Formalising CXL Cache Coherence

Chengsong Tan*
Kaihong
Shenzhen, China

Alastair F. Donaldson
Imperial College London
London, UK

John Wickerson
Imperial College London
London, UK

Abstract

We report our experience formally modelling and verifying `CXL.cache`, the inter-device cache coherence protocol of the Compute Express Link standard. We have used the Isabelle proof assistant to create a formal model for `CXL.cache` based on the English prose specification. This led to us identifying and proposing fixes to several parts of the specification that were unclear, ambiguous or inaccurate. Nearly all our issues and proposed fixes have been confirmed and tentatively accepted by the CXL consortium for adoption, save for one which is still under discussion. To validate the faithfulness of our model we performed scenario verification of essential restrictions such as "Snoop-pushes-GO", and used the Isabelle proof assistant to produce a fully mechanised proof of a coherence property of the model. The considerable size of this proof, comprising tens of thousands of lemmas, prompted us to develop new proof automation tools, which we have made available for other Isabelle users working with similarly cumbersome proofs.

CCS Concepts: • **Computer systems organization** → **Architectures**; **Architectures**; • **Theory of computation** → **Logic and verification**; **Logic and verification**;

Keywords: CXL, Cache Coherence, Proof Assistant, Heterogeneous Computing, Formal Proof

ACM Reference Format:
Chengsong Tan, Alastair F. Donaldson, and John Wickerson. 2025. Formalising CXL Cache Coherence. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3676641.3715999

1 Introduction

Compute Express Link (CXL) [8] is an emerging standard that provides cache coherence across multiple devices connected along a PCIe bus. Inter-device cache coherence is a boon to computer architects because it allows multiple devices to communicate with each other while transferring a minimal amount of data between them. CXL has the potential to be faster than other memory expansion methods [13] and save stranded memory in cloud computing clusters [19].

CXL is not the first standard for inter-device cache coherence [5, 6, 12, 17, 20, 28, 30, 34], but it is the first to enjoy broad support across the computer industry, with backers including Alibaba, AMD, Arm, Broadcom, Cisco, Dell, Ericsson, Google, Hewlett Packard, Huawei, IBM, Intel, Meta, Microsoft, Nvidia, Oracle, Qualcomm, Samsung, Synopsys, Xilinx, and many others.

The CXL standard is large, complex and new, and is set to form a trusted pillar of datacenter computers for years to come.[1] As such, now is the ideal time to study the standard intensively. Does it contain inconsistencies? Is the wording unambiguous throughout? And perhaps most importantly: does it actually provide its stated guarantee of inter-device cache coherence?

We report here on our efforts to answer those questions.

***Contribution 1: Formalising* `CXL.cache`.** The part of the CXL standard that provides inter-device cache coherence is called `CXL.cache`. (The other two parts of the standard are `CXL.io`, which governs bulk data transfers, and `CXL.mem`, which relates to disaggregated memory.) Our first contribution is a formalisation of the `CXL.cache` protocol in the Isabelle proof assistant [22]. Our formalisation takes the form of a state-transition system. It comprises a detailed model of the whole-system state (encompassing the state of caches in devices plus the contents of the various channels that contain messages sent between the devices and the 'host'), together with dozens of transition rules that define the legal ways the state can evolve in response to CXL messages being passed around and processed.

We explain in Section 3 how our formalisation corresponds to the informal prose given in the official CXL standard, and which assumptions we have made in our modelling process.

As a direct result of our modelling efforts, we uncovered five areas where the CXL standard could be improved (one inconsistency, one redundancy, one inefficiency, and two places where the intention could be clarified). We have proposed corresponding improvements to the text to the engineers who lead the drafting of the protocol. In four cases they have confirmed that these will be incorporated into the

*Work done while the author was at Imperial College London.

ASPLOS '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/2025/03
https://doi.org/10.1145/3676641.3715999

[1] Yole Group anticipates a CXL market size of $15.8 billion by 2028 [14].

```
lemma inv_preservation_i_j:
  fixes Σ, Σ′ :: state
  assumes inv_1(Σ) ∧ ... ∧ inv_796(Σ)
  assumes rule_i(Σ,Σ′)
  shows inv_j(Σ′)
```

Figure 1. Lemmas of this form state that all rules preserve all conjuncts of the invariant. They assume all 796 conjuncts hold in state Σ, and that the ith rule can evolve the state to Σ′, then show that the jth conjunct of the invariant still holds.

next version of the standard, with one of our proposed fixes still under discussion.

Contribution 2: Proving a cache coherence property. Our second contribution involves putting our formalisation to work.

First, we use it for a form of 'scenario verification': we use a number of litmus tests (some derived from message-sequence charts in the CXL specification) and run them through our model to confirm that legal interactions are indeed allowed and illegal interactions are indeed forbidden. This helps gain confidence in the faithfulness of our model to the standard. We also use scenario verification to show that if some requirements imposed by the standard are relaxed, then coherence is not met—this establishes confidence that the CXL standard is not overly 'strong'; i.e., that it does not impose requirements on CXL implementations without good reason.

Second, we prove that it satisfies the 'single writer, multiple reader' (SWMR) property [21, p. 11]. The SWMR property states that if one device has write access to a location, then no other device can simultaneously have read or write access to the same location. SMWR is one of the two properties that are, together, sufficient to establish cache coherence; the other is the 'data-value invariant' [21, p. 13], which we leave as future work.

Contribution 3: Better automation for large proofs. Our proof that our model of CXL satisfies the SWMR property is large. SWMR is not inductive on its own, so to complete the proof, we needed to devise a stronger invariant (one that implies SWMR), and prove that this invariant holds for all legal initial states of the system and is preserved by every transition rule. The invariant is made up of 796 conjuncts, and there are 68 transition rules, hence we must prove 53,332 lemmas of the form given in Figure 1.

Most of these obligations can be automatically discharged via a single call to Isabelle's *sledgehammer* [25], but this process still requires manual intervention to copy the proof snippet discovered by sledgehammer into the overall theory file. The situation is worsened by the fact that this is not a one-shot effort: our invariant had to be revised many times during the proof development process, because we frequently found that an extra conjunct was needed in order to make one of the lemmas hold. Moreover, each time a conjunct was added, it was necessary to show that it is preserved by all of the transition rules, which in turn often led to the need for further conjuncts!

As such, robust proof automation is a necessity, and our third contribution is a small but useful utility for Isabelle that allows the sledgehammer to be used in a completely unsupervised mode: the utility invokes sledgehammer on all the `sledgehammer` commands in a given theory file, and if a proof is found, substitutes it into the theory file directly. We further improve on this by automatically invoking multiple `sledgehammer` instances on a generated Isar (structured Isabelle) proof skeleton, and filling in the skeleton with the found proofs. We found this utility indispensable for completing our proof, and we have made it freely available for other Isabelle users working with similarly cumbersome proofs.

Auxiliary material. Our Isabelle theory files, containing the definitions of our CXL model and the proof that it meets the SWMR property, are available on GitHub [31].

Paper outline. To provide intuition we first provide an overview of `CXL.cache` (Section 2). We then present salient details of our Isabelle formal model using standard mathematical notation (Section 3), and describe the problems with the `CXL.cache` standard identified during the construction of this model, and our proposed fixes (Section 4). We then explain how we validated our model using scenario verification (Section 5) and by proving an important coherence-related property—SWMR (Section 6). This large proof required some innovations in proof engineering, which we describe (Section 7). We recap the assumptions and limitations on which our modelling and proof work is based (Section 8), and discuss related and future work (Section 9).

2 Overview of `CXL.cache`

Before describing our formal model in detail (Section 3), we provide an intuitive overview of the `CXL.cache` protocol.

Multicore processors employ cache coherence protocols to ensure that multiple copies of the same data across different cores' caches remain in sync. With the rise of heterogeneous computing—where CPUs, GPUs, and specialized accelerators must work closely together—there is also a need for a global cache coherence protocol that manages data consistency across heterogeneous processors. This is the problem that `CXL.cache` is designed to solve.

