A Programming Model for Disaggregated Memory over CXL

Gal Assa
Technion
Israel
galassa@technion.ac.il

Moritz Lumme
ETH Zürich
Switzerland
moritz.lumme@inf.ethz.ch

Lucas Bürgi
ETH Zürich
Switzerland
buergiluc@student.ethz.ch

Michal Friedman
ETH Zürich
Switzerland
michal.friedman@inf.ethz.ch

Ori Lahav
Tel Aviv University
Israel
orilahav@tau.ac.il

Abstract

CXL (Compute Express Link) is an emerging open industry-standard interconnect between processing and memory devices that is expected to revolutionize the way systems are designed. It enables cache-coherent, shared memory pools in a disaggregated fashion at unprecedented scales, allowing algorithms to interact with various storage devices using simple loads and stores. While CXL unleashes unique opportunities, it also introduces challenges of data management and crash consistency. For example, CXL currently lacks an adequate programming model, making it impossible to reason about the correctness and behavior of systems on top.

In this work, we present CXL0, the first programming model for concurrent programs over CXL. We propose a high-level abstraction for memory accesses and formally define operational semantics. We demonstrate that CXL0 captures a wide range of current and future CXL setups and perform initial measurements on real hardware. To illustrate the usefulness of CXL0, we present a general transformation that enhances any linearizable concurrent algorithm with durability in a distributed partial-crash setting. We believe that this work will serve as a stepping stone for systems design and programming on top of CXL.

CCS Concepts: • **Hardware → Emerging technologies**; • **Computing methodologies → Concurrent computing methodologies**; **Concurrent algorithms**.

Keywords: CXL; Programming models; Disaggregated memory

ACM Reference Format:
Gal Assa, Moritz Lumme, Lucas Bürgi, Michal Friedman, and Ori Lahav. 2026. A Programming Model for Disaggregated Memory over CXL. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 21–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3779212.3790121

ASPLOS '26, Pittsburgh, PA, USA.

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790121

1 Introduction

With the increasing demand for storage in the cloud and recent improvements in interconnects [52], disaggregated memory has become a viable solution [4, 16, 18, 27, 30, 49, 50, 55, 60, 75]. In disaggregated systems, compute nodes are separated from memory nodes. This offers greater flexibility, scalability, and resilience compared to traditional architectures [2, 30]. Rather than equipping a node to meet peak demand, it can be provisioned with enough memory to satisfy common needs, and additional memory is provided by other nodes during times of high load. As a result, data can be spread over multiple machines, and data transfers can become a major bottleneck [22, 23, 30, 55].

Compute Express Link (CXL) is an emerging standard which aims to facilitate efficient data transfer within data-centers. Several cloud hyperscalers are involved in its development, and it is expected by many to become the future industry standard for data-center interconnects [68]. CXL supports accessing external storage and caching data from remote nodes in a *coherent* manner, which is a significant change compared to current architectures [20]. Furthermore, CXL provides a uniform abstraction for accessing all types of memory, such as DRAM and SSD, across the system via simple loads and stores at cache line granularity. This allows the entire memory domain of the participating machines to be modeled as a large, unified shared main memory [50].

These opportunities also present new challenges. As a centralized system becomes distributed *and* coherent, two key assumptions no longer hold. First, heterogeneous compute units access the same memory, so the outcome of concurrent operations is not governed by a single memory model. Second, the system is no longer a single failure domain. Instead, nodes may crash independently. Special attention is required to ensure correctness under these conditions.

Failure-aware programming has been subject to vast research effort, primarily in the context of full-system crashes [17, 29, 48, 70]. The emergence of non-volatile main memory (NVMM), as a persistent, moderately fast, and byte-addressable medium, gave rise to both theoretical and practical endeavors. Correctness criteria [10, 11, 33, 44, 54], data structures [33, 66, 69, 79], verification approaches [1, 12, 28, 37, 38, 47], and transactional memory frameworks [7, 15, 21, 63] were suggested to benefit from the new technology. Although NVMM has been deprecated as a standalone product [42], it is expected to remain part of the memory hierarchy with CXL [26]. However, existing solutions are not automatically transferrable, because CXL inflates the system model, allowing multiple machines to share the same volatile and persistent memory. As these machines can be CPUs, GPUs, FPGAs, or any type of accelerator, an adequate programming model is required to correctly utilize this new technology.

Despite the expectation of CXL's wide adoption, there is currently no general programming model. This leaves programmers without answers to crucial questions. For example, it is unclear what consistency guarantees memory accesses provide, or whether the local memory model is preserved over the communication protocol. Moreover, existing multi-threaded programs for shared memory do not remain correct in the disaggregated setting, and one urgent area of investigation is how to correctly port this legacy code.

This work aims to close these gaps. We propose an abstraction of CXL's memory operations and define CXL0 as a formal programming model on top of that abstraction. Both stem from a rigorous analysis of the CXL specification [20]. We capture a system model with distributed, coherent, shared memory, that allows for subsystem crashes. Furthermore, we model each node in the system as a subsystem that emits sequences of CXL transactions, thus abstracting away the effect of the internal memory model. Using CXL0 as a tool to reason about the correctness and behavior of concurrent executions, we show that it applies to a wide range of current and future CXL setups. We also address the question of how to design concurrent programs for this new system model. Shared memory programming in a multi-processor, disaggregated memory environment with independent failures is a challenging task. We identify common pitfalls and demonstrate how to mitigate them using a general transformation that equips linearizable algorithms with fault tolerance under CXL0. Our transformation enhances legacy code with resilience to spontaneous subsystem failures, even for algorithms that are not failure-aware.

In summary, the contributions of this work are as follows:

- We provide a high-level abstraction of memory accesses in the CXL specification;
- We introduce CXL0, a programming model for coherent disaggregated memory over CXL;
- We present a transformation that makes linearizable code durable under the partial crash model and formally prove its correctness.

To our knowledge, CXL0 is the first programming model for disaggregated memory over CXL, and our transformation is the first to equip linearizable code with durability in the partial crash model. The rest of the paper is organized as follows. We provide necessary background in §2. In §3, we describe our system model, abstraction and formalize the operational semantics that form CXL0. §4 demonstrates how CXL0 captures current and future CXL setups, and §5 illustrates practical implications on real hardware. In §6, we show how to transform existing concurrent programs for CXL0. §7 reviews related work, and §8 concludes this paper.

The full version of this paper is available in [8], and the code used for the proofs is available in [9].

2 Background

This section provides the essential background and preliminaries for this work. We include a short introduction to the relevant parts of the CXL standard based on its specification.

2.1 Disaggregated memory

Disaggregated memory is a provisioning approach for improving hardware utilization in data centers. The idea is to separate compute from memory, allowing each to scale independently. It is an alternative to provisioning dedicated servers with enough resources to handle peak demand which leads to lower utilization during regular operation. In addition to enabling independent scaling, disaggregated memory also allows independent maintenance and introduces a certain degree of fault tolerance (*i.e.*, memory can outlast a compute unit accessing it).

2.2 Caching and cache coherence

Memory caching techniques are used to increase efficiency by reducing latency when accessing memory. This is accomplished by storing frequently used data in a small, high-speed memory close to the compute unit, which is much faster than main memory. Modern processors are equipped with multiple layers of caches. The presence of caches means that the value of a logical memory address may reside in multiple physical locations (*i.e.*, caches and main memory) simultaneously. Memory caches operate on fixed-size blocks called cache lines for data transfer.

Cache coherence is a system property that ensures *each load operation observes the outcome of the most recent store to the same address that reached the cache.* Cache-coherent systems also satisfy *single-writer multi-reader* exclusion and concurrent writes are impossible. Many cache coherence protocols exist, most of which are aimed at single machines

with multiple CPU cores and a hierarchy of shared and private caches. Examples include MESI, MOESI, and MESIF, and some protocols for distributed settings [60, 71].

2.3 Persistent memory programming

Because systems rely on volatile caches, persistent memory introduces consistency challenges. Stores are typically serviced by caches and are considered complete when they reach local cache rather than main memory. When programming for volatile memory without considering crashes, cache replacement is usually irrelevant thanks to cache coherence. However, when main memory is persistent, crash recovery depends on obtaining a consistent state from persistent memory. Therefore, algorithms *must* consider the order in which updates reach persistent media. To that end, programmers use explicit *flush* instructions, which propagate the content of a cache line to memory, and *memory barriers* to enforce update ordering. This ensures that the system can recover a consistent state from persistent memory regardless of when a crash occurs.

2.4 Linearizability

Consistency in the face of failures can manifest in various ways. The canonical consistency criterion for concurrent objects and algorithms is *linearizability*, as defined by Herlihy and Wing [40]. This criterion asserts that the concurrent history can be projected onto a single timeline in a manner that respects the sequential specification. The notion of linearizability has received multiple extensions in the context of partial and full-system crashes [3, 10, 11, 33, 39, 54]. This work focuses on *durable linearizability* by Izraelevitz et al. [44]. Informally, the latter notion requires that completed operations persist before returning.

2.5 A primer to CXL

This work considers the most recent version of the CXL standard at the time of writing, CXL 4.0. The specification can be found in [20]. The standard is as complex as one would expect, and a significantly shorter introduction to it was published by Das Sharma et al. [25]. Here, we describe only the details necessary for this work and refer the reader to either of the above sources for more information.

CXL consists of three sub-protocols: CXL.io, CXL.cache, and CXL.mem. CXL.io, is a slightly augmented superset of PCIe that is used for device discovery, configuration and power management. CXL.cache implements a cache coherence domain, letting one PCIe endpoint (*e.g.*, an accelerator) cache memory owned by a PCIe root complex (CPU). CXL.mem provides the underlying memory channel for accessing remote memory via ordinary loads and stores. The standard allows for different combinations of sub-protocols and defines three device types. Type 1 devices run CXL.io alongside CXL.cache. These devices only expose processing capabilities (*e.g.*, SmartNICs). Type 3 devices run CXL.io alongside CXL.mem. These devices only expose device memory (*e.g.*, main memory expanders). This work focuses on Type 2 devices, which combine all three sub-protocols. They expose both device memory and processing capabilities, such as GPUs, FPGAs and other accelerators.

