Specifying and Verifying Persistent Libraries

Léo Stefanesco
MPI-SWS

Azalea Raad
Imperial College

Viktor Vafeiadis
MPI-SWS

Abstract

We present a general framework for specifying and verifying *persistent libraries*, that is, libraries of data structures that provide some persistency guarantees upon a failure of the machine they are executing on. Our framework enables modular reasoning about the correctness of individual libraries (horizontal and vertical compositionality) and is general enough to encompass all existing persistent library speci cations ranging from hardware architectural speci cations to correctness conditions such as durable linearizability. As case studies, we specify the FliT and Mirror libraries, verify their implementations over Px86, and use them to build higher-level durably linearizable libraries, all within our framework. We also specify and verify a persistent transaction library that highlights some of the technical challenges which are speci c to persistent memory compared to weak memory and how they are handled by our framework.

2012 ACM Subject Classi cation Theory of computation → Program speci cations

Keywords and phrases persistent memory, concurrent libraries, persistent programming, speci cation, veri cation, weak memory models

1 Introduction

Persistent memory (PM), also known as non-volatile memory (NVM), is a new kind of memory, which can be used to extend the capacity of regular RAM, with the added bene t that its contents are preserved after a crash (e.g. a power failure). Employing PM can boost the performance of any program with access to data that needs to survive power failures, be it a complex database or a plain text editor.

Nevertheless, doing so is far from trivial. Data stored in PM is mediated through the processors caching hierarchy, which generally does not propagate all memory accesses to the PM in the order issued by the processor, but rather performs these accesses on the cache and only propagates them to the memory asynchronously when necessary (i.e. upon a cache miss or when the cache has reached its capacity limit). Caches, moreover, do not preserve their contents upon a power failure, which results in rather complex persistency models describing when and how stores issued by a program are guaranteed to survive a power failure. To ensure correctness of their implementations, programmers have to use low-level primitives, such as *ushes* of individual cache lines, *fences* that enforce ordering of instructions, and *non-temporal stores* that bypass the cache hierarchy.

These primitives are often used to implement higher-level abstractions, packaged into *persistent libraries*, i.e. collections of data structures that must guarantee to preserve their contents after a power failure. Persistent libraries can be thought of as the analogue of concurrent libraries for persistency. And just as concurrent libraries require a speci cation, so do persistent libraries.

The question naturally arises: what is the right speci cation for persistent libraries? Prior work has suggested a number of candidate de nitions, such as *durable linearizability*, *bu ered durable linearizability* [16], and *strict linearizability* [1], which are all extensions of

the well-known correctness condition for concurrent data structures (i.e. linearizability [14]). In general, these de nitions stipulate the existence of a total order among all executed library operations, a contiguous pre x of which is persisted upon a crash: the various de nitions di er in exactly what this pre x should be, e.g. whether it is further constrained to include all fully executed operations.

Even though these speci cations have a nice compositionality property, we argue that none of them are *the* right speci cation pattern for *every* persistent concurrent library. While for high-level persistent data structures, such as stacks and queues, a strong speci cation such as durable or strict linearizability would be most appropriate, this is certainly not the case for a collection of low-level primitives. Take, for instance, a library whose interface simply exposes the exact primitives of the underlying platform: memory accesses, fences and ushes. Their semantics, recently formalized in [28, 17, 26] in the case of the Intel-x86 architecture and in [29, 4] in the case of the ARMv8 architecture, quite clearly do not t into the framework of the durable linearizability de nitions. More generally, there are useful concurrent libraries (especially in the context of weak memory consistency) that are not linearizable [24]; it is, therefore, conceivable that making those libraries persistent will require weak speci cations.

Another key problem with attempting to specify persistent libraries *modularly* is that they often break the usual abstraction boundaries. Indeed, some models such as epoch persistency [5, 22] provide a global persistency barrier that a ects all memory locations, and therefore all libraries using them. Such global operations also occur at higher abstraction layers: persistent transactional libraries often require memory locations to be registered with the library in order for them to be used inside transactions. As such, to ensure compatibility with such transactional libraries, implementers of other libraries must register all locations they use and ensure that any component libraries they use do the same.

In this paper, we introduce a *general declarative framework* that addresses both of these challenges. Our framework provides a very exible way of specifying persistent libraries, allowing each library to have a very di erent speci cation be it durable linearizability or a more complex speci cation in the style of the hardware architecture persistency models. Further, to handle libraries that have a global e ect (such as persistent barriers above) or, more generally, that make some assumptions about the internals of all other libraries, we introduce a *tag* system, allowing us to describe these assumptions *modularly*.