`CXL.cache` allows devices like standalone GPUs and ASICs to cache a CPU's memory as if they are cores within the CPU's own multicore system. This facilitates correct and fine-grained data sharing with low latency, as the complex mechanisms that achieve cache coherence are managed at the hardware level. For example, it might be desirable for an Intel CPU to be connected with an AMD GPU and an NVIDIA

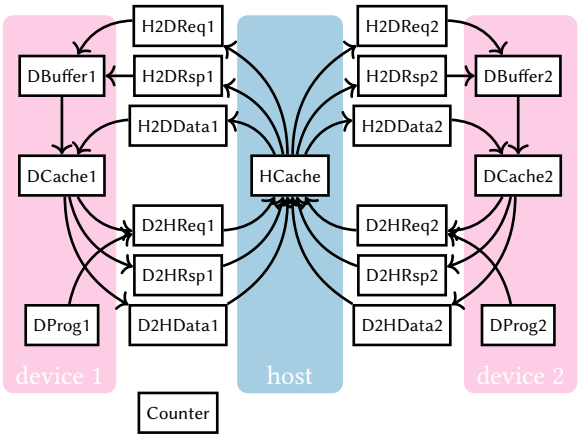

Figure 2. An overview of our model of a two-device CXL state

SmartNIC via some fast interconnect, allowing these accelerators to cache and share the CPU's memory within the same cache-coherent domain. If these devices have `CXL.cache` enabled, then this would be seamless. Capabilities like these are valuable for data centers that require composable infrastructure, where resources can be dynamically allocated and combined to optimize performance for diverse workloads.

`CXL.cache` is an asymmetric protocol, with coherence-related messages between *devices* (typically accelerators) all going via a central *host* (CPU). Figure 2 shows the main components of a CXL system with two devices, with arrows indicating the direction of the messages that are passed between them, and coloured backgrounds to indicate which components belong to a device and which belong to the host.

`CXL.cache` assumes that each device maintains the coherence of its own internal cache hierarchy, and hence is able to treat each device as having a single cache (DCache1 and DCache2). The host also has a cache (HCache). Each cacheline can be in one of four 'stable' states: modified (write-access and dirty), exclusive (write-access and clean), shared (read-access), and invalid.

There are various channels from the host to a device (H2D) and from a device to the host (D2H) along which requests, responses and data can be sent. These channels are separated to allow them to be implemented with different latencies. Transactions along these channels can be categorised as follows:

- **D2H Request:** A device may send the host a request for read-access (RdShared) or write-access (RdOwn) to a location.
- **H2D Request:** The host may need to invalidate the cacheline on a second device via a snoop-invalidate (SnpInv) request.
- **D2H Response:** The second device may respond with RspIHitSE to report that it is invalidating its cacheline having previously enjoyed shared or exclusive access.
- **H2D Response:** Finally, the host replies to the first device with a GO-Shared or GO-Modified message to grant the desired access.

Our model of the CXL state in Figure 2 also includes buffers, programs, and a counter, all of which will be explained in Section 3.2.

To improve performance and reduce latency, `CXL.cache` permits weaker ordering guarantees than traditional interconnects like PCIe [26]. Specifically, it does not enforce ordering between different memory locations and provides minimal ordering on the same cache block [10]. This allows scenarios where a device reads updated data before a synchronization flag is set, enabling more concurrency on the same physical network.

3 A formal model of CXL

Our formal model of CXL is expressed in the language of the Isabelle proof assistant [22], and is provided as a set of Isabelle theory files in our GitHub repository [31].

In this section, we present salient details of the model (using mathematical notation rather than Isabelle syntax). The model has been carefully constructed from our reading of the `CXL.cache` specification, and refined based on discussions with cache coherence experts from the CXL consortium. The creation of the model led to us identifying and proposing fixes for several problems in `CXL.cache`, of which we elaborate on a selection in Section 4. We have validated our model using scenario verification (Section 5) and mechanised proof (Section 6).

3.1 An overview of our CXL state model

The main components of our model are the host, the devices, and the channels between them, as already shown in Figure 2. The caches and channels in that figure are all directly taken from the CXL specification [9, §3.2.1]. The program components (DProg1 and DProg2) are an invention of ours—they are solely used to control the sequence of state transitions when exploring specific scenarios in Section 5. They only serve to trigger coherence transactions, and do not modify locations or read out values. The standard does not specify how devices come up with unique transaction identifiers, so we use a simple, globally accessible counter (Counter). The buffers (DBuffer1 and DBuffer2) are another invention of ours; they are used to simulate the dependence between the H2D Response and H2D Request channels that is implied by the standard [9, §3.2.5].

In an effort to keep the proof tractable, we have fixed the number of devices to two. This means that our proof cannot guarantee the absence of coherence violations that only manifest when three or more devices interact; but for analyzing and prototyping purposes it is common to start with two devices [27].

3.2 Details of our CXL state model

Figure 3 presents the type of each of the twenty components that appear in Figure 2. We now explain those types in detail.

There are three 'stable' cacheline states. We do not track the distinction between exclusive state and modified state, because transitions between these states have no effect on ownership, and the SWMR property that we are interested in proving is phrased only in terms of ownership. So, we use M when the cacheline is in either of these states, alongside S for shared and I for invalid.

While a transaction is being carried out, a cacheline can be in one of several additional 'transient' states, depending on whether it is held on a device (*DTransientState*) or on the host (*HTransientState*). For instance, $\mathsf{IM}^{\mathsf{AD}}$ refers to a cacheline that is awaiting an acknowledgement (A) and some data (D) in order to complete its transition from the invalid state (I) to the modified state (M). These transient states are not officially part of the `CXL.cache` specification, so we follow the standard notation for them [21].

A cacheline (*HCache* or *DCache*) consists of a value (*Val*) together with a stable or transient state. We are concerned with coherence, which is a property of a single memory location, so we assume without loss of generality that our caches contain just a single location.

Messages can be grouped into transactions, and each transaction has an identifier *Tid*. A single transaction may involve several request and response messages; for instance, a device may send a request to the host, which requires the host to send a request to another device, which then responds to the host, finally allowing the host to respond to the first device.

A device can request from the host (*D2HReq*) read-only access (RdShared) or write access (RdOwn). Additionally, it can relinquish access to a location that has not been written (CleanEvict) or that has been written (DirtyEvict). The message CleanEvictNoData is the same as CleanEvict, but the device is additionally signalling that it will refuse to provide the (clean) data and the host must not request it.

There are additional device-to-host requests that we exclude from our model: RdCurr simply checks the current data value and does not affect ownership (nor coherence); RdAny's functionality is already covered by RdOwn and RdShared; RdOwnNoData is no different from RdOwn from the perspective of the SWMR property; and although messages ItoMWr, WrCur, CLFlush, WOWrInv, WOWrInvF and WrInv are interesting from a memory-ordering point of view, they are not interesting for coherence.

A host can respond to a device (*H2DRsp*) by sending a 'global observation' message (GO). This signifies that the host believes the device's request has now been seen by all relevant parties, and can now be considered complete [9, §3.2.2.1]. If the device has sent an evict request, the host can respond by instructing the device to send its data to the host (GO_WritePull), or to discard its data (GO_WritePullDrop)