All CXL sub-protocols are routable through custom CXL switches, so theoretically, the standard allows to combine dozens or even hundreds of root complexes (*hosts*) and endpoints (*devices*) into a single multi-root fabric in which memory devices can be pooled or shared. In that setting, each host manages its own cache hierarchy and uses CXL.mem for access to remote memory.

Within a single-root coherence domain, *e.g.*, the classic host-device pairing, CXL.cache implements MESI-based coherence on individual cache lines. Importantly, it does not guarantee any ordering between accesses to memory locations on different cache lines. The specification suggests page-granular *host-bias* and *device-bias* modes for Type 2 device memory. In host-bias mode, the host "owns" the memory page, meaning the device must request access from the host before accessing its own memory. In device-bias mode, the device "owns" the page, and can access its memory directly without notifying the host. These two modes represent a tradeoff between low-latency memory access for devices and cache-sharing overhead with the host, but they only apply within a single CXL.cache coherence domain.

We note that the standard provides a variety of low-level transactions for reading, writing, and flushing cache lines, introducing a wide range of options for remote memory access that differ in performance and semantics.

3 A General Programming Model for CXL

In this section, we describe the most general system and failure model, and an abstract model for memory access on top of CXL. Based on these, we define operational semantics which provides us with a programming model for concurrent programs that access memory via the underlying CXL standard. To develop some intuition about valid behaviors of programs under our model, we provide a collection of litmus tests and prove various properties of the model. We present two variations of the programming model for specific hardware features.

3.1 System model

We consider an ideal model of a distributed system which consists of multiple Type 2 devices. Each machine has computational capacity, memory capacity, or both. Memory accesses are cache-coherent. Each memory address is mapped to a single physical volatile or persistent memory location on a specific machine. Memory accesses are performed at cache line granularity. Machines are connected via CXL switches and each machine manages the coherence of the memory attached to it. Machines send and receive CXL messages on

dedicated ports. A system may incorporate several types of processing units (CPUs of different architectures, GPUs, FGPAs, etc.). Attached memory may also be of various types (*e.g.*, SSD, DRAM, or NVMM). Some nodes may be only memory nodes, while others can host no shared memory at all. Deciding which parts of the memory are shared is up to the system designer. The CXL standard accommodates complex topologies and access control, so different memory segments located on a given machine may be shared with different subsystems simultaneously. Potentially, the entire address space may be shared in a cache-coherent manner.

We model each connected node as emitting CXL transactions (writes, reads, flushes) to the network in a specific order. Within each node, different consistency models may govern how instructions across cores propagate to the CXL network. The standard interconnects several machines, each potentially employing diverse underlying architectures with varying consistency guarantees. We do not consider the internal memory model of each machine, as the system is free to reorder messages within the CXL fabric (due to the properties of the underlying PCIe) [25]. Hence, ordering has to be explicitly enforced when accessing remote memory locations, as the local memory ordering is unaware of the possibility of further reordering.

Failure model. Our model permits spontaneous, independent crashes of any machine. A machine crashes with its caches and the portion of shared memory attached to it. We assume caches are volatile, so their contents vanish upon crash. Data is considered lost upon crash unless it reached physical persistent media, or another failure domain before the crash. When a machine crashes, the local state of any thread currently executing on it is lost as well (including program counter, registers, etc.). Upon recovery, new threads are spawned to replace the crashed ones, and all volatile memory is reset to some initial value. The effects of store operations performed by the crashed machine that had not yet propagated into persistence or to another machine are lost. Store operations to addresses belonging to the crashed machine may be lost if not persisted, or may also reside in another machine's cache and propagate at a later time.

3.2 Abstract CXL model

The CXL specification describes multiple types of read and write transactions as part of its sub-protocols. Such transactions refer to low-level transactions that are different from high-level, application ones. We provide an abstraction of these low-level transactions by mapping them to a single abstract read operation, three abstract store operations, and two abstract flush operations. These operations are referred to as CXL0 primitives in this work. In §5.1, we provide a mapping from CXL transactions to CXL0 primitives.

Figure 1 depicts the CXL0 primitives using a simple example. In this example, all primitives are performed by the left node and are labeled with numbers (*e.g.*, ①). Continuous and double lines represent store and flush operations, respectively. Dotted lines represent eviction steps that are induced by the cache replacement policy. An arrowhead marks where the value is guaranteed to be propagated after the instruction or propagation step. Here, we assume that x is a memory address allocated on the left node and y on the right one.

Our model considers a single `Load` primitive. The reason is that we assume cache coherence (*i.e.*, reads-see-last-write), and this work aims at modeling high-level behaviors of concurrent programs rather than the coherence protocol itself. The three abstract CXL0 store primitives are:

- Local Store (henceforth `LStore`): A store to local cache. An `LStore` is considered complete when written to a local cache, and the CXL protocol does not provide any guarantees regarding when a local store is propagated to the physical memory where the stored address belongs. This store type does not generate any data transfer between machines and is likely to be preferable in terms of performance. As the ② label shows, an `LStore` has the same behavior when storing to each of the addresses (x and y).
- Remote Store (henceforth `RStore`): A store to remote cache. An `RStore` is considered complete when it reaches either the address owner's cache or physical memory. CXL does not guarantee when a cache line is written back (namely, propagated) from the owner's cache to the physical memory. An `RStore` to x executed by the left machine is equivalent to an `LStore`, whereas an `RStore` to y (④) results in writing the new value to the right machine's cache.
- Memory Store (henceforth `MStore`): A store that completes only after reaching physical memory. The issuer may consider a write complete only after it has reached its final destination. If the destination memory is volatile and on the same failure domain, `RStore` and `MStore` are semantically equivalent under our failure model. In the example, an `MStore` to x (①) is considered complete only when the new value of x is stored in main memory, and an `MStore` to y (③) results in writing to the physical memory of the right machine.

In addition to loads and stores, we consider two distinct `Flush` primitives, which are used to explicitly evict a cache line from a cache to the next level in the memory hierarchy. Such write-back operations are common in commodity CPUs (*e.g.*, `CLFLUSH` on x86 [43] and `DC-CVAC` on ARMv8 [6]). CXL also introduces the notion of flushing a remote cache line (`CLFlush` in [20]), namely, invalidating a cache line such that the cache controller writes back its contents to remote memory. Our abstraction for flushes is as follows:

- A Local Flush (henceforth `LFlush`) writes back a cache line that is stored locally (*i.e.*, on the executing machine) to the next hierarchy, be it a remote cache or local main memory. `LFlush` to x results in updating the local memory

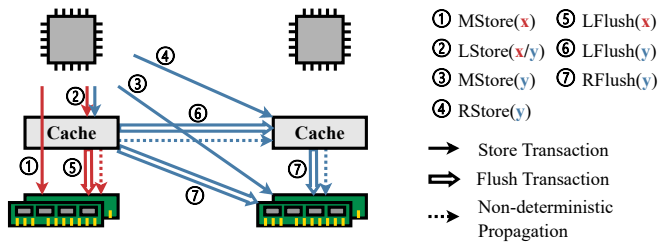


Figure 1. Semantics of CXL0 primitives as a layer of abstraction over CXL store and flush transactions (including non-deterministic propagation).

(⑤), and the effect of `LFlush` to y is an update to the right machine's cache (⑥).

- A Remote Flush (henceforth `RFlush`) writes back a cache line to the corresponding main memory. `RFlush` to y causes a write back from either cache to the right machine's physical memory y (⑦), and `RFlush` to x is equivalent to `LFlush` to x in this example.

The `Global Persistent Flush (GPF)` instruction is also supported by the specification (§9.8 in [20]). It initiates a 2-phase write-back protocol to drain *all* caches of all CXL agents in the cache coherence domain. We regard it as rather complex and highly susceptible to additional machine failures. Moreover, the specification treats GPF as optional and explicitly omits a list of triggering events. Therefore, we capture the behavior of `GPF` in our model for completeness, but we argue that it does not reliably provide the same persistence guarantees as, for example, Intel's eADR technology [41] does in a single machine setting.

Using a `GPF` in a weakly ordered memory model exposes the system to potential inconsistencies since the order by which writes reach persistent media might still differ from the program order. In this case, a value of a write operation could already reside in a cache while an earlier write by the same thread still resides in the processor's internal buffers. The persistent state is guaranteed to be consistent only if the earlier write's effects are applied to the persistent state before the later write's. This cannot be guaranteed solely by using persistent caches or `GPF`, in presence of volatile buffers in processors that do not provide total store order (TSO). `GPF` is intended to be used for a planned shutdown or reboot of the system. Therefore, we do not expect it to be used in application level algorithms. However, a carefully designed algorithm may still employ `GPF` for snapshots, thanks to its global and blocking properties.

Standard cache replacement policies introduce non-deterministic (from the perspective of the program) evictions of cache lines from a cache to the next memory hierarchy. This is not driven by any instruction and is considered normal behavior by the CXL specification. These non-deterministic steps are captured by the dotted lines: propagation is possible from a local cache to a remote cache or local memory. Propagation from local cache to remote memory is achieved by these two steps taking place one after the other.

Limitations of CXL. Under the current specification, we identify three key limitations related to concurrent programming. First, the CXL specification lacks ordering primitives. It defines neither global ordering primitives (like the `__threadfence_system` call in CUDA [58], which is a GPU-wide fence), nor local ones (like x86's `MFENCE` [43]). Consequently, our model does not have memory fences. Second, the CXL specification only specifies synchronous flushes, which take effect immediately, and does not provide asynchronous flush instructions, whose effect is delayed until a corresponding barrier. Such instructions exist on multicore architectures: `CLFLUSHOPT` and `CLWB` followed by an `SFENCE` barrier on x86, and `DC.CVAP` followed by a `DSB.SY` barrier on ARM. These allow fine-tuned efficient implementations, allowing programmers to mark certain stores that are needed to persist, and later make sure they all persist by one fence. Adding such features to CXL0 can be done along the lines of [46, 62] using an additional layer of partially ordered persistency buffers, or following [19] with view-based semantics.