Our framework supports both *horizontal* and *vertical compositionality*. That is, we can verify an execution containing multiple libraries by verifying each library separately (horizontal compositionality). Moreover, we can completely verify the implementation of a library over a set of other libraries using the speci cations of its constituent libraries without referring to their implementations (vertical compositionality). To achieve the latter, we de ne a semantic notion of substitution in terms of execution graphs, which replaces each library node by a suitably constrained set of nodes (its implementation).

For simplicity, in 2, we develop a rst version of our framework over the classical notion of an execution *history* [14], which we extend with a notion of crashes. This basic version of our framework includes full support for weak persistency models but assumes an interleaving semantics of concurrency; i.e. sequential consistency (SC) [21].

Subsequently, in 3 we generalise and extend our framework to handle weak consistency models such as x86-TSO [30] and RC11 [20], thereby allowing us to represent hardware persistency models such as Px86 [28] and PARMv8 [29], in our framework. To do so, we rebase our formal development over execution graphs using Yacovet [24] as a means of specifying the consistency properties of concurrent libraries.

We illustrate the utility of our framework by encoding in it a number of existing persistency models, ranging from actual hardware models such as PxA86 [28], to general-purpose correctness conditions such as durable linearizability [16]. We further consider two case studies, chosen to demonstrate the expressiveness of our framework beyond the kind of case studies that have been worked out in the consistency setting.

First, in 4 we use our framework to develop the rst formal speci cations of the FliT [31] and Mirror [9] libraries and establish the correctness of not only their implementations against their respective speci cations, but also their associated constructions for turning a linearizable library into a durably linearizable one. This generic theorem is new compared to the case studies in [24], and leverages our semantic approach in 3. Moreover, our proofs of these constructions are the *rst* to establish this result in a weak consistency setting.

Second, in 5 we specify and prove an implementation of a persistent transactional library $\mathsf{L}_{\mathsf{trans}}$, which provides a high-level construction to persist a set of writes *atomically*. The $\mathsf{L}_{\mathsf{trans}}$ library illustrates the need for a well-formedness speci cation (in addition to its consistency and persistency speci cations) that requires clients of the $\mathsf{L}_{\mathsf{trans}}$ library to ensure e.g. that $\mathsf{L}_{\mathsf{trans}}$ writes appear only inside transactions. Moreover, it demonstrates the use of our tagging system to enable other libraries to interoperate with it.

Contributions and Outline. The remainder of this article is organised as follows.

2 We present our general framework for specifying and verifying persistent libraries in the strong sequential consistency setting.
3 We further generalise our framework to account for weaker consistency models.
4 We use our framework to develop the rst formal speci cations of the FliT and Mirror libraries, verify their implementations against their speci cations and prove their general construction theorems for turning linearizable libraries to durably linearizable ones.
5 We specify a persistent transactional library $\mathsf{L}_{\mathsf{trans}}$, develop an implementation of $\mathsf{L}_{\mathsf{trans}}$ (over the Intel-x86 architecture) and verify it against its speci cation. We then consider two case studies of vertical and horizontal composition in our framework using $\mathsf{L}_{\mathsf{trans}}$.

We conclude and discuss related and future work in 6. The full proofs of all theorems stated in the paper are given in the technical appendix.

2 A General Framework for Persistency

We present our framework for specifying and verifying persistent *libraries*, which are collections of methods that operate on durable data structures. Following Herlihy et al. [14], we will represent program histories over a collection of libraries Λ as Λ-histories, i.e. as sequences of calls to the methods of Λ, which we will then gradually enhance to model persistency semantics. Throughout this section, we assume an underlying sequential consistency semantics; in 3 we will generalize our framework to account for weaker consistency models.

In the following, we assume the following in nite domains: **Meth** of method names, **Loc** of memory locations, **Tid** of thread identi ers, and $\mathbf{Val} \supseteq \mathbf{Loc} \cup \mathbf{Tid}$ of values. We let m range over method names, x over memory locations, t over thread identi ers, and v over values. An optional value $v_\perp \in \mathbf{Val}_\perp$ is either a value $v \in \mathbf{Val}$ or $\perp \notin \mathbf{Val}$.

2.1 Library Interfaces

A *library interface* declares a set of method invocations of the form $m(\vec{v})$. Some methods are are designated as constructors; a constructor returns a location pointing to the new library

instance (object), which is passed as an argument to other library methods. An interface additionally contains a function, loc, which extracts these locations from the arguments and return values of its method calls.

