-consistent indexes for persistent memory. LLVM could not compile P-HOT from RECIPE, so it is not included in our benchmarks. Our evaluation includes the remaining six benchmarks from RECIPE. Since the RECIPE data structures were not intended to be used in CXL programs, some modifications were required. In P-ART, locks do not use mutexes and instead used a bit in a version field. This was modified to reserve some of the bits to hold the id of the process owning the lock in order to unlock if the process's machine fails. In persistent memory programs, this is typically done in a recovery procedure, which cannot run in CXL programs since partial crashes do not result in a full restart of the system.

We also used two benchmarks from CXL-SHM [85], a memory allocator for CXL memory that operates under a partial failure model.

We did not use PMDK [20] benchmarks as Jaaru did because PMDK only supports a full-system failure model rather than CXL's partial failure model, and it is difficult to adapt PMDK to a partial failure model.

While similar bug detection tools have been evaluated on Redis and Memcached, these benchmarks only support a full-system failure model. Redis uses PMDK transactions and thus the same problem as the PMDK benchmarks. Memcached was also not included as significant development effort would be required to adapt it to support a partial failure model.

6.1 Bug Detection

We begin with the RECIPE benchmarks. We run each benchmark with 2 processes and varying number of keys and threads. We were able to find 22 bugs, 7 of which are new. We report these bugs in Table 3. Many of these bugs were previously found by Jaaru and were caused by missing flushes that resulted in segmentation faults or assertion failures. Once we found a bug, we corrected the implementation and reran the program until no more bugs were found.

We have made some modifications to these benchmarks prior to searching for bugs. First, API calls that allocated memory for persistent data were replaced with API calls to CXL memory allocators. Second, the original benchmarks only checked for the presence of inserted keys in the post-crash execution. Since this does not work in a partial-failure model, we check for the presence of inserted keys in the remaining threads after keys are inserted.

We next discuss the new bugs we found. Bug 4 in FAST_FAIR is a result of incorrect padding in the header class. The header class consists of fields totaling to 43 bytes along with 5 bytes of padding. However, one of the fields is a 2-byte integer that would not aligned if packed with the previous fields, resulting in an extra byte of padding. This caused some of the entries in the record array to be on multiple cache lines, resulting in partial flushes. It is possible that Jaaru did not find this bug since it may require a specific memory configuration to occur. Bug 7 results when the process crashes in the middle of inserting an entry in the middle of the records. Doing so requires that each entry be moved into the next slot and terminating in the middle of this results in an entry being listed twice. This bug may have been missed by Jaaru due to differences in thread schedules.

Bug 10 in P-ART was caused by atomicity violations. In `N4`, `N16`, and `N48` nodes, there are two 4-byte fields used for counting: one used for the number of child nodes currently in the array and one for the largest number of child nodes that were previously in the array. When a child is inserted, these fields are incremented as the commit for the operation. However, these fields were not incremented atomically, resulting in some logic errors if only one field was able to be persisted. Bug 11 was caused by searching for children in the full array of `N4` rather than only checking up to the number of nodes. This resulted in a logic error if the process crashed after a child was inserted into `N4` but before the counter fields are incremented to commit the operation. In `N16`, the object consists of an 8-byte prefix, two 4-byte counters as previously mentioned, followed by an array of sixteen 1-byte keys and an array of sixteen child pointers. When inserting a child node to the `N16`, the child pointer and the incremented counters are flushed, but the key is incorrectly assumed to be flushed along with the counters. However, it is possible for the `N16` to lie in between two cache lines and for the key array to be separated from the counters, resulting in Bug 12. Bug 13 occurs when a node needs to be split and the process crashes in the middle. The prefix of the node is modified after the parents have been modified to point to the split nodes. In a non-crashing scenario, the prefix cannot be read until the operation is completed as a result of the locks on the nodes that were acquired. However, if the process crashes in the middle of the operation, the locks will be released before the prefix can be modified, resulting in incorrect results when searching through the tree. We found these bugs when running the benchmarks with 48, 50, 128, and 256 keys respectively. These bugs may also be triggered with full-system failures, and the reason why they were not previously found by Jaaru might be because its evaluation used smaller numbers of keys. Additionally, P-ART does include a recovery procedure that nullifies values in the child arrays if the counters failed to persist. However the recovery procedures only run at the start of a post-crash execution in a full system failure model.