3.3 Formal model

We now introduce CXL0, the first programming model for CXL. We formally model the system as a labeled transition system (LTS). We assume N machines communicating through CXL. Each machine emits actions in an order that is determined by the program order and the memory model of its local compute unit. The set of memory locations on machine i is denoted by Loc_i, and we assume that $Loc_1, \ldots, Loc_N$ are pairwise disjoint. We let $Loc = \cup_i Loc_i$ denote the set of all shared locations in the system. Values are taken from a set Val, which we assume to include a distinguished value, denoted by 0, used for the initial value of every location. The transition labels in CXL0 consist of:

- Labels for actions emitted by the networked machines: $\mathtt{LStore}_i(x, v)$, $\mathtt{RStore}_i(x, v)$, $\mathtt{MStore}_i(x, v)$, $\mathtt{Load}_i(x, v)$, $\mathtt{LFlush}_i(x)$, $\mathtt{RFlush}_i(x)$, $\mathtt{GPF}_i$, and $\mathtt{RMW}_i(x, old, new)$ where $x \in Loc$ and $v, old, new \in Val$.
- Silent non-deterministic internal propagation τ.
- Non-deterministic local crash $\lightning_i$ where $i \in \{1, \ldots, N\}$.

In turn, CXL0 system states are pairs $\gamma = (C, M)$, where:

- C maps each machine i to its *local cache* $C_i : Loc \to Val \uplus \{\bot\}$.
- M maps each machine i to its *local memory* $M_i : Loc_i \to Val$.

For every machine i, C_i and M_i are functions from the address space to possible values. C_i may additionally receive the invalid value, denoted by $\bot$. M_i maps addresses belonging to machine i to valid values, and represents the values as they are stored in its physical memory. For brevity, we assume that for each i, M_i is either *volatile* or *non-volatile*, which

determines whether or not it loses its contents upon crash. Expanding the model to support machines with both volatile and non-volatile memory is simple using sub-indices.

The abstract components "cache" and "memory" mentioned above capture how far the latest value of an address propagated towards physical memory in the memory hierarchy. Although we refer to them as "cache" and "memory", they do *not* directly correspond to the hardware primitives with the same names. Our model merely captures how the effects of write operations on shared memory propagate and become persistent. While the CXL standard presents both cacheable and non-cacheable write transactions, CXL0 is agnostic about the local cacheability of reads and writes. It assumes the underlying cache coherence and does not provide any means to reason about it.

The following invariant holds throughout executions:

$$\forall i, j \in \{1, \ldots, N\}. \forall x \in Loc.$$
$$C_i(x) \neq \bot \wedge C_j(x) \neq \bot \implies C_i(x) = C_j(x)$$

This invariant asserts that only one unique valid value may be stored for a given logical address in all caches across the system. Roughly speaking, this is similar to a cache line being in a SHARED state for some coherence protocol. However, as noted above, our model does not aim to provide a precise description of CXL's coherence state machine. Instead, we abstract it away, only describing the behavior of concurrent program executions. It is possible to have a configuration in which the value of a location x in caches differs from the value in the memory of the machine that owns x. This is necessary to model the fact that writes become globally visible before they are persistent. In such cases, the cache value is always more recent.

Initially, all machines start with empty caches ($C_i = \lambda x. \bot$, meaning every value of x is mapped to $\bot$) and zero-initialized memories ($M_i = \lambda x. 0$). Figure 2 presents the operational semantics of the system.

Store steps. For a location x that belongs to machine k, $\texttt{LStore}_i(x, v)$ sets x to v in C_i; $\texttt{RStore}_i(x, v)$ sets x to v in C_k; and $\texttt{MStore}_i(x, v)$ sets x to v in M_k. All these steps also invalidate x in all other caches to make sure the previous value is not available anywhere.

Load steps. If a value exists in some cache, then $\texttt{Load}_i(x, v)$ takes its value v from there. The global invariant ensures that valid values in all caches are the same. In addition, a load from cache copies the value to the cache of machine i. The latter ensures the correct behavior of a future LFlush operation. If no cache has a value for x, the value is loaded from the memory that contains address x.

Propagation steps. The system propagates values non-deterministically between caches ("horizontal propagation") and from cache to memory ("vertical propagation"). Cache-to-cache propagation of a value for address x from machine i is always toward the machine that owns x and removes the value from the machine i's cache. Cache-to-memory propagation is always from the owner's cache to the owner's memory and removes the value from all caches.

Flush steps. Intuitively, $\texttt{LFlush}_i(x)$ forces the horizontal propagation of x's value from C_i to the cache of its owner. When performed by the owner of x, $\texttt{LFlush}_i(x)$ forces vertical propagation to memory. In the presence of non-deterministic propagation steps, we achieve this effect by ensuring that such propagation has already occurred through a precondition ($C_i(x) = \bot$) of $\texttt{LFlush}_i(x)$. (This is similar to the modeling of MFENCE in TSO, where, instead of actively draining the store buffer, the fence simply blocks until the buffer is drained by the silent non-deterministic propagation steps [62].) Similarly, $\texttt{RFlush}_i(x)$ forces the full propagation to the owner's memory by "blocking" until the value does not exist in any of the caches. For GPF, all data that resides in caches must be propagated to the memory on which it is allocated. This step forces both horizontal and vertical propagation, and behaves like an RFlush over the entire address space. We model this step in the same blocking manner.

Crash step. At any point during execution, a machine may crash non-deterministically. This results in the machine losing the contents of its cache, and, if its memory is volatile, re-initializing the memory to zero.

Read-modify-write Operations. The CXL specification states that: "*Each host can perform arbitrary atomic operations supported by its Instruction-Set Architecture (ISA) by gaining* EXCLUSIVE *access to a cache line,* [and] *performing the atomic operation on it within its cache*" (§2.4.4 in [20]). Therefore, we model read-modify-write (RMW) operations, such as compare-and-swap (CAS) and fetch-and-add (FAA), as an atomic combination of a load followed by a store without allowing interference between the two. This is similar to how they are specified in standard operational presentation of a strongly consistent shared-memory model. Given that we have three types of store instructions, CXL0 essentially includes L-RMW, R-RMW, and M-RMW primitives.

The operational semantics of the RMW operations are a straightforward extension of the existing semantics, and for brevity, we only depict L-RMW:

$$\frac{\begin{array}{c} C_j(x) = v \neq \bot \\ C_i' = C_i[x \mapsto v'] \\ \forall j \neq i.\ C_j' = C_j[x \mapsto \bot] \end{array}}{C, M \xrightarrow{\texttt{L-RMW}_i(x,v,v')} C', M} \qquad \frac{\begin{array}{c} \forall j.\ C_j(x) = \bot \\ x \in Loc_k \qquad M_k(x) = v \\ C_i' = C_i[x \mapsto v'] \\ \forall j \neq i.\ C_j' = C_j[x \mapsto \bot] \end{array}}{C, M \xrightarrow{\texttt{L-RMW}_i(x,v,v')} C', M}$$

We remind that these steps are combinations of the load and store steps, and hence it is required to distinguish between loads from C and M, as well as the different types of store operations in CXL0. This results in a total of 6 different RMW operations. Note also that a failed RMW, for example,

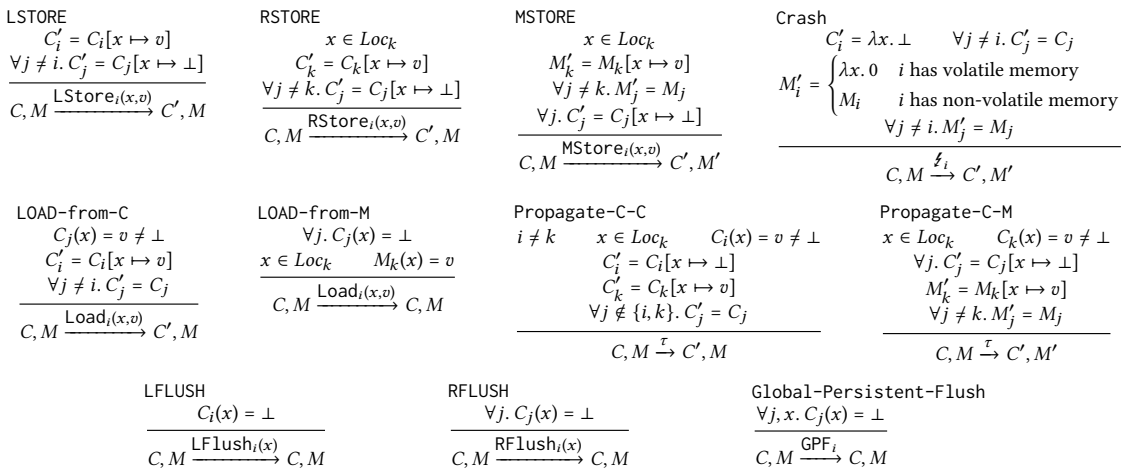


Figure 2. CXL0: Operational Semantics (Load, Store, Crash, and Flush Primitives)

a failed CAS, is equivalent to a plain read. Therefore, the citation from the CXL specification describes only our L-RMW, but implementing the other RMW operations is possible if the cache line is kept in Exclusive or Modified state until the post-conditions for the remote store are met.

We note that the intricacies of the model arise from the need to handle node crashes. Without crashes, CXL0 has simple, sequentially consistent semantics. Every load reads the last written value to the same location, flush instructions are benign, and all store instructions are equivalent and executed in program order.

3.4 Intuition

Before demonstrating the behaviors of CXL0 in a few examples, we highlight several useful (and expected) implications. We write $\gamma \xrightarrow{\alpha_1 \dots \alpha_n} \gamma'$ if there is a sequence of transitions labeled $\alpha_1, \dots, \alpha_n$, possibly interleaved with additional silent τ-steps, starting in state γ and ending in state γ'.