$$
\begin{aligned}
\mathit{StableState} &\stackrel{\text{def}}{=} \{\mathsf{M}, \mathsf{S}, \mathsf{I}\} \\
\mathit{DTransientState} &\stackrel{\text{def}}{=} \{\mathsf{IM}^{\mathsf{AD}}, \mathsf{IM}^{\mathsf{A}}, \mathsf{IM}^{\mathsf{D}}, \mathsf{SM}^{\mathsf{AD}}, \mathsf{SM}^{\mathsf{D}}, \mathsf{SM}^{\mathsf{A}}, \\
&\qquad \mathsf{IS}^{\mathsf{D}}, \mathsf{IS}^{\mathsf{AD}}, \mathsf{IS}^{\mathsf{A}}, \mathsf{MI}^{\mathsf{A}}, \mathsf{SI}^{\mathsf{A}}, \mathsf{II}^{\mathsf{A}}, \mathsf{SI}^{\mathsf{AC}}\} \\
\mathit{HTransientState} &\stackrel{\text{def}}{=} \{\mathsf{M}^{\mathsf{AD}}, \mathsf{M}^{\mathsf{A}}, \mathsf{M}^{\mathsf{D}}, \mathsf{S}^{\mathsf{AD}}, \mathsf{S}^{\mathsf{D}}, \mathsf{S}^{\mathsf{A}}, \mathsf{I}^{\mathsf{D}}, \mathsf{I}^{\mathsf{B}}, \mathsf{S}^{\mathsf{B}}, \\
&\qquad \mathsf{M}^{\mathsf{B}}\} \\
\mathit{DState} &\stackrel{\text{def}}{=} \mathit{DTransientState} \cup \mathit{StableState} \\
\mathit{HState} &\stackrel{\text{def}}{=} \mathit{HTransientState} \cup \mathit{StableState} \\
\mathit{HCache} &\stackrel{\text{def}}{=} (\!|\mathsf{Val} : \mathit{Val}, \mathsf{State} : \mathit{HState}|\!) \\
\mathit{DCache} &\stackrel{\text{def}}{=} (\!|\mathsf{Val} : \mathit{Val}, \mathsf{State} : \mathit{DState}|\!) \\
\mathit{Tid} &\stackrel{\text{def}}{=} \mathbb{N} \\
\mathit{D2HReqType} &\stackrel{\text{def}}{=} \{\mathsf{RdShared}, \mathsf{RdOwn}, \mathsf{CleanEvict}, \\
&\qquad \mathsf{DirtyEvict}, \mathsf{CleanEvictNoData}\} \\
\mathit{D2HReq} &\stackrel{\text{def}}{=} \mathit{D2HReqType} \times \mathit{Tid} \\
\mathit{D2HRspType} &\stackrel{\text{def}}{=} \{\mathsf{RspIHitSE}, \mathsf{RspIFwdM}, \mathsf{RspSFwdM}\} \\
\mathit{D2HRsp} &\stackrel{\text{def}}{=} \mathit{D2HRspType} \times \mathit{Tid} \\
\mathit{H2DReqType} &\stackrel{\text{def}}{=} \{\mathsf{SnpData}, \mathsf{SnpInv}\} \\
\mathit{H2DReq} &\stackrel{\text{def}}{=} \mathit{H2DReqType} \times \mathit{Tid} \\
\mathit{H2DRspType} &\stackrel{\text{def}}{=} \{\mathsf{GO}, \mathsf{GO\_WritePull}, \\
&\qquad \mathsf{GO\_WritePullDrop}\} \\
\mathit{H2DRsp} &\stackrel{\text{def}}{=} \mathit{H2DRspType} \times \mathit{DState} \times \mathit{Tid} \\
\mathit{Data} &\stackrel{\text{def}}{=} \mathit{Tid} \times \mathit{Val} \\
\mathit{DBuffer} &\stackrel{\text{def}}{=} \mathit{H2DRsp} \cup \mathit{H2DReq} \cup \{\bot\} \\
\mathit{Instruction} &\stackrel{\text{def}}{=} \{\mathtt{Load}, \mathtt{Store}, \mathtt{Evict}\} \\
\mathit{SystemState} &\stackrel{\text{def}}{=} (\!|\mathsf{DProg1} : \mathit{Instruction}\,\text{list}, \\
&\qquad \mathsf{DProg2} : \mathit{Instruction}\,\text{list}, \\
&\qquad \mathsf{DCache1} : \mathit{DCache}, \\
&\qquad \mathsf{DCache2} : \mathit{DCache}, \\
&\qquad \mathsf{D2HReq1} : \mathit{D2HReq}\,\text{list}, \\
&\qquad \mathsf{D2HReq2} : \mathit{D2HReq}\,\text{list}, \\
&\qquad \mathsf{D2HRsp1} : \mathit{D2HRsp}\,\text{list}, \\
&\qquad \mathsf{D2HRsp2} : \mathit{D2HRsp}\,\text{list}, \\
&\qquad \mathsf{D2HData1} : \mathit{Data}\,\text{list}, \\
&\qquad \mathsf{D2HData2} : \mathit{Data}\,\text{list}, \\
&\qquad \mathsf{H2DReq1} : \mathit{H2DReq}\,\text{list}, \\
&\qquad \mathsf{H2DReq2} : \mathit{H2DReq}\,\text{list}, \\
&\qquad \mathsf{H2DRsp1} : \mathit{H2DRsp}\,\text{list}, \\
&\qquad \mathsf{H2DRsp2} : \mathit{H2DRsp}\,\text{list}, \\
&\qquad \mathsf{H2DData1} : \mathit{Data}\,\text{list}, \\
&\qquad \mathsf{H2DData2} : \mathit{Data}\,\text{list}, \\
&\qquad \mathsf{DBuffer1} : \mathit{DBuffer}, \\
&\qquad \mathsf{DBuffer2} : \mathit{DBuffer}, \\
&\qquad \mathsf{DProg1} : \mathit{Instruction}\,\text{list}, \\
&\qquad \mathsf{DProg2} : \mathit{Instruction}\,\text{list}, \\
&\qquad \mathsf{HCache} : \mathit{HCache}, \\
&\qquad \mathsf{Counter} : \mathbb{N}\,|\!)
\end{aligned}
$$

Figure 3. Our model of a CXL state

[9, §3.2.4.2.14]. In all cases, a host-to-device response includes the new *DState* that the device's cacheline should enter. There are additional host-to-device responses that we exclude from our model: WritePull is only used in response to WrInv requests which, as mentioned above, we do not model, and FastGOWritePull and ExtCmp provide an advanced optimisation where a device can indicate that an update is partially observable (FastGOWritePull) and then globally observable (ExtCmp). We currently do not model non-ideal network conditions or error-handling and therefore leave out the GOErrWritePull message too.

A host-to-device request (*H2DReq*) is a snoop, used to check (and change) the status of the device's cacheline [9, §3.2.4.4]. If the request is a SnpData, the device must downgrade its cacheline state to either S or I, and if the request is a SnpInv, the device must downgrade its cacheline state to I. In both cases, the device must send its data to the host if it is dirty. There exists also a SnpCur request for checking a device's cacheline without changing it, but we omit this from our model because it does not affect coherence.

A device-to-host response (*D2HRsp*) can be a RspIHitSE (which means that the device has downgraded from S or E to I [9, §3.2.4.3.3]), a RspIFwdM (which means that the device has downgraded from M to I and is also forwarding its dirty data [9, §3.2.4.3.6]), or a RspSFwdM (which means that the device has downgraded from M to S and is also forwarding its dirty data [9, §3.2.4.3.5]). There are additional device-to-host responses that we exclude from our model. RspIHitI is not used because our model's host tracks device states and does not send out snoops unnecessarily. The transaction flows of RspVHitV, RspSHitSE and RspVFwdV are very similar to those of RspSHitSE, RspIHitSE and RspSFwdM, respectively; we leave them out to avoid duplication in our proof.

Finally, each device's buffer (*DBuffer*) contains a single request or response message from the host, or is empty ($\perp$).

3.3 CXL transitions

Our model consists of 68 rules that describe transitions between CXL states. Figure 4 presents a selection of these rules. Each rule consists of a name, a set of **guards** that must all hold in order for a rule to fire, and a set of **actions** by which some components of the state are (atomically) updated.

The InvalidLoad1 rule says that if device 1's cache is in the invalid (I) state (first guard) and it wishes to perform a load (second guard), then it can request an upgrade to the shared (S) state (first action), enter the $\mathsf{IS}^{\mathsf{AD}}$ state in the meantime (second action), and increment the transaction-identifier counter (third action).

The ModifiedStore1 rule says that if device 1's cache is already in the modified (M) state (first guard) and it intends to do a store (second guard), then no coherence messages are necessary; it need only write to its own cache (first action) and consider the instruction complete (second to fourth actions). To provide the reader with an intuitive store semantics we include the value written here, but we drop this during our proof because the SWMR property is independent of values; it cares only about ownership.

InvalidLoad1
guards: DCache1.State = I
head(DProg1) = Load
actions: D2HReq1 := D2HReq1@[(RdShared, Counter)]
DCache1.State := $\mathsf{IS}^{\mathsf{AD}}$
Counter := Counter + 1

ModifiedStore1
guards: DCache1.State = M
head(DProg1) = Store
actions: DCache1.*Val* := v
DProg1 := *tail*(DProg1)
DBuffer1 := EmptyBuffer
Counter := Counter + 1

SharedSnpInv1
guards: DCache1.State = S
head(H2DReq1) = (SnpInv, *txid*)
H2DRsp1 = []
actions: DCache1.State := I
H2DReq1 := *tail*(H2DReq1)
DBuffer1 := (SnpInv, *txid*)
D2HRsp1 := D2HRsp1@[(RspIHitSE, *txid*)]

HostModifiedDirtyEvict1
guards: HCache.State = M
DCache1.State = MI^{A}
head(D2HReq1) = (DirtyEvict, *txid*)
H2DData1 = D2HRsp1 = []
actions: HCache.State := I^{D}
D2HReq1 := *tail*(D2HReq1)
H2DRsp1 := H2DRsp1@[(GO_WritePull, I, *txid*)]
DBuffer1 := EmptyBuffer

Figure 4. A selection of our transition rules.

The SharedSnpInv1 rule describes how a device deals with snoop requests from the host. If device 1's cache is in shared (S) state (first guard) and the head of its H2D Requests channel is a SnpInv (guard 2), then its cache is invalidated (action 1), the H2D Request is removed (action 2) and put into the device's buffer (action 3), and a response is sent back to the host using the same transaction-identifier (action 4). This rule only fires if there are no outstanding H2D Responses

(guard 3); this requirement captures the 'Snoop-pushes-GO' rule, which dictates that an H2D Request (snoop) message cannot overtake an H2D Response (GO) message to the same device:

> When the host returns a GO response to a device, the expectation is that a snoop arriving to the same address of the request receiving the GO would see the results of that GO. [9, §3.2.5.2]

We will show how relaxing this rule leads to a coherence violation in Section 5.