#	Benchmark	Type of Bug
1	CCEH	Missing flush in CCEH constructor
2	CCEH	Missing flush in CCEH constructor
3	CCEH	Missing flush in CCEH constructor
4	FAST_FAIR*	Incorrect padding in header
5	FAST_FAIR	Missing flush in header constructor
6	FAST_FAIR	Missing flush in entry constructor
7	FAST_FAIR*	Missing failure detection in key insertion
8	FAST_FAIR	Missing flush in btree constructor
9	P-ART	Missing flush during key creation
10	P-ART*	Count fields not updated atomically
11	P-ART*	Missing bounds check for N4 children
12	P-ART*	Missing flush in N16 insertion
13	P-ART*	Node prefix not updated atomically
14	P-BwTree	Missing flush of GC metadata pointer
15	P-BwTree	Missing flush of GC metadata
16	P-BwTree	Missing flush in AllocationMeta constructor
17	P-BwTree	Missing flush in allocation
18	P-BwTree	Missing flush in BwTree constructor
19	P-CLHT	Missing flush in clht constructor
20	P-CLHT	Missing flush for hashtable object
21	P-CLHT	Missing flush for hashtable array
22	P-MassTree*	Missing failure detection in key insertion

Table 3. Bugs found in RECIPE. Bugs with a * are new.

Bug 22 from P-MassTree can only occur in a partial failure model, which is why it was not found previously. Nodes are checked if recovery is needed during traversal, which only covers full system crashes. Normally, locks are used to prevent the thread from accessing the node while another thread is modifying it. The recovery process checks the state of the locks to determine if a crash has occurred. However, it is possible that after traversal, another process crashes leaving the node in an inconsistent state, causing an error. In this case, our lock API can determine if the lock was released due to a process failure, and if so, part of the recovery procedure can be rerun to fix the inconsistency.

We next discuss the 2 new bugs found in the CXL-SHM benchmarks. We ran each benchmark together with the provided `recovery_check` program in separate processes. The `recovery_check` program performs partial failure recovery and checks for unfreed memory allocated by failed processes. For bug 1, we found that the recovery procedure was not able to correctly garbage-collect a crashed kv program. Comments in the code suggest that recovery for kv data is yet to be implemented due to an ABA problem. For bug 2, we ran the failure monitor continuously as described in the paper [85], together with the `stress_test` benchmark. We

#	Benchmark	Type of Bug
1	kv*	Unimplemented free procedure
2	test_stress*	Divide-by-zero error

Table 4. Bugs found in CXL-SHM benchmarks. Bugs with a * are new bugs.

found that when `stress_test` fails in the middle of a monitor loop, a struct holding page metadata may be zeroed in the later part of the loop iteration, and a field of the zeroed struct may be used in the next loop iteration, resulting in a division by zero error.

We have reported the new bugs found in RECIPE and the CXL-SHM benchmarks to their authors. The authors of CXL-SHM have confirmed that the verification failure is due to missing code and the divide-by-zero error is a bug in their benchmarks and encouraged us to contribute to the repository. We have not yet received a response from the authors of RECIPE.

6.2 GPF Mode

To further evaluate the effects of Global Persistent Flush (GPF), we run the benchmarks with a GPF mode that assumes GPF always succeeds. This implies that no cached value is lost after crashes, and executions follow a regular TSO memory model even in the presence of machine failures. Failures are injected before flush instructions as before to closely match our non-GPF mode, though flushes have no effects when GPF succeeds. Under this setting, none of the RECIPE benchmark bugs in Table 3 are detectable, as they are either triggered by the loss of a cached value alone or by a combination of the loss of cached value and partial failures. The two bugs from CXL-SHM were detected, as they are caused by unexpected partial failures during recovery.

6.3 Performance

Table 5 presents the performance result of running the RECIPE benchmarks on CXLMC, both with and without GPF mode. We ran each RECIPE benchmark with 2 processes of 2 threads each and a total of 10 keys. The intended use of CXLMC is to find bugs in programs. If CXLMC finds a bug, it reports the bug and stops. The developer can then debug and fix the bug in an iterative process until the tool reports no more bugs found. Because of this, we measure the performance of CXLMC on the program once all bugs have been fixed. We were unable to fix all of the bugs that we found in the CXL-SHM benchmarks as some of them require significant code rewrites. As a result, we were unable to measure the performance of CXLMC on those benchmarks. CXLMC is the only bug checking tool for CXL programs, so there are no comparable tools to compare against.