Proposition 1. *The following hold assuming $x \in Loc_k$; $j \neq k$:*

1. RStore *is stronger than* LStore*:*
   If $\gamma \xrightarrow{\mathsf{RStore}_i(x,v)} \gamma'$, *then* $\gamma \xrightarrow{\mathsf{LStore}_i(x,v)} \gamma'$.
2. RStore *and* LStore *by the owner are equivalent:*
   If $\gamma \xrightarrow{\mathsf{LStore}_k(x,v)} \gamma'$, *then* $\gamma \xrightarrow{\mathsf{RStore}_k(x,v)} \gamma'$.
3. MStore *is stronger than* RStore*:*
   If $\gamma \xrightarrow{\mathsf{MStore}_i(x,v)} \gamma'$, *then* $\gamma \xrightarrow{\mathsf{RStore}_i(x,v)} \gamma'$.
4. RFlush *is stronger than* LFlush*:*
   If $\gamma \xrightarrow{\mathsf{RFlush}_i(x)} \gamma'$, *then* $\gamma \xrightarrow{\mathsf{LFlush}_i(x)} \gamma'$.
5. LFlush *after* RStore *by non-owner is redundant:*
   If $\gamma \xrightarrow{\mathsf{RStore}_j(x,v)} \gamma'$, *then* $\gamma \xrightarrow{\mathsf{RStore}_j(x,v)\cdot\mathsf{LFlush}_j(x)} \gamma'$.
6. RFlush *after* MStore *is redundant:*
   If $\gamma \xrightarrow{\mathsf{MStore}_i(x,v)} \gamma'$, *then* $\gamma \xrightarrow{\mathsf{MStore}_i(x,v)\cdot\mathsf{RFlush}_i(x)} \gamma'$.
7. RStore *by non-owner is simulated by* LStore + LFlush*:*
   If $\gamma \xrightarrow{\mathsf{LStore}_j(x,v)\cdot\mathsf{LFlush}_j(x)} \gamma'$, *then* $\gamma \xrightarrow{\mathsf{RStore}_j(x,v)} \gamma'$.
8. MStore *is simulated by* LStore + RFlush*:*
   If $\gamma \xrightarrow{\mathsf{LStore}_i(x,v)\cdot\mathsf{RFlush}_i(x)} \gamma'$, *then* $\gamma \xrightarrow{\mathsf{MStore}_i(x,v)} \gamma'$.

To establish confidence, we have formulated and proved Proposition 1 in Rocq. The Rocq files are available in [9]. Most of the arguments are straightforward. The full version of the paper [8] contains proofs for (7) and (8), which are more difficult to prove.

Litmus tests. Next, we discuss possible and impossible behaviors under the CXL0 model through a set of litmus tests. We find that some of the possible behaviors are undesirable and raise consistency issues in the face of a crash. Figure 3 depicts the litmus tests as sequences of events and operations as they are performed by the local machine in *execution order*. We assume that the execution order may differ from the program order, as is common in commodity CPUs. We denote memory allocated on machine i as x^i, and we assume that all memory in the following tests is non-volatile. Note that $\mathsf{RStore}_2(y^1, x^2)$ is a shorthand for reading x^2 into a local register and then performing an RStore to y^1.

We present the litmus tests as traces of CXL0 primitives. Although the CXL specification allows transactions operating on different addresses to execute concurrently, we serialize them in some possible order for simplicity. We use a sequential presentation to determine the behavior of retired instructions, after they have been internally ordered by the processors executing them. This also enables us to separate the discussion from any processor-specific memory model. Allowed behaviors are marked with ✔, and illegal behaviors are marked with ✗. We present scenarios that highlight the similarities and key differences between CXL0 and previous models that consider persistent memory.

$\texttt{RStore}_1(x^1, 1);\ \lightning_1;\ \texttt{Load}_1(x^1, 0)$ ✔ (1)

$\texttt{MStore}_1(x^1, 1);\ \lightning_1;\ \texttt{Load}_1(x^1, 0)$ ✗ (2)

$\texttt{LStore}_1(x^1, 1);\ \texttt{LFlush}_1(x^1);\ \lightning_1;\ \texttt{Load}_1(x^1, 0)$ ✗ (3)

$\texttt{LStore}_1(x^2, 1);\ \texttt{LFlush}_1(x^2);\ \lightning_2;\ \texttt{Load}_1(x^2, 0)$ ✔ (4)

$\texttt{LStore}_1(x^2, 1);\ \texttt{RFlush}_1(x^2);\ \lightning_2;\ \texttt{Load}_1(x^2, 0)$ ✗ (5)

$\texttt{LStore}_1(x^3, 1);\ \texttt{Load}_2(x^3, 1);\ \lightning_1;\ \texttt{Load}_2(x^3, 0)$ ✗ (6)

$\texttt{LStore}_1(x^3, 1);\ \texttt{Load}_2(x^3, 1);\ \texttt{LFlush}_2(x^3);\ \lightning_1;\ \lightning_2;\ \texttt{Load}_2(x^3, 0)$ ✗ (7)

$\texttt{RStore}_1(x^2, 1);\ \texttt{RStore}_2(y^1, x^2);\ \lightning_2;\ \texttt{Load}_1(y^1, 1);\ \texttt{Load}_1(x^2, 0)$ ✔ (8)

$\texttt{MStore}_1(x^2, 1);\ \texttt{RStore}_2(y^1, x^2);\ \lightning_2;\ \texttt{Load}_1(y^1, 1);\ \texttt{Load}_1(x^2, 0)$ ✗ (9)

Figure 3. Litmus tests for CXL0

Tests 1-3 span a single machine. In the first test, the model permits the loss of a stored value upon crashing when using `RStore` because it does not guarantee that the value has propagated to persistence before the crash. However, such behavior is impossible in test 2 thanks to `MStore`, which guarantees the persistence of the update before it returns. Test 3 shows that a value cannot be lost if the store has been flushed (*i.e.*, propagated) to the local persistent memory using `LFlush` before the crash.

Tests 4-7 involve multiple machines. In Test 4, a stored value may be lost due to a crash if it has not reached remote persistent memory. Test 5 uses the stronger `RFlush` which requires the value to propagate further (by requiring $\forall j.\, C_j = \bot$), thus preventing the loss of the stored value. Tests 6 and 7 demonstrate that copying a value to the local cache upon loading helps prevent the loss of stored values when a machine performing an `LStore` to remote memory crashes. A load operation reads the value in Test 6 from C_2 after the crash, and in Test 7 from C_3 after the crash, thanks to the flush operation by machine 2.

The last two tests concern writes to multiple variables. Test 8 shows that it is possible to lose a stored value that another operation had already observed. Specifically, a possible recovered state would include the effects of a later operation without the first. In that case, using `MStore` for the first write would make it impossible to obtain an inconsistent state upon recovery of the second machine, as seen in Test 9.

3.5 Model Variants

We describe two possible variants of our model to capture potential hardware implementation alternatives.

Crash with cache line poisoning. Cache line poisoning marks a line as unusable after corruption (*e.g.*, uncorrectable ECC errors) to prevent access to it. CXL supports cache line poisoning upon crash of a machine. In the event of a surprise hot unplug, link down, or transaction timeout where the device is unresponsive, the system uses CXL Isolation (§9.9, §12.3 in [20]). A Non-Existent Memory (MemData-NXM) response is generated with the poison bit set for read requests targeting the unreachable memory addresses, if the Poison on Decode Error capability is enabled (§8.2.4, Table 8-117 in [20]). To capture this behavior, we provide the following alternative `Crash` step:

`Crash` (PSN)

$$\frac{\begin{array}{c} C_i' = \lambda x.\, \bot \\ \forall j \neq i, x \notin Loc_i.\ C_j'(x) = C_j(x) \qquad \forall j \neq i, x \in Loc_i.\ C_j'(x) = \bot \\ M_i' = \begin{cases} \lambda x.\, 0 & i \text{ has volatile memory} \\ M_i & i \text{ has non-volatile memory} \end{cases} \qquad \forall j \neq i.\, M_j' = M_j \end{array}}{C, M \xrightarrow{\lightning_i} C', M'}$$

After a crash, the crashed machine's addresses are mapped to $\bot$ in all caches. We call this version of the model CXL0^{PSN}.

Remote loads with implicit write-back. Load steps in CXL0 are deterministic. We do not model details like hardware write-back responsibility because the CXL specification does not explicitly require remote loads to update memory. However, if hardware turns out to implement such a policy, this would affect observable persistence and we can update our semantics to reflect this behavior. To capture implicit write-backs for remote loads, *i.e.*, every change is written to main memory before being propagated to another cache, we may replace our `LOAD-from-C` step by the following:

$$\texttt{LOAD-from-C (LWB)}\ \frac{C_i(x) = v \neq \bot}{C, M \xrightarrow{\texttt{Load}_i(x,v)} C, M}$$

Intuitively, this forces that all loads for non-local addresses are served from memory. We use the same trick from before to force the full propagation to the owner's memory by "blocking" until the value does not exist in any of the caches. We call this version of the model CXL0^{LWB}.

Comparison of the different variants. Every trace allowed by the above variants is also allowed by CXL0. We encode all model variants as communicating sequential processes, and use the FDR4 refinement checker [35] to obtain examples showing traces of CXL0 that are not allowed by the variants, as well as examples showing that the two variants are incomparable.

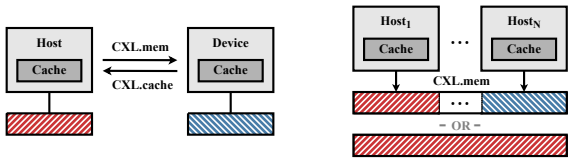
For that matter, consider two machines: machine 1 with NVMM and machine 2 with volatile memory. We denote the locations on machine i as x^i and report the validity (✔/✗) as a triple (CXL0, CXL0^{LWB}, CXL0^{PSN}):

$\texttt{RStore}_2(x^1, 1);\ \texttt{Load}_2(x^1, 1);\ \lightning_1;\ \texttt{Load}_2(x^1, 0)$ ✔,✗,✔ (10)

$\texttt{LStore}_1(x^1, 1);\ \texttt{Load}_2(x^1, 1);\ \lightning_1;\ \texttt{Load}_1(x^1, 0)$ ✔,✗,✔ (11)

$\texttt{LStore}_2(x^1, 1);\ \lightning_1;\ \texttt{Load}_1(x^1, 1);\ \lightning_1;\ \texttt{Load}_2(x^1, 0)$ ✔,✔,✗ (12)

In Test 10, machine 2 updates x in the cache of machine 1 and then loads x into its own cache. After a crash of machine 1, machine 2 reads the old value of x under both CXL0 and CXL0^{PSN}. The same happens when the initial `RStore` of machine 2 is replaced by an `LStore` of machine 1 in Test 11. This happens because both CXL0 and CXL0^{PSN} allow updates issued by `LStore` or `RStore` to be lost, as in both models loads



(a) Host-Device Pairing **(b)** Memory Pool

Figure 4. Current System Models with CXL

may be served from the cache before the update becomes persistent. In particular, the value may still be propagated to the cache of machine 1 after the load. Tests 10 and 11 illustrate that CXL0$^{\text{LWB}}$ is the most resilient model with respect to losing updates.