The HostModifiedDirtyEvict1 rule describes a device requesting to evict dirty data. The rule fires if the host's cache is in modified (M) state (first guard), the device's cache is in the process of changing state from modified (M) state to invalid (I) state (MI^A, second guard), and the device has sent a DirtyEvict request (third guard). The host's cache enters the I^D state because it will enter the invalid state once data arrives (first action), the D2H Request is removed (second action) and a corresponding H2D Response is issued (third action). The requirement that there are no H2D Data or D2H Response messages in-flight (fourth guard) is derived from the following 'GO-cannot-tailgate-snoop' rule:

> When the host is sending a snoop to the device, the requirement is that no GO response will be sent to any requests with that address in the device until after the Host has received a response for the snoop and all implicit writeback (IWB) data [...] has been received. [9, §3.2.5.2]

This requires the H2D Request, D2H Response, and D2H Data channels to contain no messages to the same address when sending a GO message.

4 Fixing problems in the standard

The weight of industrial support behind CXL makes it likely that the standard will be implemented by multiple vendors over the coming years. To ensure compatibility between implementations from different vendors, it is thus essential that the standard is precise and unambiguous.

Unfortunately, we have found that the current `CXL.cache` standard [9] suffers from numerous inaccuracies and ambiguities. We give some examples, which were discovered during the process of creating the formal model described in Section 3. We have proposed fixes to address these shortcomings, and have discussed them with members of the CXL consortium who lead drafting of the `CXL.cache` protocol. As detailed below, in most cases our proposed fixes have been agreed and will be adopted in future versions of the standard.

4.1 Ambiguity/inaccuracy regarding multiple snoops

We believe that the following rule

> The host is only allowed to have one snoop pending at a time per cacheline address per device. [9, §3.2.5.5]

is ambiguous because 'per' appears more than once. If a host has two snoops pending, must they be to different addresses *and* different devices? Or must they be to different addresses *or* different devices? In fact, neither of these interpretations is quite correct, because what the rule does not mention that it can be legal to have multiple pending snoops on the same cacheline, as long as they belong to the same transaction.

Proposed fix. We propose to amend the text as follows:

> ~~The host is only allowed to have one snoop pending at a time per cacheline address per device.~~ At no time is the host allowed to have two or more snoops on the same cacheline address pending, unless they use the same transaction identifier and target different devices.

Our proposal has tentatively been agreed by the CXL consortium and we are working with them to fine-tune the wording.

4.2 Redundant rule about multiple snoops

There is some redundancy between rules about sending multiple snoops to the same address. Specifically, the following rule:

> 11. The Host must not send a second snoop request to an address until all responses, plus data if required, for the prior snoop are collected. [9, §3.2.5.14]

repeats what is already specified earlier (we note that we have faithfully transcribed quotes from the specification, which inconsistently capitalises "host"):

> The host must wait until it has received both the snoop response and all IWB data (if any) before dispatching the next snoop to that address. [9, §3.2.5.5]

Proposed fix. To avoid confusion, we propose removing Rule 11 from §3.2.5.14. Our proposal has been accepted by the CXL consortium and is due to be adopted.

4.3 Clarification about WritePull responses

The following rule:

> Conversely, the host may not launch a WritePull for a write until it has received the snoop response (including data in case of RspFwd) for any snoops to the pending write's address. [9, §3.2.5.3]

enjoys a subtle interaction with a rule in §3.2.5.2 that restricts the launching of GO messages. Restricting GO messages is almost enough on its own; the only 'gap' that the restriction on WritePull messages fills relates to WrInv requests, to which hosts can respond with a WritePull rather than a GO.

Proposed fix. To clarify this subtlety to the reader, we suggest the following amendment:

> Conversely, the host may not launch a WritePull (in response to a WrInv) for a write until it has received the snoop response (including data in case of RspFwd) for any snoops to the pending write's address.

Our proposal has tentatively been agreed by the CXL consortium and we are working with them to fine-tune the wording.

4.4 Potential optimisation when evicting stale data

CXL requires that:

> if a device Evict transaction has been issued [...] but has not yet processed its WritePull from the host, and a snoop hits the writeback, the device must [...] set the Bogus field in all the D2H data messages sent to the host. The intent is to communicate to the host that [...] the data from the Evict is potentially stale. [9, §3.2.5.4]

In other words, if a device has requested to evict some data, and the host has determined via a snoop that this data is already stale, then the device should not send the data back to the host; it should instead mark its data messages as 'bogus'. An alternative in this situation could be for the host to send a WritePullDrop to the device rather than a WritePull, which instructs the device not to send any data messages at all. This could offer an efficiency gain by avoiding some D2H data traffic.

Proposed fix. In Table 3-23 ("D2H Request (Targeting Non Device-attached Memory) Supported H2D Responses"), add a '⋆' in the "DirtyEvict / GO_WritePullDrop" cell. The meaning of the '⋆' shall be:

> if the Host has been able to determine that the device's data is stale, by means of a prior snoop, then the Host may issue a GO_WritePullDrop rather than a GO_WritePull.

This proposal remains under discussion with the CXL consortium, who are evaluating its backward-compatibility and whether it represents a meaningful opportunity for improving performance.

4.5 Other clarifications

We have also discussed some other minor clarifications to the specification with the CXL consortium, such as adding a note to the beginning of the Device-to-Host Requests section [9, §3.2.4.2] to clarify that certain requests are only legal when the device's cache is in a certain state (as this is currently not explained until about 20 pages later [9, §3.2.5.15]).

5 Scenario verification

In Section 6 we turn our attention to using Isabelle to prove that our model of `CXL.cache` satisfies the SWMR property. Before that, in this section, we describe the *scenario verification* activities we undertook before embarking on this proof. These serve as important smoke tests to confirm that our formal model of `CXL.cache` actually behaves as one would expect in a variety of scenarios; if this were not the case then our formal proof would be meaningless. Additionally, scenario verification allows us to scrutinise various restrictions that the `CXL.cache` protocol imposes to assess whether they are really necessary—i.e. whether relaxing these restrictions can lead to coherence violations. This represents an important use case for our model beyond being a vehicle for formal proofs: it has the potential to allow protocol designers to rigorously understand the implications of the protocol rules and the consequences of relaxing them.

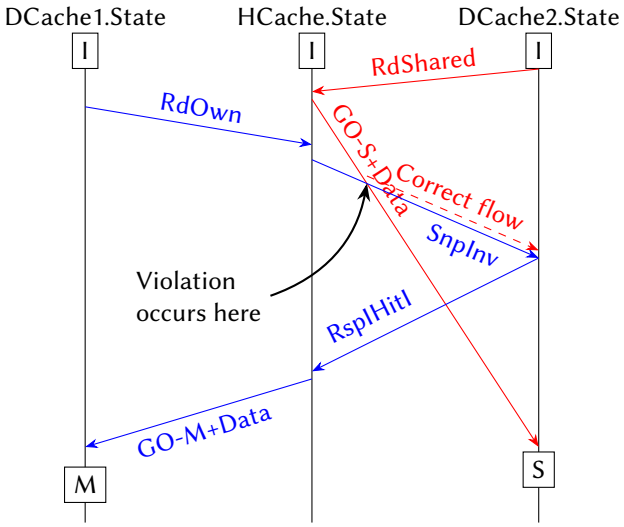

Figure 5. A message sequence chart from [33] demonstrating a coherence violation if the snoop-pushes-GO rule is relaxed

All three of the scenarios described in this section are produced in a semi-automatic way by Isabelle using its `value` command. We provide the programs that the two devices should run, and give Isabelle a bound on the path depth to explore. We say *semi*-automatic because in some cases it is also necessary to manually prune the tree of possible paths by adding extra predicates, in order to guide Isabelle towards a solution that we already have in mind. Without this, the search space can be so large that Isabelle does not terminate within a reasonable time bound.

5.1 Smoke testing via litmus tests

To smoke test our model, we created a series of litmus tests. Each litmus test initialises the system in a state where the two devices are poised to issue a particular series of requests, and confirms that, regardless of how nondeterminism in the transition rules is resolved, the model ends up in an expected final state and that no coherence violations occur in this or any intermediate states. We illustrate this by describing two such litmus tests.

- **Litmus test: `clean_evict_test`.** Table 1 illustrates this litmus test, showing the sequence of transitions

that our model makes starting from an initial state. Intuitively, this test confirms that an eviction from a clean cache ends successfully. In the initial state, both devices are in the S state, with device 1 having multiple evictions as instructions. The transitions show that the first `Evict` first triggers a CleanEvict in the D2HReq1 channel, and causes a downgrade to the SI^A state. Then the request is processed by the host. The host sends a GO_WritePullDrop message, which marks the overall state of all caches as S since another device also has a copy at that point. Finally, the device receives the GO_WritePullDrop message, and downgrades to I state. The `Evict` instruction is removed from the instruction list (DProg1). Subsequent `Evicts` have no effect on DCache1 because it is already invalid.