We compare the total number of executions with the number of failure injection points for each benchmark. This ranges from slightly over 1 to about 20 executions per failure injection point. For these benchmarks, the longest running time is just under a minute which is reasonable for a programmer to check if a given program contains a bug.

Benchmarks	#Execs	Time	#FPoints
CCEH	1097	4.31s	1058
FAST_FAIR	77	0.13s	28
P-ART	104	0.22s	92
P-BwTree	1266	5.08s	62
P-CLHT	4128	42.96s	4122
P-MassTree	20	0.03s	16
CCEH_{GPF}	1050	4.33s	1049
$\text{FAST\_FAIR}_{\text{GPF}}$	24	0.12s	23
$\text{P-ART}_{\text{GPF}}$	92	0.24s	91
$\text{P-BwTree}_{\text{GPF}}$	59	0.28s	58
$\text{P-CLHT}_{\text{GPF}}$	4119	44.6s	4118
$\text{P-MassTree}_{\text{GPF}}$	15	0.03s	14

Table 5. Performance Results for CXLMC. The GPF subscript stands for GPF mode. Reported are the number of executions (#Execs), total running time (Time), and number of failure injection points (#FPoints)

The performance results with and without GPF mode are similar in most cases, with only a slight decrease in the number of executions and numbers of crash injection points when GPF mode is enabled. This is not surprising given the commit store pattern we discussed earlier. The only exception is that P-BwTree's executions, numbers of crash injection points, and running time all decreased significantly under GPF. We observed during the evaluation that P-BwTree has a large number of unflushed stores to its garbage collection epoch number, because reading a stale (i.e. strictly smaller) epoch number would not compromise the correctness of garbage collection. Unflushed stores lead to many alternative stores that may be read after a crash when GPF is not guaranteed to succeed, which explains the discrepancy. Compared to P-BwTree, unflushed stores in other benchmarks are relatively rare.

7 Related Work

Existing work on CXL memory focuses primarily on system support [52, 78, 85, 87], applications [36, 40, 51], and performance evaluations [34, 42, 68, 70, 73, 74, 77, 84]. Wang et al. studied the effects of CXL memory-tiering and page-interleaving policies on HPC and LLM workloads [73] and presented an object-based interleaving policy. Li et al. introduced a performance characterization framework for CXL memory based on analyses of 256 workloads, and developed new page-interleaving and tiering policies based on the results [53]. Wu et al. studied the impact of CXL memory topology on HPC and LLM workloads [77]. Wang et al. developed a benchmark suite to understand the performance of heterogeneous CXL memory systems [74]. Tang et al. investigated the performance and cost model of ASIC-based CXL devices [70]. Ji et al. focused on CXL type-2 devices along

with their applications in Linux memory optimization [42]. Yang et al. studied the performance of CXL-tiered memory at the system and architectural level [84] There has been a body of work that discusses how CXL shared memory can be used to optimize distributed processing of memory-intensive workloads [3, 17, 19, 77, 78].

In comparison, work that focuses on correctness has been scarce, with the exception of CXL0 [10], an abstract programming model for the CXL protocol, and Tan et al's effort [69] on formally verifying cache coherence of the `CXL.cache` protocol, which are complementary to our tool that targets CXL memory software.

Several recent works [13, 17, 41, 78, 80] discuss emerging applications of cache-coherent CXL shared memory. While current systems mostly assign private CXL memory partitions to each node, there is increasing interest in leveraging CXL shared memory for cross-node coordination and communication in the form of RPC [56, 57], AllGather operations [8], database transactions [36, 39, 40, 51], and memory object sharing [63, 72]. When devices supporting CXL 3.0 become commercially available, we expect this use case to become more popular.

Most closely related to CXLMC is the body of work on bug-finding in persistent memory programs due to the similar memory semantics and failure models. Yat [47] uses an eager model checking approach that enumerates all post-crash cache states. Jaaru [33] introduced cache line refinement to perform stateless model checking on PM programs. PSan [31] builds on Jaaru and applies a robustness condition to discover missing flush bugs, while Yashme [32] checks for atomicity violations when writing to persistent memory. In terms of testing and dynamic checking, XFDetector [54] tracks the persistency and consistency state of data as a finite state machine and relies on programmer annotations to identify commit variables, and PMTest [55] checks the persistency of and ordering relations between writes to PM against user-specified rules. Pmemcheck [2] uses binary rewriting to detect missing and redundant flushes as well as memory overwrites. Agamotto [61] applies symbolic execution to explore different paths of a PM program.