In Test 12, machine 2 updates x in its own cache, after which machine 1 crashes. Subsequently, machine 1 loads the latest value of x and then crashes again. Finally, machine 2 loads the old value of x. This test shows that CXL0$^{\text{PSN}}$ prevents inconsistencies across consecutive crashes. In contrast, both CXL0 and CXL0$^{\text{LWB}}$ allow such behavior, as remotely cached updates must propagate back to memory through a set of volatile caches which are susceptible to crashes. Poison semantics undercut this chain by poisoning of cache entries immediately upon the first crash. Note that CXL0$^{\text{LWB}}$ only writes back data to memory in the event of a remote load. Therefore, $Load_1(x^1, 1)$ does not guarantee the data reaches main memory before the load finishes.

4 System Model Variations

As with any new standard, the adoption of CXL will be gradual as the hardware, software, and standard evolve. In this section, we present a roadmap of viable system configurations based on discussions with experts from CXL vendors and on existing setups. We characterize each configuration and demonstrate CXL0's applicability to each.

Host-device pair. We begin with a simple setting consisting of a host and an accelerator that share memory in a cache-coherent manner using CXL.cache and CXL.mem. It is depicted in Fig. 4a. The host manages its own memory's coherence, while device memory coherence is either host-managed or bypassed, depending on the bias mode. If memory is volatile, failure of the host or device may cause the loss of the attached memory's content. This is the first step in the evolution of CXL deployments, as this single-machine setup has already been deployed [45, 61].

CXL0 applies to this setting with a few restrictions. Both the host and the device are modeled as machines with their own cache and memory. According to the CXL specification, the host can issue all available CXL0 primitives apart from `RStore`, `LFlush` and remote RMWs (`R-RMW` and `M-RMW`). The device can issue all stores, including `RStore`, but cannot issue `LFlush` and remote RMWs. Note that CXL0 applies to device memory in host-bias mode.

Partitioned disaggregated memory pool. Next, we discuss a setting in which multiple hosts access disjoint parts of a memory pool for memory extension. It is depicted in Fig. 4b. Memory is only shared between threads per host, therefore we only have intra-host memory coherence. CXL.mem is the key protocol in use. This realization of a disaggregated memory pool via CXL already exists [52, 76]. It is conceptually similar to having a set of isolated machines with NVMM. The memory pool acts as an external failure domain to the hosts. Machine crashes do not affect the contents of the pool.

CXL0 applies to this setting with some restrictions. The disjoint memory partitions are modeled as N additional memory nodes. Further, there is a bijection from compute nodes to memory nodes, and no other memory is shared in the system. Per the CXL specification, we apply the following restrictions to the available CXL0 primitives. We exclude `RStore`, `LOAD-from-C`, `Propagate-C-C`, and remote `RMW`s, as there is no interaction between hosts. Additionally, we point out that `LFlush` and `RFlush` are semantically equivalent in this setting. The remaining CXL0 primitives suffice to reason about any concurrent program execution interleaved with a crash.

Shared disaggregated memory pool. Finally, we discuss a globally shared memory pool. It is depicted in Fig. 4b. We identify a gap between the CXL specification and available hardware. Currently, there is no CPU or pool device that implements CXL 4.0 back invalidation flows, so cache-coherent sharing is unavailable [77]. Therefore, we distinguish between a non-coherent, realistic memory pool and a cache-coherent pool envisioned according to the specification.

CXL0 does not apply to the non-coherent setting, because the central assumption of cache coherence is not satisfied. This means that concurrent program executions would violate CXL0's global cache invariant. Bypassing caches, *i.e.*, only allowing the CXL0 primitives `MStore`, `LOAD-from-M`, and `M-RMW`, retains correctness. However, this would likely result in a massive performance loss. Compensating for the lack of hardware guarantees with software is an interesting open research question [14, 53], particularly in the face of potentially partially coherent pools. We note that, while CXL0 assumes cache coherence, it is agnostic about whether this is implemented by hardware or software. If a suitable software protocol exists with a similar load/store interface and coherence guarantees, then CXL0 applies to it as well.

Naturally, CXL0 applies to the fully cache-coherent version. The memory pool is modeled as an extra memory node. Per the CXL specification, interactions with remote caches and remote RMWs are unavailable, so `RStore`, `LOAD-from-C`, `LFlush`, `Propagate-C-C`, and remote `RMW`s are excluded.

Future configurations. We see that current CXL configurations are limited compared to the general operational semantics of CXL0. However, we have ensured that CXL0 captures the evolving nature of CXL configurations up to

coherent memory sharing and fully symmetric capabilities between hosts and devices.

5 Practical Implications

Next, we conduct a set of experiments to evaluate the practical implications of the CXL0 model on real hardware. The goal is to determine how concrete CXL transactions map to CXL0 primitives, and to measure the latency of individual CXL0 primitives in isolation.

Our setup consists of an x86 CPU (host) and an FPGA (device), both running CXL 1.1, with a protocol analyzer connected between them. The CPU accesses FPGA memory via CXL.mem, and the FPGA accesses CPU memory via CXL.cache. Therefore, both types of memory are shared and coherent. On the FPGA side, we use a proprietary CXL IP by Intel to configure it as a CXL Type 2 device. Besides the IP's default behavior, we can add custom logic between the FPGA memory and the CXL IP. This allows us to manipulate the CXL traffic by responding to requests or sending new ones according to our needs. The analyzer is a *Teledyne LeCroy's T516 Protocol Analyzer* for PCIe 5.0 and CXL, which supports all CXL sub-protocols and device types.

This setup corresponds to the first variation depicted in Fig. 4. We generate and observe both CXL.mem and CXL.cache transactions, and cover all of CXL0's primitives in the evaluation. As noted in §4, all other settings provide, at most, the same set of primitives.

5.1 Mapping the CXL specification to CXL0

Table 1 depicts the mapping from CXL transactions to abstract CXL0 primitives. As host requests to cache device memory are handled by CXL.mem, and device requests to cache host memory are handled by CXL.cache, we analyze CXL0 from the perspectives of both the CPU and the FPGA. The CPU is denoted as host and the the FPGA as device in the table. We distinguish between the types of memory to which the target address belongs: *Host-attached Memory (HM)* or *Host-managed Device Memory (HDM)*. We then create all possible pairs of cache coherence states for the host and the device, and all possible variations of available operations. We use the CXL analyzer to observe the CXL transactions that are generated on the link.

We find that there is a many-to-one mapping from CXL.cache and CXL.mem transactions to the abstract primitives captured in CXL0. For example, the first row of Table 1 represents a CXL0 `Read` primitive issued by the host CPU, *e.g.*, a normal `load`. We denote the cache coherence states of the host and the device before executing the CXL0 primitives as $(Host, Device)$ where $Host, Device \in \{M, E, S, I\}$ and $*$ indicates any of the possible MESI states. If it targets HM, then two possible CXL transactions can be triggered. If $(Host, Device)$ is $(*, I)$ then no CXL transaction is observed (None). Else, $(Host, Device)$ is one of (S, S), (I, S), (I, E), (I, M),

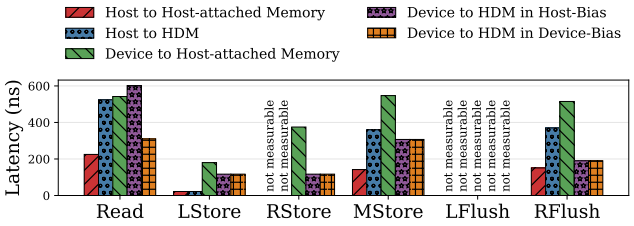


Figure 5. Latency of CXL0 primitives on host and device.

and we observe a CXL.cache Host-to-Device (H2D) SnpInv request. Similarly, when a CXL0 `Read` primitive targets HDM, we observe two possible CXL transactions, again. If $(Host, Device)$ is $(I, *)$, a CXL.mem Master-to-Subordinate (M2S) MemRdData request occurs. If $(Host, Device)$ is (S, S), (S, I), (E, I) or (M, I), no CXL transaction is observed (None).

In addition, not all CXL0 primitives are available (??? in Table 1). For instance, neither an `RStore` nor an `LFlush` can be generated on the CPU by a single instruction or sequence of instructions. Although the FPGA provides extra flexibility for generating CXL transactions, invoking an `LFlush` still remains impossible since the proprietary IP does not offer enough control over the underlying CXL link to issue new custom CXL transactions which are not defined by the specification. These limitations restrict the programmer's control over the underlying CXL link, rendering those primitives currently unavailable for algorithmic use. This is a critical shortcoming because the choice of CXL0 primitive makes a big difference to performance, as we will show next.

5.2 Latency Measurements

To measure the latency of individual CXL0 primitives in isolation, we make the following configuration:

- For `Load` instructions, the initial state of a cache line is INVALID in all caches.
- For `Store` instructions, full cache line writes are performed.
- To measure latency from the CPU side, we use LATTester [74].
- To measure latency from the FPGA side, we implement custom logic. To execute a new memory load/store request using CXL, we issue an AXI-bus request to the CXL IP. Upon completion, the CXL IP responds via the designated read/write-response channels of the AXI-bus. Counting the number of FPGA clock cycles that elapse between the two events allows us to determine the latency of a CXL0 primitive on the device.

Figure 5 depicts latency measurements for various CXL0 primitives. We characterize five types of accesses. Host accesses to (1) Host-attached Memory (local) or (2) Host-managed Device Memory (HDM, remote), and device accesses to (3) Host-attached Memory (remote), (4) HDM in host-bias (local, but requires permission from the host) or (5) HDM in device-bias (local). We report the median over 1000 measurements of sequential memory accesses.