▶ **De nition 2.1.** *A* library interface L *is a tuple* $\langle \mathcal{M}, \mathcal{M}_c, \mathrm{loc} \rangle$*, where* $\mathcal{M} \subseteq \mathcal{P}(\mathbf{Meth} \times \mathbf{Val}^*)$ *is the set of method invocations,* $\mathcal{M}_c \subseteq \mathcal{M}$ *is the set of constructors, and* $\mathrm{loc} : \mathcal{M} \times \mathbf{Val}_\perp \to \mathcal{P}(\mathbf{Loc})$ *is the location function.*

▶ **Example 2.2** (Queue library interface)**.** The queue library interface, $\mathsf{L}_{\mathsf{Queue}}$, has three methods: a constructor $\mathsf{QueueNew}()$, which returns a new empty queue; $\mathsf{QueueEnq}(x, v)$ which adds value v to the end of queue x; and $\mathsf{QueueDeq}(x)$ which removes the head entry in queue x. We de ne $\mathrm{loc}(\mathsf{QueueNew}(), x) = \mathrm{loc}(\mathsf{QueueEnq}(x, \_), \_) = \mathrm{loc}(\mathsf{QueueDeq}(x), \_) = \{x\}$.

A *collection* Λ is a set of library interfaces with disjoint method names. When Λ consists of a single library interface L, we often write L instead of $\{L\}$.

2.2 Histories

Given a collection Λ, an event $e \in \mathbf{Events}(\Lambda)$ of Λ is either a method invocation $m(\vec{v})_{\mathsf{t}}$ with $m(\vec{v}) \in \bigcup_{L \in \Lambda} L.\mathcal{M}$ and $\mathsf{t} \in \mathbf{Tid}$ or method response (return) event $\mathbf{ret}(v)_{\mathsf{t}}$.

A Λ-history is a sequence of events of Λ whose projection to each thread is an alternating sequence of invocation and return events which starts with an invocation.

▶ **De nition 2.3** (Sequential event sequences)**.** *A sequence of events* $e_1 \dots e_n$ *is* sequential *if all its odd-numbered events* $e_1, e_3, \dots$ *are invocation events and all its even-numbered events* $e_2, e_4, \dots$ *are return events.*

▶ **De nition 2.4** (Histories)**.** *A* Λ-history *is a nite sequence of events* $H \in \mathbf{Events}(\Lambda)^*$*, such that for every thread* t*, the sub-sequence* $H[\mathsf{t}]$ *comprising only of* t *events is sequential. The set* $\mathbf{Hist}(\Lambda)$ *denotes the set of all* Λ*-histories.*

When clear from the context, we refer to *occurrences* of events in a history by their corresponding events. For example, if $H = e_1 \dots e_n$ and $i < j$, we say that e_i *precedes* e_j and that e_j *succeeds* e_i. Given an invocation $m(\vec{v})_{\mathsf{t}}$ in H, its *matching return* (when it exists) is the rst event of the form $\mathbf{ret}(v)_{\mathsf{t}}$ that succeeds it (they share the same thread). A *call* is a pair $m(\vec{v})_{\mathsf{t}}{:}v_\perp$ of an invocation and either its matching return $v_\perp \in \mathbf{Val}$ (*complete call*) or $v_\perp = \perp$ (*incomplete call*).

A *library* (speci cation) comprises an interface and a set of *consistent* histories. The latter captures the allowed behaviors of the library, which is a guarantee made by the library implementation.

▶ **De nition 2.5.** *A* library speci cation *(or simply a* library*)* L *is a tuple* $\langle L, \mathcal{S}_c \rangle$*, where* L *is a library interface, and* $\mathcal{S}_c \subseteq \mathbf{Hist}(L)$ *denotes its set of* consistent *histories.*

2.3 Linearizability

Linearizability [14] is a standard way of specifying concurrent libraries that have a sequential speci cation S, denoting a set of nite sequences of complete calls. Given a sequential speci cation S, a concurrent library L is linearizable under S if each consistent history of L can be *linearized* into a sequential one in S, while respecting the *happens before* order, which captures causality between calls. It is su cient to consider consistent executions because inconsistent executions are, by de nition, guaranteed by the library to never happen. Happens-before is de ned as follows.