There is also a body of work on detecting crash-consistency bugs in file systems. Janus [79] and Hydra [44] perform fuzzing on file systems by mutating disk images and file operations. B3 [58] uses bounded testing to explore a bounded state space, while EXPLODE [82], FiSC [83], and SAMC [50] use model checking to systematically explore the state space of file system implementations. Their techniques may be transferable to the CXL memory setting, although the differences in access granularity, failure model and memory semantics have to be considered.

The problem of partial failure has been well-studied in the context of distributed systems [21, 28], with a variety of frameworks and tools for testing, model checking, and formally verifying fault tolerance of distributed systems [9, 23, 75, 76, 81]. Work on persistent atomicity [35] builds the abstraction of a crash-tolerant shared memory by using message passing in a distributed system. Failure injection is commonly used by tools such as LEGOLAS [76], MODIST [81] and ORCHESTRA [23] to expose bugs in distributed systems. CXLMC's failure injection policy is distinct as it is designed to work with our DPOR technique and injects partial failures based on cache line constraints to explore post-failure states of the shared memory with consideration for CXL shared memory's caching and persistence semantics. In comparison, fault injection tools for distributed systems generally assume a message-passing model with no shared state and no equivalent notion of hardware-implemented caching and persistence. However, general techniques for failure injection [9, 76] could be adapted and applied on top of a CXL memory model such as that provided by CXLMC.

Researchers have applied a number of abstract models and correctness conditions for persistent memory programs [12], some of which were initially devised for distributed systems [7, 35]. Partial failures have been assumed in some of them, such as the Parallel Persistent Memory model [14], strict linearizability [7], nesting-safe recoverable linearizability [11], and persistent atomicity [35]. However, this assumption has not been adopted by existing testing/verification tools because sharing of PM has not been a common use case and lacked hardware support before CXL 3.0. The failure model in this paper is most similar to the one in the original formulation of strict linearizability [7], which assumes that processes may fail independently and failed processes may not be started again.

Stateless model checking has been explored to find bugs in concurrent data structures [29, 30, 59, 60], supporting the TSO/PSO model [4, 38, 86], the release-acquire fragment of C/C++11 [6, 45], and the full C/C++11 model [62]. To improve the efficiency of stateless model checking, various forms of dynamic partial order reduction (DPOR) [16, 25, 48, 65, 71, 86] have been proposed to avoid exploration of equivalent executions. Our formulation of DPOR for CXL memory has some similarity to prior work [5, 6, 15, 45, 46] that defines execution equivalence based on read-from relations, as we use cache line constraints to determine possible read-from relations after partial failures.

8 Conclusion

CXLMC is the first model checker for CXL shared memory programs. CXLMC adapts the constraint refinement-based approach to both CXL's shared memory model and CXL's partial failure model. Our evaluation shows that CXLMC effectively finds bugs in our benchmark applications and can model check our benchmark applications typically within a few seconds and within a minute for the longest example.

References

[1] [n. d.]. CXL 3.2 Specification. https://computeexpresslink.org/cxl-specification/.

[2] 2015. An introduction to pmemcheck (part 1) - basics. https://pmem.io/2015/07/17/pmemcheck-basic.html.

[3] 2024. Advantages of CXL Memory Sharing for Emerging Applications. https://computeexpresslink.org/wp-content/uploads/2025/06/CXL_Q2-2025-Webinar_FINAL.pdf.

[4] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. *CoRR* abs/1501.02069 (2015). arXiv:1501.02069 http://arxiv.org/abs/1501.02069

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal Stateless Model Checking for Reads-from Equivalence Under Sequential Consistency. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 150 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360576

[6] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal Stateless Model Checking Under the Release-acquire Semantics. *Proceddings of the ACM on Programming Languages* 2, OOPSLA, Article 135 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276505

[7] Marcos K. Aguilera and Svend Frølund. 2003. Strict Linearizability and the Power of Aborting. https://api.semanticscholar.org/CorpusID:16064246

[8] Hooyoung Ahn, Seonyoung Kim, Yoomi Park, Woojong Han, Shinyoung Ahn, Tu Tran, Bharath Ramesh, Hari Subramoni, and Dhabaleswar K. Panda. 2024. MPI Allgather Utilizing CXL Shared Memory Pool in Multi-Node Computing Systems. In *2024 IEEE International Conference on Big Data (BigData)*. 332–337. https://doi.org/10.1109/BigData62323.2024.10825804