CXL0 primitive	Node	Operation	to HM	to HDM in Host-Bias
		x86 Instructions	**CXL.cache H2D**	**CXL.mem M2S**
Read	Host	Load	None, SnpInv	None, MemRdData
LStore		Store	None, SnpInv	None, MemRdData, MemRd
RStore		???	???	???
MStore		Non-Temporal Store + Fence	SnpInv	MemWr
LFlush		???	???	???
RFlush		CLFlush	None, SnpInv	None, MemInv, MemWr
		Device Operation	**CXL.cache D2H**	**CXL.cache & CXL.mem**
Read	Device	Caching Read	None, RdShared	None, RdShared
LStore		Caching Write	None, RdOwn	None, RdOwn
RStore		HM: ItoMWr / HDM: Caching Write	ItoMWr	None, RdOwn
MStore		Caching Write + CLFlush	(RdOwn +) DirtyEvict, WOWrInv/F, WrInv	None, MemRd
LFlush		???	???	???
RFlush		CLFlush	CleanEvict, DirtyEvict	None, MemRd

Table 1. Observable CXL transactions for all possible CXL0 primitives.

We observe that loads and stores to local memory are twice as fast as those to remote memory for Read and MStore primitives. For the CPU, it is 2.34x faster, and for the device, it is 1.94x faster (see *red* vs. *blue* and *orange* vs. *green* bars). These results are consistent with previous findings [25, 51]. Furthermore, accesses from the host and the device to their respective remote CXL memory yield the same latency, despite using different CXL sub-protocols.

We observe our predicted latency trends for store operations. LStore is unsurprisingly fast. However, there is a difference between the LStore writes on the CPU and FPGA (*red* and *blue* vs. other LStores). The CPU takes advantage of its write buffers, while the device employs a single level of caches and no specialized hardware. LStore primitives issued by the FPGA also differ, with slower local cache writes to Host-attached Memory (*green*) than to HDM (*purple* and *orange*). This is due to the CXL IP using two separate caches of different size, depending on the memory target. When a device writes to Host-attached Memory (*green* bars), MStore is 1.45x slower than RStore, which is, in turn, 2.08x slower than LStore. These differences are expected, as access to more distant memory incurs higher latency, and this trend should persist in subsequent CXL versions. RFlush is the only type of flush that both the CPU and the FPGA can invoke. The measured latencies are nearly identical to those of the MStore primitive.

To summarize, we provide a mapping from CXL transactions to CXL0 primitives, and find that the CXL specification currently does *not* support all transactions captured in CXL0, such as RStore from the host-side or LFlush from either side. However, we also show that the choice of CXL0 primitive *significantly* affects performance. Therefore, we advocate to extend the specification and expose precise ISA-level controls, similar to flushes in x86, as a larger set of primitives gives the programmer finer options for data placement and expands the design space for correctness and performance.

6 General Transformations

Ensuring correct execution of concurrent programs under spontaneous failures is a challenging task [17, 29, 48, 70]. To alleviate the programmer's burden, we present a transformation that provides concurrent algorithms with fault tolerance guarantees under CXL0. Our transformation is an adaptation of FliT by Wei et al. [72]. We show that with minor adjustments, FliT can be applied to a broader range of use cases than originally intended. This transformation equips *any* linearizable object with durable linearizability under CXL0. First, we present a simple example that clearly illustrates the need for a new transformation. Then, we define the desired correctness guarantees in the CXL0 model. Next, we provide an introduction to FliT and then discuss our adaptation of it. The correctness proof of our transformation is given in the full version of this work [8]. Throughout this section, we assume that all shared memory is either non-volatile or in a different failure domain. In addition, we do not assume a specific memory model. All of the presented algorithms list instructions in *program order*, so additional architecture-specific memory fences may be necessary (*e.g.*, MFENCE for x86, and DMB for ARMv8) when implementing them on real hardware.

6.1 Motivating example

To emphasize the difference between the single-machine setting modeled in [44] and the distributed setting with CXL, we present a simple litmus test.

In the program below, $r1$ and $r2$ are local registers and x is a shared variable initialized to 0. The program runs on machine M1 and x is allocated on machine M2 ($x \in Loc_{M2}$).

```
x=1;    r1=x;    r2=x;    assert(r1==r2);  ✗   (13)
```

We expect $r1$ and $r2$ to be equal, but our system model allows for different behaviors, *e.g.*, if M2 crashes and recovers between the execution of the two load steps. When M1 updates x, the update resides in its cache, then propagates to M2's cache, and finally to M2's memory, where it persists. If M2 crashes after the update has left M1's cache but before it reaches M2's memory, the update is lost. Thus, a remote machine's crash can affect the correctness of a local program, something that doesn't occur in single-machine settings.

The same would hold true if a flush which evicts x from M1's cache is placed between the first two commands, or if an additional flush is placed between the reads of x. In the full-system crash model, this inconsistency is impossible, and either of the aforementioned flushes would preserve the persistence ordering of future reads with respect to the update of x. However, our system model and the CXL0 semantics require a flush which ensures that the update reaches physical memory in order to guarantee that the assertion is always satisfied.

To conclude, the partial-system crash model introduces consistency challenges, and there is a need to distinguish between flush operations with different post-conditions.

6.2 Correctness Guarantees

Our transformation provides *durable linearizability*, a correctness criterion presented by Izraelevitz et al. [44] in a full-system crash model. We discuss the applicability of the definition to our partial-system crash model, review the key steps in the definition's formulation and show that it applies to CXL0's system model as is.

The abstract model by Izraelevitz et al. [44] treats high-level objects and considers the invocation and response of operations on these objects as events in the system's history. In our model, we consider partial-system crashes, e.g., single-machine failures, as events as well. Machines may accommodate multiple running threads that crash together and instantaneously. As in [44], we assume that, when recovering from a crash, new threads with new and distinct identifiers are spawned to replace the ones that operated prior to the crash. However, we limit this to the crashed machine, i.e., threads running on other machines are uninterrupted. Simultaneous failures of any combination of machines is captured in CXL0 by appending local crashes one after the other, in any order.

We define an *abstract history* as a sequence of invocation, response, and single machine crash events. All the instructions that compose into a high-level operation take place between its invocation and response. We keep the definition of abstract well formedness as originally defined in [44]. A *well formed history* is a history in which the local history of every thread is a properly interleaved sequence of invocation and response events, possibly ending with a local crash.

A history is *durably linearizable* if it is well formed, and linearizable after all crash events are removed from it. Linearizability itself is defined using the *abstract happens-before relation* between events, as described in [40, 65], which is the partial order obtained by restricting the real-time order in a history to consist of the order between events of the same thread, as well as the order from a response event to an invocation event. We remind that the definition of linearizability allows both completing or omitting pending invocations without matching responses.

While Izraelevitz et al. [44] redefine the "abstract happens-before relation" so it orders events also with respect to crashes, we observe that this is unnecessary, as the history checked for linearizability is a crash-free history obtained from the original history by removing all the crash events. Thus, there is no need to modify the relation, and we keep the original definition of [40, 65] as is. Therefore, the definition of durable linearizability is agnostic about the scope of a crash and applies to our model without modification. From this point on, we use the term "durably linearizable" in the context of CXL0's system model and failure model, which captures crashes at single machine granularity.

We recall that linearizability and durable linearizability are local, composable properties. In other words, combining (durably) linearizable objects yields (durably) linearizable histories. This locality has immense practical implications, as objects designed to satisfy some local property can be used together without further reasoning or adaptations, and the resulting executions will also satisfy that property.

6.3 Introduction to FliT

We seek to devise a general and correct transformation that satisfies durable linearizability and chose to adapt FliT by Wei et al. [72]. FliT takes linearizable objects and makes them durably linearizable in the full-system crash model. Among existing transformations that extend objects with durable linearizability in the single-machine, full-system crash model (§7), FliT is general as it only requires the original object to be linearizable. Following prior FliT proofs [13, 72], we separate liveness from safety. Our work focuses exclusively on durable linearizability, which is a safety property, and leaves liveness to the programmer. Therefore, just like original FliT, we assume the original data structure to be non-blocking to trivially guarantee liveness.

Due to the locality of durable linearizability, it is possible to transform every object independently to ensure that an

Alg. 1. The Original FliT Transformation (for x86)

```
1  T private_load(T* X) {
2      return X->Load();
3  }
4  void private_store(T* X, T args, bool pflag) {
5      if (pflag) {
6          X->Store(args);
7          Flush(X);
8          MFENCE();
9      }
10     else X->Store(args);
11 }
12 T shared_load(T* X, bool pflag) {
13     T val = X->Load();
14     if (pflag && flit_counter(X) > 0) Flush(X);
15     return val;
16 }
17 void shared_store(T* X, T args, bool pflag) {
18     FENCE();
19     if (pflag) {
20         flit_counter(X).fetch&add(1);
21         X->Store(args);
22         Flush(X);
23         MFENCE();
24         flit_counter(X).fetch&sub(1);
25     }
26     else X->Store(args);
27 }
28 void completeOp() {
29     MFENCE();
30 }
```

algorithm spanning multiple objects inherits its correctness from the underlying objects. It is important to note that FliT transforms objects that are *already* linearizable, *i.e.*, synchronization over high-level objects is handled at the object level. Consequently, FliT does not introduce extra concurrency control, nor does our adaptation.

At the core of FliT is a classification of memory accesses. FliT distinguishes between shared and private memory operations. Private memory operations operate on data that will never be accessed by more than one thread concurrently. Examples include thread-local counters and logs. Conversely, shared operations span shared data that may be accessed concurrently by multiple threads. Furthermore, FliT also differentiates between operations that need to become persistent and those that do not. To avoid blindly persisting every memory write, FliT uses a persistence flag (`pflag`) to tag only the necessary operations. This flag indicates whether the address is in persistent memory, and whether updates to that address should be durably linearizable. Only tagged operations are handled with the proper flush instructions.

The original version of the FliT transformation assumes execution on an x86 processor and is depicted in Alg. 1. With x86, FliT assumes the total-store-order memory model (TSO), and that a `Flush` corresponds to an optimized asynchronous flush instruction (*e.g.*, `CLFLUSHOPT`). FliT uses `SFENCE`, which acts as a memory barrier that no store operation can cross in any direction. Prior flush operations must be completed before a thread can perform store instructions after this barrier.