▶ **De nition 2.6** (Happens-Before). *A method call* C_1 happens before *another method call* C_2 *in a history* H*, written* $C_1 \prec_H C_2$ *if the response of* C_1 *precedes the invocation of* C_2 *in* H*. When the choice of* H *is clear from the context, we drop the* H *subscript from* $\prec$*.*

A history H is *linearizable* under a sequential speci cation S if there exists a linearization (in the order-theoretic sense) of $\prec_H$ that belongs to S. The subtlety is the treatment of incomplete calls, which may or may not have taken e ect. We write $\mathsf{compl}(H)$ for the set of histories obtained from a history H by appending zero or more matching return events. We write $\mathsf{trunc}(H)$ for the history obtained from H by removing its incomplete calls. We can then de ne linearizability as follows [13].

▶ **De nition 2.7.** *A sequential history* H_ℓ *is a* sequentialization *of a history* H *if there exists* $H' \in \mathsf{trunc}(\mathsf{compl}(H))$ *such that* H_ℓ *is a linearization of* $\prec_{H'}$*. A history* H *is* linearizable under S *if there exists a sequentialization of* H *that belongs to* S*. A library* L *is* linearizable *under* S *if all its consistent histories are linearizable under* S*.*

For instance, we can specify the notion of *linearizable queues* as those that linearizable under the following sequential queue speci cation, S_{Queue}.

▶ **Example 2.8** (Sequential queue speci cation)**.** The behaviors of a sequential queue, S_{Queue}, is expressed as a set of sequential histories as follows. Given a history H of L_{Queue} and a location $x \in \mathbf{Loc}$, let $H[x]$ denote the sub-history containing calls c such that $\mathrm{loc}(c) = \{x\}$. We de ne S_{Queue} as the set of all sequential histories H of L_{Queue} such that for all $x \in \mathbf{Loc}$, $H[x]$ is of the form $\mathsf{QueueNew}()_{\mathsf{t}_0}{:}x\ e_1\ \cdots\ e_n$, where each $\mathsf{QueueDeq}$ call in $e_1\ \cdots\ e_n$ returns the value of the k-th $\mathsf{QueueEnq}$ call, if it exists and precedes the $\mathsf{QueueDeq}$, where k is the number of preceding $\mathsf{QueueDeq}$ calls returning non-null values; otherwise, it returns null.

2.4 Adding Failures

Our framework so far does not support reasoning about persistency as it lacks the ability to describe the persistent state of a library after a failure. Our rst extension is thus to extend the set of events of a collection, $\mathbf{Events}(\Lambda)$, with another type of event, a crash event $\lightning$.

Crash events allow us to specify the durability guarantees of a library. For instance, a library that does not persist any of its data may specify that a history with crash events is consistent if all of its sub-histories between crashes are (independently) consistent. In other words, in such a library, the method calls before a crash have no e ect on the consistency of the history after the crash. We modify the de nition of happens-before accordingly by treating it both as an invocation and a return event. We also assume that, after a crash, the thread ids of the new threads are distinct from that of all the pre-crash threads. For libraries that do persist their data, a useful generic speci cation is *durable linearizability* [16], de ned as follows.

▶ **De nition 2.9.** *Given a history* H*, let* $\mathsf{ops}(H)$ *denote the sub-history obtained from* H *by removing all its crash markers. A history* H *is* durably linearizable *under* S *if there exists a sequentialization* H_ℓ *of* $\mathsf{ops}(H)$ *such that* $H_\ell \in S$*.*

Intuitively, this ensures that operations persist before they return, and they persist in the same order as they take e ect before a crash.

Although durable linearizability can specify a large range of persistent data-structures, it can be too strong. For example, consider a (memory) register library $\mathsf{L}_{\mathsf{wreg}}$ that only guarantees that writes to the *same* location are persisted in the order they are observed

by concurrent reads. The $\mathsf{L}_{\mathsf{wreg}}$ methods comprise $\mathsf{RegNew}()$ to allocate a new register, $\mathsf{RegWrite}(x, v)$ to write v to register x, and $\mathsf{RegRead}(x)$ to read from register x. The sequential specification S_{wreg} is simple: once a register is allocated, a read R on x returns the latest value written to x, or 0 if R happens before all writes. The associated durable linearizability specification requires that writes be persisted in the linearization order; however, this is often not the case on existing hardware, e.g. in Px86 (the Intel-x86 persistency model) [28].