- **Litmus test: `dirty_evict_test`.** Similarly, Table 2 illustrates the `dirty_evict_test` litmus test. This test involves a sequence with a writer issuing a DirtyEvict, to which the host responds with a GO_WritePull. This triggers a writeback from the eviction device, which the host copies in, marking the completion of this operation.

Our GitHub repository [31] includes 8 litmus tests that cover scenarios such as a read and a write being issued concurrently by two devices, multiple reads, multiple writes and multiple evicts, and alternating reads, writes and evicts. We have evaluated all intermediate states in the execution traces, ensuring that tests complete successfully, maintaining a coherent state throughout.

5.2 Assessing the `CXL.cache` restrictions

Recall from Section 3.3 that `CXL.cache` imposes various restrictions on implementations, such as the 'Snoop-pushes-GO' rule that we discussed in relation to the SharedSnpInv1 transition rule (Figure 4). These restrictions are formalised as predicates on system states that appear in the guards of various transition rules.

Because such restrictions place constraints on implementations of `CXL.cache`, one would reasonably expect that each of these restrictions is *necessary*—i.e. that removing a restriction would compromise the correctness of the protocol. We show that scenario verification using our Isabelle model can confirm this: that if a particular restriction is relaxed, additional states become reachable, and coherence violations can be observed. This helps to establish confidence that CXL is not imposing restrictions unnecessarily.

We illustrate this for one such restriction.

***Restriction test:* `snoop_pushes_go_test`.** Table 3 shows how a coherence violation can be reached if the rule from section 3.3 that SnpInv messages cannot overtake GO messages is relaxed. The violation recreates a scenario that was explained in the form of a message-sequence chart in a CXL webinar [33].

In the initial state, both devices' cachelines are invalid, and program 1 has a pending write and program 2 a pending read. Both devices start requests, and device 2's RdShared gets processed first, causing the host to send a (GO, S) message and the associated data. Before device 2 takes these two messages, device 1's RdOwn gets processed, causing the host to send a SnpInv to the device, invalidating its cacheline.

ISADSnpInv2(⚠)
guards: DCache2.State = IS^{AD}
head(H2DReq2) = (SnpInv, *txid*)
~~H2DRsp2 = []~~
actions: H2DReq2 := *tail*(H2DReq2)
D2HRsp2 := D2HRsp2@[(RspIHitI, *txid*)]
DBuffer2 := (SnpInv, *txid*)

The modified ISADSnpInv2 (⚠) rule above allows a snoop to be processed before the H2DRsp2 queue is empty. The rest of the steps are just the host forwarding the response to device 1, and both devices taking the GO and data messages. Observe that in the final row of Table 3 both devices hold their cachelines in the M state, violating coherence. In our correct model, DCache2 would not take the snoop until it has received the GO message and upgraded to IS^D.

6 Proving the SWMR property

In this section, we present our proof that our model satisfies the Single-Writer-Multiple-Reader (SWMR) property.

The proof as a whole consists of 73 theory files totalling around 211k lines of code. Most of these lines are taken up by 68 giant rule lemmas, each lemma taking up about 2.5k lines of code with its 796 subgoals. It took us about 12 person-months to reach this. Most of the code has been generated by our `super_sketch` tool (as we explain in Section 7), and only the definitions are purely handwritten, taking up less than ten thousand lines of code. It takes approximately 1–2 minutes to check each rule file, and 3–5 hours to build a session consisting of all rule files on an Intel Core i9-14900HX running at 2.20 GHz.

The SWMR property states that if one device has write access (M) to a location, then no other device can have read access (S) or write access to the same location.

Definition 6.1 (SWMR)**.**

$$\wedge_{i \neq j} \neg \big(\mathsf{DCache}_i.\mathsf{State} = \mathsf{M} \wedge \mathsf{DCache}_j.\mathsf{State} \in \{\mathsf{S}, \mathsf{M}\}\big)$$

Let us write $\Sigma \rightarrow \Sigma'$ if state Σ can evolve in one step to state Σ' via any of our transition rules. Unfortunately SWMR is not inductive; that is, the following does not hold:

$$\text{If } \Sigma \rightarrow \Sigma' \text{ and } \mathrm{SWMR}(\Sigma) \text{ then } \mathrm{SWMR}(\Sigma').$$

Table 1. A transition sequence witnessing `clean_evict_test`, a clean eviction from device 1.

transition rule	DProg1	DCache1	D2HReq1	H2DRsp1	HCache	DCache2	Counter
(initial state)	[Evict, Evict]	(0, S)	[]	[]	(0, S)	(0, S)	0
SharedEvict1	[Evict, Evict]	(0, SI^A)	[(CleanEvict, 1)]	[]	(0, S)	(0, S)	1
Shared_CleanEvict_NotLastDrop1	[Evict, Evict]	(0, SI^A)	[]	[(GO_WritePullDrop, 1)]	(0, S)	(0, S)	1
SIA_GO_WritePullDrop1	[Evict]	(0, I)	[]	[]	(0, S)	(0, S)	1
SIA_GO_WritePullDrop1	[Evict]	(0, I)	[]	[]	(0, S)	(0, S)	1

Table 2. A transition sequence witnessing `dirty_evict_test`, a writeback triggered by GO_WritePull.

transition rule	DProg1	DCache1	D2HReq1	D2HRsp1	H2DData1	HCache	DCache2	Counter
(initial state)	[Evict]	(1, M)	[]	[]	[]	(0, M)	(0, I)	0
ModifiedEvict1	[Evict]	(0, MI^A)	[(DirtyEvict, 1)]	[]	[]	(0, M)	(0, I)	1
HostModifiedDirtyEvict1	[Evict]	(0, MI^A)	[]	[(GO_WritePull, 1)]	[]	(0, I^D)	(0, I)	1
MIAGO_WritePull1	[]	(1, I)	[]	[]	[(Data, 1)]	(1, I^D)	(0, I)	1
IDData1	[]	(1, I)	[]	[]	[]	(1, I)	(0, I)	1

Table 3. A transition sequence witnessing `snoop_pushes_go_test`, leading to an incoherent state if rule ISADSnpInv2 is broken. In each row, DProg1 = [Store] and DProg2 = [Load].

transition rule	DCache1	D2HReq1	H2DRsp1	H2DData1	HCache	D2HReq2	D2HRsp2	H2DReq2	H2DRsp2	H2DData2	DCache2	Counter
(initial state)	(−1, I)	[]	[]	[]	(0, I)	[]	[]	[]	[]	[]	(−1, I)	0
InvalidStore1	(−1, IM^{AD})	[(RdOwn, 0)]	[]	[]	(42, I)	[]	[]	[]	[]	[]	(−1, I)	1
InvalidLoad2	(−1, IM^{AD})	[(RdOwn, 0)]	[]	[]	(42, I)	[(RdShared, 1)]	[]	[]	[]	[]	(−1, IS^{AD})	2
InvalidRdShared2	(−1, IM^{AD})	[(RdOwn, 0)]	[]	[]	(42, S)	[]	[]	[]	[(GO, S, 1)]	[(*Data*(42), 1)]	(−1, IS^{AD})	2
SharedRdOwn1	(−1, IM^{AD})	[]	[]	[(*Data*(42), 0)]	(42, M^A)	[]	[]	[(SnpInv, 0)]	[(GO, S, 1)]	[(*Data*(42), 1)]	(−1, IS^{AD})	2
ISADSnpInv2(o)	(−1, IM^{AD})	[]	[]	[(*Data*(42), 0)]	(42, M^A)	[]	[(RspIHitI, 0)]	[]	[(GO, S, 1)]	[(*Data*(42), 1)]	(−1, IS^{AD})	2
ISADGO+Data2	(−1, IM^{AD})	[]	[]	[(*Data*(42), 0)]	(42, M^A)	[]	[(RspIHitI, 0)]	[]	[]	[]	(42, S)	2
MARspIHitI1	(−1, IM^{AD})	[]	[(GO, M, 0)]	[(*Data*(42), 0)]	(42, M)	[]	[]	[]	[]	[]	(42, S)	2
IMADGO+Data1	(42, M)	[]	[]	[]	(42, M)	[]	[]	[]	[]	[]	(42, S)	2

A straightforward counterexample is a state that is about to become incoherent, such as:

$$\begin{aligned}&(\!|\ \mathsf{DCache1} = (\!|0, \mathsf{IM}^\mathsf{A}|\!),\\ &\ \ \mathsf{H2DRsp1} = [(\!|\mathsf{GO}, \mathsf{M}, \mathit{txid}|\!)],\\ &\ \ \mathsf{DCache2} = (\!|0, \mathsf{M}|\!)\ |\!)\end{aligned}$$

However, this state is not reachable from any valid initial state. We need a stronger property than SWMR to rule out erroneous and unreachable states like this one. That is, we require an invariant `inv` such that:

- If `initial_state`(Σ) then `inv`(Σ).
- If $\Sigma \to \Sigma'$ and `inv`(Σ) then `inv`(Σ').
- If `inv`(Σ) then SWMR(Σ).