The `private_load`, `private_store`, `shared_load`, and `shared_store` methods serve as wrappers for memory accesses that provide persistence and durable linearizability with the help of `completeOp`. The `private_load` method performs a load of the specified object. If flagged as persistent, a `private_store` of an object stores it in persistent memory by performing a `Store` operation (Line 6) followed by a `Flush` to write the value to memory. Otherwise, it performs a `Store` (Line 10), which has no guarantees with respect to persistence, so its propagation to memory may be reordered with those of later stores. The `shared_load` and `shared_store` methods satisfy a guarantee stronger than just persistence of flagged operations: durable linearizability. Because a store operation is not immediately persistent after execution, it may become globally visible without being guaranteed to survive a crash. FliT ensures that, when a value is read, it is already persistent by the time the high-level operation containing that read completes and before any subsequent `shared_store`.

To avoid naïvely flushing every location upon reading, FliT uses a shared counter for each shared object, called the *FliT counter*. This counter signals to readers that a store is being performed, but it is not necessarily persistent yet. Shared store operations increment the counter (Line 20) before applying the store to shared memory and persisting it (Lines 22 and 23), and decrement it after the store persists (*i.e.*, after a corresponding full memory barrier, Line 24). A shared load operation checks if the counter's value is positive. If so, it helps to propagate the stored value by flushing it without issuing a `FENCE` instruction. The complementing `FENCE` instructions are located in the `completeOp` method and at the beginning of a `shared_store` (Line 18). FliT's methods take an additional argument (`pflag`) to distinguish between tagged and non-tagged operations. This flag indicates whether the address is in persistent memory, and whether updates to that address should be durably linearizable. If `pflag` is not set, operations do not flush the object to persistent memory.

Lastly, FliT facilitates `completeOp`, a method which guarantees that all effects of the high-level operation are persisted before the operation returns, as asserted by the definition of durable linearizability. This method consists of a single memory fence (Line 29) and guarantees that flushes executed by `shared_load` operations take effect before it returns. Write persistence is already guaranteed at the end of the store operation itself.

When using FliT, shared and private memory access methods are expected to wrap concrete memory accesses of the high-level object, and a `completeOp` is expected to be placed at the end of every high-level object operation.

6.4 Adapting FliT to CXL0

As shown in the brief example, FliT cannot be adopted as is in CXL0, due to the semantics of the store and flush instructions

Alg. 2. FliT Transformation for CXL0

```
31 T private_load(T* X) {
32     return X->Load();
33 }
34 void private_store(T* X, T args, bool pflag) {
35     if (pflag) {
36         X->LStore(args);
37         RFlush(X);
38     }
39     else X->LStore(args);
40 }
41 T shared_load(T* X, bool pflag) {
42     T val = X->Load();
43     if (pflag && flit_counter(X) > 0) RFlush(X);
44     return val;
45 }
46 void shared_store(T* X, T args, bool pflag) {
47     if (pflag) {
48         flit_counter(X).fetch&add(1);
49         X->LStore(args);
50         RFlush(X);
51         flit_counter(X).fetch&sub(1);
52     }
53     else X->LStore(args);
54 }
55 void completeOp() {}
```

that it uses. In addition, FliT employs asynchronous flushes, deferring the time where reads are *guaranteed* to be flushed until the following memory fence. Hence, it is vulnerable and must be adapted for partial-system crashes.

Our transformation is designed for high-level code. We assume support for CXL0's load, store, and flush primitives in the form of respective high-level instructions or methods. We also assume that there exists an atomic fetch-and-add (`FAA`) operation, and remind that CXL0 captures the behavior of `RMW` operations as part of its operational semantics. Unlike the original FliT, we do not assume a fixed memory model.

Alg. 2 depicts our adaptation of FliT. All `Flush` instructions are replaced with `RFlush`, and all stores, whether marked for persistence or not, use `LStore` instead of the architecture's store instruction. The choice of `RFlush` ensures that locally cached updates become persistent before the high-level operation is completed, thereby achieving durable linearizability. We mark the translated lines with a colored background. In the full version [8], we prove that our transformation yields durably linearizable objects when applied to linearizable ones.

The `completeOp` method is empty in our transformation. In the original FliT, we recall that the method comprises a single memory fence. This fence is unnecessary when assuming both in-order execution and synchronous flushes. However, when applying our transformation to real hardware with weaker memory models, Alg. 2 should employ appropriate fences to enforce correct ordering.

6.5 Implementation and performance

`LStore` and `RFlush` are available in all cache-coherent settings of the system model evolution, which highlights the applicability of this transformation to a wide variety of current and future CXL setups. Generally, it is possible to substitute one sequence of primitives for another that is semantically equivalent (see Proposition 1). For instance, replacing all stores with `MStore` automatically satisfies the desired durability guarantees, even in systems without cache coherence. However, this is expected to yield inferior performance, and even the CXL specification advises using weaker transactions for better performance (§3.5.2.3 in [20]). One way to achieve this is by exploiting additional address information. For example, `RFlush` can be replaced with `LFlush` for all memory locations owned by the writing machine. In addition, if implicit write-back on remote loads is supported system-wide, FliT counters can be kept in local memory. Our adaptation allows automatic instrumentation of the transformation based solely on the memory address. However, as with FliT, manual tuning is still required to reduce the number of flush instructions for optimal performance.

7 Related Work

A variety of work has provided formal semantics for shared-memory models that were originally specified in prose or internal engineering discussions. This includes seminal work for x86 [59, 67], other multi-core architectures [5, 31], GPU concurrency [56, 64], non-volatile memory with full-system crashes [62], and remote memory access (RMA) [24]. Our work falls within this scope. However, given that CXL 3.0 hardware is not currently available, we cannot test our model against actual implementations. Thus, our evaluation is confined to version 1.1 of the CXL standard (see §5).

Goens et al. [36] describe how specific memory models fuse together, focusing on PTX and x86. Our work proposes a higher-level programming model on top of CXL and is agnostic about the memory model's internals.

Multiple attempts have been made to provide concurrent algorithms with durability guarantees. Our work is inspired by Izraelevitz et al. [44], who propose a programming model for persistent memory, study correctness notions under this model, and suggest the first general construction for linearizable objects. However, their construction has limited applicability because of its performance overhead. Since then, several more specialized constructions have been proposed. NVTraverse [32] targets lock-free traversal data structures. Mirror [34] is an automatic transformation for any lock-free data structure. FliT [72] transforms any linearizable object without restriction, as discussed before. Montage [73] guarantees recoverability to *some* consistent state, but not necessarily to the most recent one prior to the crash. These constructions are designed for the full-system crash model and local persistent memory. In this paper we show how to

transform such objects to work in the more general partial-system failure model and with disaggregated memory.

Some systems are CXL ready. However, they neither provide a general programming model nor a transformation for existing algorithms. Āpta [60] is a design for function-as-a-service (FaaS) over CXL, it does not target shared memory and does not consider multiple architectures. Li et al. [52] designed Pond, a memory pooling system for CXL. Maruf et al. [57] propose TPP for OS-level page placement on top of CXL. Lupin [78] helps distributed applications tolerate partial failures using a shared CXL pod for replication. These system designs are orthogonal to this work.

Frameworks and systems for general disaggregated memory architectures enjoy increasing interest in recent years. Lee et al. [49] propose an RDMA based memory management system. Teleport [75] and Cowbird [18] offer techniques for better distributing work across the system. Calciu et al. [16] develop a software runtime that aims to improve memory access times. However, these do not target CXL and aim to improve either work distribution or memory access patterns. By contrast, our work contributes a correct general transformation for any concurrent algorithm.

8 Conclusion and Future Work

This work introduces CXL0, the first programming model designed for disaggregated memory using CXL technology. We demonstrate the applicability of CXL0 to current and future CXL setups, and illustrate its usefulness by developing a transformation that enables linearizable and durably linearizable concurrent algorithms to function correctly within a CXL environment. Adapting existing algorithms to the new model allows both researchers and practitioners to explore new opportunities in this design space.

We consider CXL0 to be a solid foundation that can be refined in various ways. We have already discussed two variants in §3.5. It would be useful to extend CXL0 with heterogeneous memory settings with non-CXL, privatized memory to capute more application use-cases. Another interesting refinement would be to distinguish between synchronous (posted, in PCIe terminology) and asynchronous (non-posted) CXL transactions (see §5.2 of the survey by Das Sharma et al. [25]). Finally, relaxing durability semantics has generally been shown to be beneficial for performance [7, 73] and can be explored here as well. Enhancing CXL0 with these perspectives will result in a richer model and potentially better-performing transformations and algorithms.

Acknowledgments

We thank Idit Keidar, Mark Silberstein, and Yoram Moses for their constructive comments and advice. We also thank Alberto Lerner and Sangjin Lee for assisting with the experiments and hardware setup, and the anonymous reviewers for their insightful feedback. Gal Assa was supported by the Hasso Plattner Institute. Ori Lahav was supported by the Israel Science Foundation (grant 814/22) and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811).

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. 2021. Deciding reachability under persistent x86-TSO. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. doi:10.1145/3434337

[2] Marcos K. Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. 2023. Memory disaggregation: why now and what are the challenges. *SIGOPS Oper. Syst. Rev.* 57, 1 (jun 2023), 38–46. doi:10.1145/3606557.3606563

[3] Marcos K Aguilera and Svend Frølund. 2003. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241* (2003).

[4] Hasan Al Maruf and Mosharaf Chowdhury. 2023. Memory Disaggregation: Advances and Open Challenges. *SIGOPS Oper. Syst. Rev.* 57, 1 (jun 2023), 29–37. doi:10.1145/3606557.3606562

[5] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. doi:10.1145/2627752

[6] ARM Corporation. 2023. Arm A-profile A64 Instruction Set Architecture. https://developer.arm.com/documentation/ddi0602/2023-12?lang=en.

[7] Gal Assa, Andreia Correia, Pedro Ramalhete, Valerio Schiavoni, and Pascal Felber. 2023. Tl4x: buffered durable transactions on disk as fast as in memory. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 245–259.

[8] Gal Assa, Moritz Lumme, Lucas Burgi, Michal Friedman, and Ori Lahav. 2024. A Programming Model for Disaggregated Memory over CXL. In *arXiv preprint arXiv:2407.16300*.