A more relaxed and realistic specification would consider two linearizations of the events: the standard *volatile* order lin and a *persistent* order nvo expressing the order in which events are persisted. The next sections will handle this more refined model, this paragraph only gives a quick tastes of the kind of models that are implemented by hardware. To capture the same-location guarantees, we stipulate a per-location ordering on writes that is respected by both linearizations. Specifically, we require an ordering mo of the write calls such that for all locations x: 1) restricting mo to x, written mo_x, totally orders writes to x; and 2) $\mathrm{mo}_x \subseteq \mathrm{lin}$ and $\mathrm{mo}_x \subseteq \mathrm{nvo}$. Given a history H, we can then combine these two linearizations by using lin after the last crash and nvo before.

Formally, a history H with $n-1$ crashes can be decomposed into n (crash-free) *eras*; i.e. $H = H_1 \cdot \lightning \cdots \lightning \cdot H_n$ where each H_i is crash-free. Let us write lin_i for $\mathrm{lin} \cap (H_i \times H_i)$ and so forth. We then consider k-sequentializations of the form $H_\ell^k = H_\ell^{(1)} \cdots H_\ell^{(k-1)} \cdot H_\ell^{(k)}$, where $H_\ell^{(k)}$ is a sequentialization of E_k w.r.t. lin_k and $H_\ell^{(i)}$ is a sequentialization of E_i w.r.t. nvo_i, for $i < k$. We can now specify our weak register library as follows, where H comprises n eras:

$$H \in \mathsf{L}_{\mathsf{wreg}}.\mathcal{S}_{\mathsf{c}} \iff \forall k \leq n.\ \exists H_\ell^k\ k\text{-seq. of } H.\ H_\ell^k \in S_{\mathsf{wreg}}$$

▶ **Example 2.10.** The following history is valid according to this specification but not according to the durably linearizable one:

$$W_{\mathsf{t}_1}(x,1) \cdot W_{\mathsf{t}_2}(y,1) \cdot R_{\mathsf{t}_3}(y) \cdot \mathbf{ret}_{\mathsf{t}_3}(1) \cdot R_{\mathsf{t}_3}(x) \cdot \mathbf{ret}_{\mathsf{t}_3}(0) \cdot \lightning \cdot R_{\mathsf{t}_4}(y) \cdot \mathbf{ret}_{\mathsf{t}_4}(0) \cdot R_{\mathsf{t}_4}(x) \cdot \mathbf{ret}_{\mathsf{t}_4}(1)$$

While the writes to x ($W_{\mathsf{t}_1}(x,1)$) and y ($W_{\mathsf{t}_2}(y,1)$) are executing, thread t_3 observes the new value (1) of y but the old value (0) of x; i.e. lin must order $W_{\mathsf{t}_2}(y,1)$ before $W_{\mathsf{t}_1}(x,1)$. By contrast, after the crash the new value (1) of x but the old value of y (0) is visible; i.e. nvo must order the two writes in the opposite order to lin ($W_{\mathsf{t}_1}(x,1)$ before $W_{\mathsf{t}_2}(y,1)$).

Persist Instructions. The persistent registers described above are too weak to be practical, as there is no way to control how writes to different locations are persisted. In realistic hardware models such as Px86, this control is afforded to the programmer using per-location *persist* instructions (e.g. `CLFLUSH`), ensuring that all writes on a location x persist before a write-back on x. Here, we consider a coarser (stronger) variant, denoted by PFENCE, that ensures that *all* writes (on *all* locations) that happen before a PFENCE are persisted. Later in §3 we describe how to specify the behavior of per-location persist operations.

Formally, we specify PFENCE by extending the specification of $\mathsf{L}_{\mathsf{wreg}}$ with as follows: given history H, write call c_w and PFENCE event c_f, if $c_w \prec_H c_f$, then $(c_w, c_f) \in \mathrm{nvo}$.

▶ **Example 2.11.** Consider the history obtained from Example 2.10 by adding a PFENCE:

$$W_{t_1}(x,1) \cdot W_{t_2}(y,1) \cdot R_{t_3}(y) \cdot \mathbf{ret}_{t_3}(1) \cdot R_{t_3}(x) \cdot \mathbf{ret}_{t_3}(0) \cdot \mathsf{PFENCE}_{t_4}() \cdot \mathbf{ret}_{t_4}() \cdot \lightning \cdot R_{t_4}(y) \cdot \mathbf{ret}_{t_4}(0) \cdot R_{t_4}(x) \cdot \mathbf{ret}_{t_4}(1)$$

This history is no longer consistent according to the extended specification of $\mathsf{L}_{\mathsf{wreg}}$: as PFENCE has completed (returned), all its $\prec$-previous writes must have persisted and thus must be visible after the crash (which is not the case for $W_{t_2}(y,1)$).