[9] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 331–346. https://doi.org/10.1145/2723372.2723711

[10] Gal Assa, Lucas Bürgi, Michal Friedman, and Ori Lahav. 2025. A Programming Model for Disaggregated Memory over CXL. arXiv:2407.16300 [cs.DC] https://arxiv.org/abs/2407.16300

[11] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing* (2018). https://api.semanticscholar.org/CorpusID:51910876

[12] Naama Ben-David, Michal Friedman, and Yuanhao Wei. 2022. Brief Announcement: Survey of Persistent Memory Correctness Conditions. In *36th International Symposium on Distributed Computing (DISC 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 41:1–41:4. https://doi.org/10.4230/LIPIcs.DISC.2022.41

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

[14] Guy E. Blelloch, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. The Parallel Persistent Memory Model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18)*. Association for Computing Machinery, New York, NY, USA, 247–258. https://doi.org/10.1145/3210377.3210381

[15] Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. 2021. The reads-from equivalence for the TSO and PSO memory models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 164 (Oct. 2021), 30 pages. https://doi.org/10.1145/3485541

[16] Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. 2019. Value-centric dynamic partial order reduction. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 124 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360550

[17] Chen Chen, Xinkui Zhao, Guanjie Cheng, Yuesheng Xu, Shuiguang Deng, and Jianwei Yin. 2025. Next-Gen Computing Systems with Compute Express Link: a Comprehensive Survey. arXiv:2412.20249 [cs.DC] https://arxiv.org/abs/2412.20249

[18] Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021. Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 16–31. https://doi.org/10.1145/3453483.3454027

[19] Jungmin Choi. 2024. CXL Disaggregated Memory Solution for HPC and AI Workloads. https://files.futurememorystorage.com/proceedings/2024/20240807_CXLT-202-1_Choi.pdf.

[20] Intel Corporation. 2020. Persistent Memory Development Kit. https://pmem.io/pmdk/.

[21] Flavin Cristian. 1991. Understanding fault-tolerant distributed systems. *Commun. ACM* 34, 2 (Feb. 1991), 56–78. https://doi.org/10.1145/102792.102801

[22] 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. https://doi.org/10.1145/3669900

[23] Scott Dawson, Farnam Jahanian, Todd Mitton, and Teck-Lee Tung. 1996. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proceedings of Annual Symposium on Fault Tolerant Computing*. IEEE, 404–414.

[24] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 503–516. https://doi.org/10.1145/3445814.3446744

[25] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.* 40, 1 (Jan. 2005), 110–121. https://doi.org/10.1145/1047659.1040315

[26] LLVM Foundation. 2024. LLVM-20. https://github.com/llvm/llvm-project/tree/release/20.x.

[27] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. 2021. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles* (Virtual) *(SOSP 2021)*. Association for Computing Machinery, New York, NY, USA, 100–115. https://doi.org/10.1145/3477132.3483556

[28] Felix C. Gärtner. 1999. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.* 31, 1 (March 1999), 1–26. https://doi.org/10.1145/311531.311532

[29] Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

[30] Patrice Godefroid. 2005. Software Model Checking: The VeriSoft Approach. *Form. Methods Syst. Des.* 26, 2 (March 2005), 77–101. https://doi.org/10.1007/s10703-005-1489-x

[31] Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. 2022. Checking robustness to weak persistency models. In

Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 490–505. https://doi.org/10.1145/3519939.3523723

[32] Hamed Gorjiara, Guoqing Xu, and Brian Demsky. 2022. Yashme: Detecting Persistency Races. In *Proceedings of the Twenty-Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. https://doi.org/10.1145/3503222.3507766

[33] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2021. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 415–428. https://doi.org/10.1145/3445814.3446735

[34] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory pooling with CXL. *IEEE Micro* 43, 2 (2023), 48–57.

[35] R. Guerraoui and R.R. Levy. 2004. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* 400–407. https://doi.org/10.1109/ICDCS.2004.1281605

[36] Yunyan Guo and Guoliang Li. 2024. A CXL-Powered Database System: Opportunities and Challenges. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 5593–5604. https://doi.org/10.1109/ICDE60146.2024.00447

[37] Sudhanva Gurumurthi, Alexander J. Branover, Bryan Hornung, Vinny Michna, Mahesh Natu, and Chris Petersen. 2021. An Overview of Reliability, Availability, and Serviceability (RAS) in Compute Express Link™ 2.0. https://computeexpresslink.org/wp-content/uploads/2023/12/CXL-RAS-Whitepaper-Post-WG-Revision_FINAL.pdf.