[9] Gal Assa, Moritz Lumme, Lucas Burgi, Michal Friedman, and Ori Lahav. 2026. Roq code of CXL0. https://www.github.com/cores-lab/cxl0.

[10] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, Calvin Newport and Idit Keidar (Eds.). ACM, 7–16. doi:10.1145/3212734.3212753

[11] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. 2016. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[12] Eleni Vafeiadi Bila, Brijesh Dongol, Ori Lahav, Azalea Raad, and John Wickerson. 2022. View-Based Owicki-Gries Reasoning for Persistent x86-TSO. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 234–261. doi:10.1007/978-3-030-99336-8_9

[13] Stefan Bodenmüller, John Derrick, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. 2023. A fully verified persistency library. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 26–47.

[14] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.* 11, 11 (2018), 1604–1617. doi:10.14778/3236187.3236209

[15] Wentao Cai, Haosen Wen, and Michael L. Scott. 2023. Transactional Composition of Nonblocking Data Structures. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (Orlando, FL, USA) *(SPAA '23)*. Association for Computing Machinery, New York, NY, USA, 187–197. doi:10.1145/3558481.3591079

[16] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. 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, 79–92. doi:10.1145/3445814.3446713

[17] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (feb 1985), 63–75. doi:10.1145/214451.214456

[18] Xinyi Chen, Liangcheng Yu, Vincent Liu, and Qizhen Zhang. 2023. Cowbird: Freeing CPUs to Compute by Offloading the Disaggregation of Memory. In *Proceedings of the ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 1060–1073. doi:10.1145/3603269.3604833

[19] 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 *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 16–31. doi:10.1145/3453483.3454027

[20] Compute Express Link Consortium. 2025. CXL 4.0 Specification. https://www.computeexpresslink.org.

[21] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th Symposium on Parallelism in Algorithms and Architectures*. ACM, 271–282.

[22] Bill Dally. 2011. Power, Programmability, and Granularity: The Challenges of ExaScale Computing. In *2011 IEEE International Parallel & Distributed Processing Symposium*. 878–878. doi:10.1109/IPDPS.2011.420

[23] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (jun 2020), 48–57. doi:10.1145/3361682

[24] Andrei Marian Dan, Patrick Lam, Torsten Hoefler, and Martin T. Vechev. 2016. Modeling and analysis of remote memory access programming. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 129–144. doi:10.1145/2983990.2984033

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

[26] Peter Desnoyers, Ian Adams, Tyler Estro, Anshul Gandhi, Geoff Kuenning, Mike Mesnier, Carl Waldspurger, Avani Wildani, and Erez Zadok. 2023. Persistent Memory Research in the Post-Optane Era. In *Proceedings of the 1st Workshop on Disruptive Memory Systems*. 23–30.

[27] Nan Ding, Samuel Williams, Hai Ah Nam, Taylor Groves, Muaaz Gul Awan, LeAnn Lindsey, Christopher Daley, Oguz Selvitopi, Leonid Oliker, and Nicholas Wright. 2022. Methodology for Evaluating the Potential of Disaggregated Memory Systems. In *2022 IEEE/ACM International Workshop on Resource Disaggregation in High-Performance Computing (REDIS)*. 1–11. doi:10.1109/RESDIS56595.2022.00006

[28] Emanuele D'Osualdo, Azalea Raad, and Viktor Vafeiadis. 2023. The Path to Durable Linearizability. *Proc. ACM Program. Lang.* 7, POPL (2023), 748–774. doi:10.1145/3571219

[29] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Comput. Surv.* 34, 3 (sep 2002), 375–408. doi:10.1145/568522.568525

[30] Mohammad Ewais and Paul Chow. 2023. Disaggregated Memory in the Datacenter: A Survey. *IEEE Access* 11 (2023), 20688–20712. doi:10.1109/ACCESS.2023.3250407

[31] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 608–621. doi:10.1145/2837614.2837615

[32] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. 2020. NVTraverse: in NVRAM data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 377–392.

[33] Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 28–40. doi:10.1145/3178487.3178490

[34] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: making lock-free data structures persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1218–1232.

[35] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A. W. Roscoe. 2014. FDR3 — A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014) (Lecture Notes in Computer Science, Vol. 8413)*. Springer, 187–201. doi:10.1007/978-3-642-54862-8_13

[36] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound Memory Models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1145–1168.

[37] Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. 2022. Checking robustness to weak persistency models. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 490–505. doi:10.1145/3519939.3523723

[38] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2021. Jaaru: efficiently model checking persistent memory programs. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 415–428. doi:10.1145/3445814.3446735

[39] Rachid Guerraoui and Ron R Levy. 2004. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* IEEE, 400–407.

[40] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[41] Intel Corporation. 2021. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html Accessed: 2024-09-18.

[42] Intel Corporation. 2022. Intel earnings statements, for Q2 2022. https://download.intel.com/newsroom/2022/corporate/Intel-CEO-CFO-2Q22-earnings-statements.pdf.

[43] Intel Corporation. 2023. Intel 64 and IA-32 Architectures Software Developer's Manual. https://cdrdv2.intel.com/v1/dl/getContent/671200.

[44] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing: 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings 30*. Springer, 313–327.

[45] 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 *57th IEEE/ACM International Symposium on Microarchitecture, MICRO 2024, Austin, TX, USA, November 2-6, 2024*. IEEE, 1504–1517. doi:10.1109/MICRO61859.2024.00110

[46] Artem Khyzha and Ori Lahav. 2021. Taming x86-TSO persistency. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. doi:10.1145/3434328

[47] Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. 2021. PerSeVerE: persistency semantics for verification under ext4. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. doi:10.1145/3434324

[48] R. Koo and S. Toueg. 1987. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering* SE-13, 1 (1987), 23–31. doi:10.1109/TSE.1987.232562

[49] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 488–504. doi:10.1145/3477132.3483561

[50] Alberto Lerner and Gustavo Alonso. 2024. CXL and the Return of Scale-Up Database Engines. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2568 – 2575. doi:10.14778/3675034.3675047

[51] Philip Levis, Kun Lin, and Amy Tai. 2023. A Case Against CXL Memory Pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 18–24. doi:10.1145/3626111.3628195

[52] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835

[53] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.* 7, 4 (1989), 321–359. doi:10.1145/75104.75105

[54] Nan Li and Wojciech Golab. 2021. Detectable sequential specifications for recoverable shared objects. In *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[55] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. 1–12. doi:10.1109/HPCA.2012.6168955

[56] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 257–270. doi:10.1145/3297858.3304043

[57] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. doi:10.1145/3582016.3582063

[58] NVIDIA Corporation. 2024. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[59] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 391–407. doi:10.1007/978-3-642-03359-9_27

[60] 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. doi:10.1109/DSN58367.2023.00030

[61] Derrick Quinn, Mohammad Nouri, Neel Patel, John Salihu, Alireza Salemi, Sukhan Lee, Hamed Zamani, and Mohammad Alian. 2025. Accelerating Retrieval-Augmented Generation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Kaitai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 15–32. doi:10.1145/3669940.3707264

[62] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency semantics of the Intel-x86 architecture. *Proc. ACM Program. Lang.* 4, POPL (2020), 11:1–11:31. doi:10.1145/3371079

[63] Pedro Ramalhete, Andreia Correia, and Pascal Felber. 2021. Efficient algorithms for persistent transactional memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–15.

[64] Kiran Ranganath, Jesun Firoz, Joshua Suetterlein, Joseph Manzano, Andres Marquez, Mark Raugas, and Daniel Wong. 2022. LC-MEMENTO: A Memory Model for Accelerated Architectures. In *Languages and Compilers for Parallel Computing*, Xiaoming Li and Sunita Chandrasekaran (Eds.). Springer International Publishing, Cham, 67–82.

[65] Gal Sela, Maurice Herlihy, and Erez Petrank. 2021. Brief Announcement: Linearizability: A Typo. In *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen (Eds.). ACM, 561–564. doi:10.1145/3465084.3467944

[66] Gal Sela and Erez Petrank. 2021. Durable Queues: The Second Amendment. In *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, Kunal Agrawal and Yossi Azar (Eds.). ACM, 385–397. doi:10.1145/3409964.3461791

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

[68] Debendra Das Sharma. 2018. Compute Express Link. https://docs.wixstatic.com/ugd/0c1418_d9878707bbb7427786b70c3c91d5fbd1.pdf.

[69] Anubhav Srivastava and Trevor Brown. 2022. Elimination (a, b)-trees with fast, durable updates. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 416–430. doi:10.1145/3503221.3508441

[70] Jim Stevens, Paul Tschirhart, and Bruce Jacob. 2016. Fast full system memory checkpointing with SSD-aware memory controller. In *Proceedings of the Second International Symposium on Memory Systems* (Alexandria, VA, USA) *(MEMSYS '16)*. Association for Computing Machinery, New York, NY, USA, 96–98. doi:10.1145/2989081.2989126

[71] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. 2021. Concordia: Distributed shared memory with {In-Network} cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 277–292.

[72] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. 2022. FliT: A Library for Simple and Efficient Persistent Algorithms *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 309–321. doi:10.1145/3503221.3508436

[73] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. 2021. A Fast, General System for Buffered Persistent Data Structures. In *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, Xian-He Sun, Sameer Shende, Laxmikant V. Kalé, and Yong Chen (Eds.). ACM, 73:1–73:11. doi:10.1145/3472456.3472458

[74] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[75] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2022. Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1345–1359. doi:10.1145/3514221.3517856

[76] Yuhong Zhong, Daniel S. Berger, Carl A. Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 37–56. https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong

[77] Yuhong Zhong, Daniel S. Berger, Pantea Zardoshti, Enrique Saurez, Jacob Nelson, Antonis Psistakis, Joshua Fried, and Asaf Cidon. 2025. My CXL Pool Obviates Your PCIe Switch. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems, HotOS 2025, Banff, AB, Canada, May 14-16, 2025*. ACM, 58–66. doi:10.1145/3713082.3730393

[78] 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, DIMES 2024, Austin, TX, USA, 3 November 2024*. ACM, 41–50. doi:10.1145/3698783.3699377

[79] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 128:1–128:26. doi:10.1145/3360554