With this invariant in-hand, we can show that our system indeed satisfies the SWMR property (writing $\to^*$ for a sequence of zero or more transitions):

Theorem 6.2 (SWMR_CXL_cache). *Assume that* $\Sigma \to^* \Sigma'$ *and* `initial_state`(Σ). *Then* SWMR(Σ').

The process of obtaining the invariant that enables this proof required a few dozen iterations to converge. We started with SWMR and then successively added conjuncts to rule out erroneous and unreachable states as they became apparent. Whenever we added a conjunct, we sought to make it as simple and general as possible, in order to rule out as many bad states as possible in one go, while not excluding any reachable states.

We now present four of the conjuncts of `inv` to give the reader a flavour of the entire invariant.

Transient states need similar SWMR constraints. The following conjunct of our invariant:

$$\left(\begin{array}{l}\mathsf{DCache1.State} \in \{\mathsf{IM}^\mathsf{D}, \mathsf{SM}^\mathsf{D}\} \vee\\ \mathsf{DCache1.State} \in \{\mathsf{IM}^\mathsf{AD}, \mathsf{SM}^\mathsf{AD}\} \wedge \mathsf{H2DRsp1} \neq []\end{array}\right) \implies$$

$$\mathit{head}(\mathsf{H2DReq2}) \neq (\mathsf{SnpInv}, \_) \implies$$

$$\left(\begin{array}{l}\mathsf{DCache2.State} \notin \{\mathsf{IS}^\mathsf{D}, \mathsf{IM}^\mathsf{D}, \mathsf{SM}^\mathsf{D}, \mathsf{IS}^\mathsf{A},\\ \qquad\qquad \mathsf{IM}^\mathsf{A}, \mathsf{SM}^\mathsf{A}, \mathsf{S}, \mathsf{M}\} \wedge\\ \mathsf{H2DData2} = [] \wedge\\ \big(\mathsf{DCache2.State} \notin \{\mathsf{IS}^\mathsf{AD}, \mathsf{IM}^\mathsf{AD}, \mathsf{SM}^\mathsf{AD}\} \vee\\ \quad \mathsf{H2DRsp2} = []\big)\end{array}\right)$$

says that if device 1 has almost upgraded to the M state, and is just awaiting an acknowledgement, then the other device must not be in any valid (or *about to be* valid) states, unless a SnpInv is on its way to invalidate that valid cache.

Snoop responses need to be honest. If a device responds to a snoop that it has invalidated its cacheline, then it must, unsurprisingly, be in an invalid state:

$$\mathit{head}(\mathsf{D2HRsp1}) \in \{(\mathsf{RspIFwdM}, \_), (\mathsf{RspIHitSE}, \_)\} \implies$$
$$\mathsf{DCache1.State} \in \{\mathsf{I}, \mathsf{IS}^{\mathsf{DI}}, \mathsf{IS}^{\mathsf{AD}}, \mathsf{IM}^{\mathsf{AD}}, \mathsf{II}^{\mathsf{A}}\}$$

Channels are singleton lists. As a result of our restriction to a single location, it is the case that each channel can contain at most one message at any given time:

$$\mathit{length}(\mathsf{H2DReq1}) \leqslant 1 \wedge \mathit{length}(\mathsf{H2DReq2}) \leqslant 1 \wedge$$
$$\mathit{length}(\mathsf{H2DRsp1}) \leqslant 1 \wedge \mathit{length}(\mathsf{H2DRsp2}) \leqslant 1 \wedge$$
$$\mathit{length}(\mathsf{H2DData1}) \leqslant 1 \wedge \mathit{length}(\mathsf{H2DData2}) \leqslant 1 \wedge \ldots$$

Host and device data channels must not conflict. This is a stronger restriction than the previous conjunct. It says that each of the different data message channels $\mathsf{H2DData}_i$ and $\mathsf{D2HData}_j$ has at most one data message pending:

$$i \neq j \implies (\mathsf{D2HData}_i = [] \vee \mathsf{H2DData}_j = [])$$

7 Better automation for large proofs

The difficulty associated working with large proofs of properties of computer systems is well known, and has been discussed e.g. in the context of the IronFleet project on proving correctness properties of distributed systems [16], and the L4.verified project on verify an OS microkernel [4]. Proof scalability was a key challenge that we faced in working towards our proof of the SWMR property for our model of `CXL.cache`. This is because deriving an inductive invariant that implied the SWMR property required many iterations of proof attempts, with each iteration taking a significant amount of human and machine time, and the time required increasing as the invariant grew.

We now outline the iterative process that we used to work towards an inductive invariant, explaining why this process was difficult and time-consuming (Section 7.1). We believe our report on this experience will be valuable for researchers interested in embarking on a formal verification project who do not yet have experience working on large inductive proofs.

We then describe a simple Isabelle utility, `super_sketch`, which we have created to accelerate the iterative development of inductive invariants (Section 7.2). This contribution is targeted more specifically at researchers intending to use the Isabelle prover for their verification efforts.

7.1 The challenge of iterative inductive invariant development

Recall from Section 6 that our proof of the SWMR property hinges on an inductive invariant, `inv`. In practice, it was relatively easy to prove that the SWMR property was implied by `inv` and that all initial states of the system satisfied `inv`. Much more challenging was to prove that `inv` was actually inductive—i.e. that every successor of a state satisfying `inv` also satisfies `inv`.

Viewing `inv` as a conjunction of sub-invariants, so that $\mathtt{inv}(\Sigma) = \mathtt{inv}_1(\Sigma) \wedge \mathtt{inv}_2(\Sigma) \wedge \ldots \mathtt{inv}_n(\Sigma)$, we can treat the proofs we need to do to show the inductiveness of `inv` as an $n \times m$ matrix, where n is the number of conjuncts and m is the number of transition rules. Cell (i, j) of this matrix represents the obligation to prove that $\mathtt{inv}(\Sigma) \implies \mathtt{inv}_i(\Sigma')$ whenever the transition $\Sigma \to \Sigma'$ is enabled by rule j (we shall write $\Sigma \xrightarrow{j} \Sigma'$ for this). Demonstrating inductiveness involves generating proofs for all cells.

When we find that the proof for a cell (i, j) does *not* go through—i.e. we cannot prove that $\mathtt{inv}(\Sigma) \implies \mathtt{inv}_i(\Sigma')$ holds for $\Sigma \xrightarrow{j} \Sigma'$—we *strengthen* the invariant: we devise a new conjunct $\mathtt{inv}_{n+1}$ such that we *are* able to prove that $\mathtt{inv}(\Sigma) \wedge \mathtt{inv}_{n+1}(\Sigma) \implies \mathtt{inv}_i(\Sigma')$ holds for $\Sigma \xrightarrow{j} \Sigma'$.

Let $\mathtt{inv}'$ denote the strengthened invariant we get by conjoining $\mathtt{inv}_{n+1}$ to `inv`. Having added this new conjunct means we must now prove that $\mathtt{inv}'(\Sigma) \implies \mathtt{inv}_{n+1}(\Sigma')$ for all $\Sigma \to \Sigma'$, adding a new row consisting of cells $(n+1, 1), (n+1, 2), \ldots, (n+1, m)$ for rules 1 to m in our proof obligation matrix. However, there is no guarantee that proofs for these new cells (other than for the $(n+1, j)$ cell) will go through. If the proof for one such cell fails to go through we might add a new conjunct $\mathtt{inv}_{n+2}$ in response, introducing another row of proof obligations, which in turn may necessitate further conjuncts, and so on.

An even more problematic scenario is when we need to change or delete a conjunct $\mathtt{inv}_i$ from `inv` because it turned out to be incorrectly excluding valid states. In that case, we must invalidate not just the row i, but also any proofs that may depend on $\mathtt{inv}_i$. Because we cannot be sure which proofs these are, it is necessary to re-check proofs for the entire matrix of proof obligations.

In practice, this iterative development proved to be very expensive both in terms of the machine time required to search for proofs and re-check existing proofs, and the human effort required when working with the proof assistant to coordinate this process.