[38] Shiyou Huang and Jeff Huang. 2016. Maximal Causality Reduction for TSO and PSO. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 447–461. https://doi.org/10.1145/2983990.2984025

[39] Yibo Huang, Haowei Chen, Newton Ni, Yan Sun, Vijay Chidambaram, Dixin Tang, and Emmett Witchel. 2025. Tigon: A Distributed Database for a CXL Pod. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.

[40] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. 2025. Pasha: An Efficient, Scalable Database Architecture for CXL Pods. In *Proceedings of the 2025 Annual Conference on Innovative Data Systems Research*.

[41] 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

[42] Houxiang Ji, Srikar Vanavasam, Yang Zhou, Qirong Xia, Jinghan Huang, Yifan Yuan, Ren Wang, Pekon Gupta, Bhushan Chitlur, Ipoom Jeong, and Nam Sung Kim. 2024. Demystifying a CXL Type-2 Device: A Heterogeneous Cooperative Computing Perspective. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1504–1517. https://doi.org/10.1109/MICRO61859.2024.00110

[43] Artem Khyzha and Ori Lahav. 2021. Taming x86-TSO persistency. *Proc. ACM Program. Lang.* 5, POPL, Article 47 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434328

[44] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 147–161. https://doi.org/10.1145/3341301.3359662

[45] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 17 (December 2017), 32 pages. https://doi.org/10.1145/3158105

[46] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. 96–110. https://doi.org/10.1145/3314221.3314609

[47] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A validation framework for persistent memory software. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 433–438. https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz

[48] Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. 2010. Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. In *Fundamental Approaches to Software Engineering*, David S. Rosenblum and Gabriele Taentzer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 308–322.

[49] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

[50] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 399–414. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa

[51] Alberto Lerner and Gustavo Alonso. 2024. CXL and the Return of Scale-Up Database Engines. *Proc. VLDB Endow.* 17, 10 (Aug. 2024), 2568–2575. https://doi.org/10.14778/3675034.3675047

[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. https://doi.org/10.1145/3575693.3578835

[53] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. 2025. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1203–1217. https://doi.org/10.1145/3676641.3715987

[54] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1187–1202. https://doi.org/10.1145/3373376.3378452

[55] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming*

Languages and Operating Systems (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 411–425. https://doi.org/10.1145/3297858.3304015

[56] 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.

[57] Suyash Mahar, Ehsan Hajyjasini, Seungjin Lee, Zifeng Zhang, Mingyao Shen, and Steven Swanson. 2024. Telepathic Datacenters: Fast RPCs using Shared CXL Memory. *CoRR* abs/2408.11325 (2024). https://doi.org/10.48550/ARXIV.2408.11325 arXiv:2408.11325

[58] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 33–50. https://www.usenix.org/conference/osdi18/presentation/mohan

[59] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 267–280.

[60] Madanlal Musuvathi, Shaz Qadeer, Piramanayagam Arumuga Nainar, Thomas Ball, Gerard Basler, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the Eighth USENIX Symposium on Operating Systems Design and Implementation*. 267–280. http://dl.acm.org/citation.cfm?id=1855741.1855760

[61] Ian Neal, Ben Reeves, Ben Stoler, and Andrew Quinn. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Banff, Alberta, 18 pages.

[62] Brian Norris and Brian Demsky. 2013. CDSChecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *Proceedings of the 2013 Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 131–150. http://doi.acm.org/10.1145/2544173.2509514

[63] Adarsh Patil, Vijay Nagarajan, Nikos Nikoleris, and Nicolai Oswald. 2023. Āpta: Fault-tolerant object-granular CXL disaggregated memory for accelerating FaaS. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 201–215. https://doi.org/10.1109/DSN58367.2023.00030

[64] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency Semantics of the Intel-x86 Architecture. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 11 (December 2020), 31 pages. https://doi.org/10.1145/3371079

[65] Olli Saarikivi, Kari Kahkonen, and Keijo Heljanko. 2012. Improving Dynamic Partial Order Reductions for Concolic Testing. In *Proceedings of the 2012 12th International Conference on Application of Concurrency to System Design (ACSD '12)*. IEEE Computer Society, USA, 132–141. https://doi.org/10.1109/ACSD.2012.18

[66] D Das Sharma and Ishwar Agarwal. 2022. Compute Express Link 3.0. *white paper, CXL Consortium* (2022).