7.2 The `super_sketch` tool

When manually writing the rule lemmas for the obligation matrix discussed above, we observed that the proof obligations for most cells of the matrix were relatively simple to prove individually using Isabelle's automated proof-generation tool, *sledgehammer*. In our proof, we usually invoke dozens of sledgehammer calls simultaneously. Sledgehammer is a rather expensive command: it encodes the current goal into solver inputs and invoke many instances of various solvers concurrently to maximize the chance of finding a proof quickly. This means that one sledgehammer call can result in hundreds of automatic prover and SMT solver queries, and take several seconds to a minute to terminate.

```
lemma
  assumes "goal1 ∧ goal2 ∧ goal3 ∧ goal4"
  shows "goal1′ ∧ goal2′ ∧ goal3′ ∧ goal4′"
proof -
show ?thesis
  proof (intro conjI)
    show g1p: "goal1′" Proof 1 using smt queries
  next
    show g2p: "goal2′" Proof 2 using metis
  next
    show g3p: "goal3′" Proof 3 using smt queries
  next
    show g4p: "goal4′" Proof 4 using smt queries
  qed
qed
```

Figure 6. An example of our `super_sketch` utility

Once a proof is found, a manual click is needed to adopt sledgehammer's proof into the script. This needs to be done with care; for instance, we must adopt the proof from bottom to top, to prevent the insertion of earlier calls invalidating the results of later ones.

An easy automation step was to redirect sledgehammer's output to a file and then use a Python script to insert generated proofs into our overall proof script. Each lemma took 30-60 minutes to generate the proofs, but at least the process is automatic. This allowed us to scale our invariant up to about 300 conjuncts.

Still, we faced challenges when a simple sledgehammer call turned out to be insufficient for discharging a subgoal. In our proof we have around a dozen rules that require an initial case analysis to become tractable for Isabelle. In that case, we might need to split subgoals into sub-subgoals and make sledgehammer work on those. Making this automatic via external scripting at this nested depth of subgoal is rather clumsy and error-prone, and is very fragile under changes to the `inv` invariant.

In response, we developed a tool, `super_sketch`, which breaks down a goal into (possibly) multiple subgoals using a method supplied by the user, concurrently calls sledgehammer on each of subgoal with several user-supplied heuristics, and finally generates a complete proof script with all the generated sub-proofs filled in. This utility is based on Haftmann's `Sketch` tool [15]. `Sketch` generates a proof skeleton that shows what needs to be proven for each subgoal (the blue text in Figure 6), leaving out the actual proofs for the user to manually put in. What `super_sketch` does in addition is that it invokes sledgehammer to search for proofs for the user (highlighted in pink in Figure 6).

In the case where a subgoal cannot be solved automatically, `super_sketch` emits a `sorry` to indicate that no proof was found, in which case human intervention is required. In our setting this happened less than 1% of the time. This tool allowed us to continue refining the inductive invariant from 300 conjuncts to almost 800, so that it finally converged.

Although developed in response to our particular use case, `super_sketch` can be applied more generally to prove Isabelle lemmas and theorems that can be efficiently broken down into subgoals that can be handled by automated theorem provers. The idea on which `super_sketch` is based—closing the loop between a proof assistant (Isabelle in our setting) and a proof search tactic (sledgehammer in our setting)—could be applied in the context of other proof assistants.

We provide more details about the design and implementation of `super_sketch` in a separate paper [32].

8 Assumptions and limitations

We summarise the assumptions made by our work and corresponding limitations of our proof and modelling effort, most of which have been discussed earlier in the paper.

Restriction to coherence and the SWMR property. Our proof efforts have focused on the SWMR property—a key part of proving coherence. We do not consider other properties such as deadlock freedom and liveness properties. Because our focus is on coherence, we have not needed to model silent upgrades from the E to M state, and have hence collapsed these states together.

Two devices, one location. As discussed in Section 3, our model considers two devices and one location. Restricting to a single location is standard practice when reasoning about the SWMR property, which was our goal, but other properties such as deadlock-freedom and liveness would require consideration of multiple locations [21]. By restricting to two devices, certain scenarios are excluded such as invalidating multiple sharers and waiting for all their acknowledgements for an ownership-obtaining request. A few rules and a fraction of the conjuncts in our inductive invariant rely on there being just two devices, e.g. if a device is upgrading to the M state, it can be immediately granted ownership if the other device's cache is in the I state. It should be straightforward to extend the model to cater for more devices, which would immediately allow more elaborate scenario verification (see Section 5). However, adapting our proof to this setting would require suitable abstraction and generalisation efforts.

A restricted set of **`CXL.cache`** ***messages.*** As discussed in Section 3.2, our model omits certain device-to-host requests that form part of the `CXL.cache` protocol: RdCurr, RdAny, RdOwnNoData, ItoMWr, WrCur, CLFlush, WOWrInv, WOWrInvF, WrInv, and CacheFlushed [9, §3.2.4.2.5]. We exclude these because they are not normally part of a cache-coherence protocol. RdCurr simply checks the most up-to-date data value and does not change cache state (and coherence). RdAny's functionality can be achieved by RdOwn and RdShared already. RdOwnNoData is no different to RdOwn from the perspective of the SWMR property. WrCur does not

affect coherence. ItoMWr, CLFlush, WOWrInv, WOWrInvF and WrInv all require modelling more levels of memory hierarchies, and are left for future work.

Tracking mechanisms. The host and devices sometimes need information about other system components to determine how to respond to messages. For instance, in some cases the host needs to check whether the currently-evicting device is the last sharer in the system. Such tracking is often achieved via bit-vectors in a physical implementation. Our model assumes that the host does perfect tracking as if it can look at the state of the device caches (which would not be feasible in practice for efficiency reasons). Currently 14 rules rely on this "perfect tracking" assumption; details are in the `PerfectTrackingRules.txt` document in our GitHub repository [31].

Additional validity threats. We may have transcribed parts of the intention of the CXL standard incorrectly into Isabelle; however, we have mitigated this risk by carefully justifying where in the standard each of our modelling decisions comes from, and by consulting several experts on the CXL committee when we had doubts. Finally, although a mechanised proof in a proof assistant such as Isabelle is considered the gold standard of correctness, it is always a remote possibility that software bugs in Isabelle or hardware bugs in the machine running it could undermine its guarantees.

9 Related and future work

There have been several efforts to reason about a variety of cache-coherence protocols over the years [27, 29]. Oswald et al. have developed a domain-specific language called ProtoGen [23], which automatically generates and verifies a cache-coherence protocol given a stable-state specification. It would be interesting to try to use their tool to specify `CXL.cache` and generate a complete protocol and then compare our model to that. Oswald et al. have also developed HeteroGen [24], which, like CXL, seeks to bridge heterogeneous systems for cache coherence, but uses "proxy caches" instead; it would be interesting to compare the functionalities of a CXL host and a proxy cache. Goens et al. [11] have developed operational and axiomatic models for the memory ordering behind such heterogeneous systems. They abstract away the role of the interconnect; our work is complementary as we have modelled the interconnect in detail.

Hemiola [7] is another domain-specific language for designing (and proving the correctness of) cache-coherence protocols over ordered networks. An important innovation is its 'serializability proofs', whereby a user can prove properties about a system assuming transactions happen one by one, which can greatly reduce the number of concurrent situations to be considered. We believe that our proof could benefit from this technique once Hemiola has been extended to handle unordered networks; still, Hemiola would likely do little to reduce the complexity of our system-state invariant. However, Bourgeat hints at a way to reduce the complexity of invariants used for verifying pipelined processors [3, p. 131] by making the invariant itself inductively-defined, and his approach may be adaptable to cache-coherence protocols.

In ongoing work, Assa, Friedman, and Lahav [2] propose a model for programming on top of CXL. One of the assumptions they make is that CXL provides cache coherence. Our work can be seen as complementary to theirs in the sense that it helps to justify that assumption.

In future work, we would like to strengthen our theorem by relaxing our idealised tracking assumption. This will involve refining our inductive invariant further, and one mechanism for better managing the inevitable complexity of this would be to make the invariant more hierarchical—having more intermediate predicates between the top-level invariant and the atomic formulas. We would also like to extend our model to handle more than two devices, and to handle more than one location so that the memory *consistency* model can be investigated [21]. The sister protocol `CXL.mem`, which enables disaggregated memory [18], is a natural target for future formalisation efforts too, and being a somewhat higher-level protocol, it should be amenable to more traditional litmus testing [1].

Acknowledgements

We thank the anonymous reviewers for their valuable feedback, and Chris Hawblitzel for serving as our shepherd. This work was supported by an EPSRC Programme Grant on *Interface reasoning for interacting systems* (EP/R006865/1). We thank Martin Desharnais and Jasmin Blanchette for valuable discussions, and for pointing us to the `Sketch` tool, which inspired `super_sketch`. We thank Christian Urban for trying out `super_sketch` and providing useful feedback. We thank Dan Iorga for sharing his insights on his investigation of CXL. We thank Vijay Nagarajan for sharing his expertise in cache coherence and heterogeneous protocols. We thank Debendra Sharma, Rob Blankenship and Thibaut Palfer-Sollier for helpful discussions related to the CXL standard.