[67] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 81–98. https://www.usenix.org/conference/fast23/presentation/shen

[68] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. 2023. Demystifying CXL memory with genuine CXL-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 105–121.

[69] Chengsong Tan, Alastair F. Donaldson, and John Wickerson. 2024. Formalising CXL Cache Coherence. arXiv:2410.15908 [cs.AR] https://arxiv.org/abs/2410.15908

[70] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 818–833. https://doi.org/10.1145/3627703.3650061

[71] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. 2012. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In *Formal Techniques for Distributed Systems*, Holger Giese and Grigore Rosu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 219–234.

[72] Yong Tian. 2023. Project Gismo Global I/O-free Shared Memory Objects. https://memverge.com/wp-content/uploads/MemVerge-Gismo-FMS2023.pdf.

[73] Xi Wang, Jie Liu, Jianbo Wu, Shuangyan Yang, Jie Ren, Bhanu Shankar, and Dong Li. 2025. Exploring and Evaluating Real-world CXL: Use Cases and System Adoption. In *Proceedings of the 39th IEEE International Parallel and Distributed Processing Symposium*.

[74] Zixuan Wang, Suyash Mahar, Luyi Li, Jangseon Park, Jinpyo Kim, Theodore Michailidis, Yue Pan, Mingyao Shen, Tajana Rosing, Dean Tullsen, Steven Swanson, and Jishen Zhao. 2025. The Hitchhiker's Guide to Programming and Optimizing Cache Coherent Heterogeneous Systems: CXL, NVLink-C2C, and AMD Infinity Fabric. arXiv:2411.02814 [cs.PF] https://arxiv.org/abs/2411.02814

[75] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. *SIGPLAN Not.* 50, 6 (June 2015), 357–368. https://doi.org/10.1145/2813885.2737958

[76] Haoze Wu, Jia Pan, and Peng Huang. 2024. Efficient exposure of partial failure bugs in distributed systems with inferred abstract states. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 70, 17 pages.

[77] Jianbo Wu, Jie Liu, Gokcen Kestor, Roberto Gioiosa, Dong Li, and Andres Marquez. 2024. Performance Study of CXL Memory Topology. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '24)*. Association for Computing Machinery, New York, NY, USA, 172–177. https://doi.org/10.1145/3695794.3695809

[78] Tong Xing and Antonio Barbalace. 2025. Rethinking Applications' Address Space with CXL Shared Memory Pools. In *Proceedings of the 4th Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS '25)*. Association for Computing Machinery, New York, NY, USA, 52–59. https://doi.org/10.1145/3723851.3723858

[79] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*. 818–834. https://doi.org/10.1109/SP.2019.00035

[80] Yi Xu, Suyash Mahar, Ziheng Liu, Mingyao Shen, and Steven Swanson. 2024. CXL Shared Memory Programming: Barely Distributed and Almost Persistent. arXiv:2405.19626 [cs.DC] https://arxiv.org/abs/2405.19626

[81] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent model checking of unmodified distributed systems. In *NSDI 2009*. 213–228.

[82] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. USENIX Association, Seattle, WA. https://www.usenix.org/conference/osdi-06/explode-lightweight-general-system-finding-serious-storage-system-errors

[83] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.* 24, 4 (Nov. 2006), 393–423. https://doi.org/10.1145/1189256.1189259

[84] Yujie Yang, Lingfeng Xiang, Peiran Du, Zhen Lin, Weishu Deng, Ren Wang, Andrey Kudryavtsev, Louis Ko, Hui Lu, and Jia Rao. 2025. Architectural and System Implications of CXL-enabled Tiered Memory. arXiv:2503.17864 [cs.AR] https://arxiv.org/abs/2503.17864

[85] 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. https://doi.org/10.1145/3600006.3613135

[86] Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic partial order reduction for relaxed memory models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 250–259. https://doi.org/10.1145/2737924.2737956

[87] Zhiting Zhu, Newton Ni, Yibo Huang, Yan Sun, Zhipeng Jia, Nam Sung Kim, and Emmett Witchel. 2024. Lupin: Tolerating Partial Failures in a CXL Pod. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems* (Austin, TX, USA) *(DIMES '24)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/3698783.3699377

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009