A Artifact Appendix

A.1 Abstract

This artifact consists of the formal model of the cache coherence protocol of Compute Express Link (CXL)–`CXL.cache`. It contains the model and proof of the Single-Writer-Multiple-Reader (SWMR) property of `CXL.cache` in the Isabelle/HOL proof assistant, as described in the paper "Formalising CXL Cache Coherence". The protocol is modelled as a transition system over system state, where a system state comprises cacheline states, communication channels, buffers and other auxiliary structures.

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

- **Algorithm:** `CXL.cache` Cache Coherence Protocol
- **Program:** Isabelle theories
- **Compilation:** isabelle jedit
- **Run-time environment:** Windows, with Isabelle2023 installed. Double-click the "Cygwin-Terminal" .bat file in the installation folder, and run the command (in the Execution step) from that terminal.
- **Execution:** First change into the artifact top-level directory. Then run the command "isabelle jedit -l AllFixes -J -Xmx8192m" in the artifact top-level directory.
- **Output:** messages shown on the proof state panel indicating theory files have been successfully processed.
- **How much disk space required (approximately)?:** 50GB
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes.
- **How much time is needed to complete experiments (approximately)?:** 3-10 hours (It is recommended to use a machines with at least 32GB memory to achieve this lower bound.)
- **Publicly available?:** Yes,
- **Workflow automation framework used?:** Isabelle sessions.

A.3 Description

The file that contains the definitions of the system state with type-1 devices (corresponding to the datatype definition in Figure 3 in Section 3) is `Transposed.thy` (see line 157 the record definition "`Type1State`"). Together with the record type some functions for manipulating certain fields of `Type1State` are also defined. The transition rules as shown in Figure 4 in Section 3 of the system are defined in the file `BuggyRules.thy`.

The coherence property that is shown to be an inductive invariant of the system lies in `CoherenceProperties.thy` (see line 199, definition `SWMR_state_machine`).

The `BasicInvariants.thy` file contains some basic invariants related to certain transitions and functions we already defined in `BuggyRules.thy` and `Transposed.thy`.

The proofs are in the rest of the .thy files in this artifact. Each transition rule is proven to maintain the inductive property (`SWMR_state_machine`). Since `SWMR_state_machine` is quite large (consisting of around 800 conjuncts), the proof of just a single rule is lengthy, each spanning more than 1,000 lines. They are therefore each stored in a dedicated file, where the filename corresponds to the name of the rule (up to a prefix).

As an example, the `FixSIAGO_WritePull.thy` file contains the proof that the `SIAGO_WritePull` rule maintains the `SWMR_state_machine` property. The main lemma stating this fact is at the end of the file (line 2915 with name `SIAGO_WritePull_coherent`).

The most important auxiliary lemma leading to this is `SIAGO_WritePull'_coherent_aux_simpler` (see line 233). This auxiliary lemma breaks down the proof into hundreds of subgoals. We call lemmas like this "rule lemmas" as they each correspond to a rule.

The top level theorem stating the Single-Writer-Multiple-Reader property of the transition system is the corollary named `SWMR_pplus_cache` in `TopLevelTheorem.thy` (line 354). It corresponds to Theorem 6.2 in the paper. Some main theorems lead to this corollary:

- If `initial_state`(Σ) then `SWMR_state_machine`(Σ) (Theorem `SWMR_state_machine_CXL_cache`, line 321).
- If $\Sigma \rightarrow \Sigma'$ and `SWMR_state_machine`(Σ) then `SWMR_state_machine`(Σ') (Theorem `all_transitions_coherent`, line 103).

These two theorems correspond to the first two of the three properties described in the paper just before Section 6.

A.3.1 How to access.

The artifact is available on GitHub: https://github.com/ChengsongTan/CXLcacheFormalisation

A.3.2 Software dependencies.

The artifact depends on Isabelle2023, available at: https://isabelle.in.tum.de/website-Isabelle2023/index.html

A.4 Installation

See the "Running experiment" section on GitHub: https://github.com/ChengsongTan/CXLcacheFormalisation?tab=readme-ov-file#running-experiment

A.5 Evaluation and expected results

See the "Expected results" section on GitHub: https://github.com/ChengsongTan/CXLcacheFormalisation?tab=readme-ov-file#expected-results

References

[1] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 41–44. Springer, 2011.

[2] Gal Assa, Michal Friedman, and Ori Lahav. A Programming Model for Disaggregated Memory over CXL, 2024. https://arxiv.org/pdf/2407.16300.

[3] Thomas Bourgeat. *Specification and verification of sequential machines in rule-based hardware languages.* PhD thesis, MIT, 2023. .

[4] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. Challenges and experiences in managing large-scale proofs. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 2012.

[5] CCIX Consortium. CCIX Consortium Enables Next Generation Compute Architectures with the Availability of Base Specification 1.0, 2018. https://bwnews.pr/4ePcMLM.

[6] Richard Chirgwin. Intel's Omni-Path InfiniBand-killer debuts at sizzling 100 Gb/sec, 2015. https://bit.ly/omnipath.

[7] Joonwon Choi, Adam Chlipala, and Arvind. Hemiola: A DSL and verification tools to guide design and proof of hierarchical cache-coherence protocols. In *CAV (2)*, volume 13372 of *Lecture Notes in Computer Science*, pages 317–339. Springer, 2022.

[8] Ian Cutress. CXL Specification 1.0 Released: New Industry High-Speed Interconnect From Intel, 2019. https://bit.ly/cxl-spec.

[9] CXL Consortium. Compute Express Link Specification, Revision 3.1, 2023. https://bit.ly/cxl31.

[10] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An Introduction to the Compute Express Link (CXL) Interconnect, 2023. https://arxiv.org/pdf/2306.11227.

[11] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. Compound memory models. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[12] Hiroshige Goto. AMD's Infinity Fabric will be the foundation for all of its chips from 2017 onwards, 2017. https://bit.ly/infinityfabric.

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

[14] Thibault Grossi. Memory Processor Interface 2023, Focus on CXL, 2024. https://bit.ly/yole_cxl.

[15] Florian Haftmann. The Sketch and Explore library, 2023. https://bit.ly/sketch_explore.

[16] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael Lowell Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17. ACM, 2015.

[17] Intel. Intel Ultra Path Interconnect, 2020. https://bit.ly/ultrapath.

[18] Intel. Orchestrating memory disaggregation with Compute Express Link, 2024. https://intel.ly/48j1JIv.

[19] 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. 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*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[20] Mellanox Technologies Inc. Introduction to InfiniBand, 2003. https://bit.ly/infiniband.

[21] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[22] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[23] Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications. In *ISCA*, pages 247–260. IEEE Computer Society, 2018.

[24] Nicolai Oswald, Vijay Nagarajan, Daniel J. Sorin, Vasilis Gavrielatos, Theo Olausson, and Reece Carr. HeteroGen: Automatic Synthesis of Heterogeneous Cache Coherence Protocols. In *HPCA*, pages 756–771. IEEE, 2022.

[25] Lawrence C. Paulson. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In *PAAR@IJCAR*, volume 9 of *EPiC Series in Computing*, pages 1–10. EasyChair, 2010.

[26] PCIE Consortium. PCIE specification library, 2023. https://pcisig.com/specifications.

[27] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Comput. Surv.*, 29(1):82–126, March 1997.

[28] Agam Shah. Hardware makers unite to challenge Intel with Gen-Z spec, 2016. https://bit.ly/gen-z-spec.

[29] Joseph E. Stoy, Xiaowei Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 43–71. Springer, 2001.

[30] Jeffrey Stuecheli, William J. Starke, John D. Irish, L. Baba Arimilli, Daniel M. Dreps, Bart Blaner, Curt Wollbrink, and Brian Allison. IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI. *IBM J. Res. Dev.*, 62(4/5):8:1–8:8, 2018.

[31] Chengsong Tan. GitHub repository for formalisation of CXL.cache, 2025. Accessed: 2025-02-05.

[32] Chengsong Tan, Alastair F. Donaldson, Jonathan Julián Huerta y Munive, and John Wickerson. The burden of proof: Automated tooling for rapid iteration on large mechanised proofs. In *FormaliSE@ICSE*. ACM, 2025.

[33] Siamak Tavallaei, Kurt Lender, and Robert Blankenship. Compute Express Link (CXL): Exploring Coherent Memory and Innovative Use Cases. https://bit.ly/cxlwebinar, 2020.

[34] Jon Worrel. Nvidia NVLINK 2.0 arrives in IBM servers next year, 2016. https://bit.ly/nvlink.