2.5 Adding Well-formedness Constraints

Our next extension is to allow library specifications to constrain the *usage* of the library methods by the client of the library. For example, a library for a mutual exclusion lock may require that the release lock method is only called by a thread that previously acquired the lock and has not released it in between. Another example is a transactional library, which may require that transactional read and write methods are only called within transactions, i.e. between a transaction-begin and a transaction-end method call.

We call such constraints library *well-formedness* constraints, and extend the library specifications with another component, $\mathcal{S}_{\mathsf{wf}} \subseteq \mathbf{Hist}(L)$, which records the set of well-formed histories of the library. Ensuring that a program produces only well-formed histories of a certain library is an obligation of the clients of that library, so that the library implementation can rely upon well-formedness being satisfied.

2.6 Tags and Global Specifications

The goal of our framework is not only to specify libraries in isolation, but also to express how a library can enforce persistency guarantees across other libraries. For example, consider a library $\mathsf{L}_{\mathsf{trans}}$ for persistent transactions, where all operations wrapped within a transaction persist together *atomically*; i.e. either all or none of the operations in a transaction persist.

The $\mathsf{L}_{\mathsf{trans}}$ methods are: PTNewReg to allocate a register that can be accessed (read/written) within a transaction; PTBegin and PTEnd to start and end a transaction, respectively; PTRead(x) and PTWrite(x, v) to read from and write to $\mathsf{L}_{\mathsf{trans}}$ register x, respectively; and PTRecover to restore the atomicity of transactions whose histories were interrupted by a crash.

Consider the snippet below, where the PEnq(q, 33) (enqueuing 33 into persistent queue q) and PSetAdd(s, 77) (adding 77 to persistent set s) are wrapped within an $\mathsf{L}_{\mathsf{trans}}$ transaction and thus should take effect atomically and at the latest after the end of the call to PTEnd.

```
PTBegin();
  PEnq(q, 33);
  PSetAdd(s, 77);
PTEnd();
```

Such guarantees are not offered by existing hardware primitives e.g. on Intel-x86 or ARMv8 [28, 29] architectures. As such, to ensure atomicity, the persistent queue and set implementations cannot directly use hardware reads/writes; rather, they must use those provided by the transactional library whose implementation could use e.g. an undo-log to provide atomicity.

Our framework as described so far cannot express such cross-library persistency guarantees. The difficulty is that the transactional library relies on other libraries using certain primitives. This, however, is against the spirit of *compositional specification*, which precludes the transactional library from referring to other libraries (e.g. the queue or set libraries). Specifically, there are two challenges. First, both well-formedness requirements and consistency guarantees of $\mathsf{L}_{\mathsf{trans}}$ must apply to *any* method call that is designed to use (transitively) the primitives of $\mathsf{L}_{\mathsf{trans}}$. Second, we must formally express atomicity (all operations persist atomically), without $\mathsf{L}_{\mathsf{trans}}$ knowing what it means for a method of an arbitrary library to persist. In other words, $\mathsf{L}_{\mathsf{trans}}$ needs to introduce an abstract notion of having persisted for an operation, and guarantee that all methods in a transaction persist atomically.

To remedy this, we introduce the notion of *tags*. Specifically, to address the first challenge, the transactional library provides the tag T to designate those operations that are transaction-aware and as such must be used inside a transaction. To address the second challenge, the transaction library provides the tag P^{tr}, denoting an operation that has abstractly

persisted. The specification of $\mathsf{L}_{\mathsf{trans}}$ then guarantees that all operations tagged with T inside a transaction persist atomically, in that either they are all tagged with P^{tr} of none of them are. Dually, using the well-formedness condition, $\mathsf{L}_{\mathsf{trans}}$ requires that all operations tagged with T appear inside a transaction. Note that as the persistent queue and set libraries tag their operations with T, verifying their implementations incurs related proof obligations; we will revisit this later when we formalize the notion of library implementations.

▶ Remark 2.12 (Why bespoke persistency?). The reader may question why having persisted is not a primitive notion in our framework, as in an existing model of Px86 [17] where histories track the set P of persisted events. This is because associating a Boolean (having persisted) flag with an operation may not be sufficient to describe whether it has persisted. To see this, consider a library $\mathsf{L}_{\mathsf{pair}}$ with operations $\mathsf{Write}(x, l, r)$ (writing (l, r) to pair x), $\mathsf{Readl}(x)$ and $\mathsf{Readr}(x)$ (reading the left and right components of x, respectively). Suppose $\mathsf{L}_{\mathsf{pair}}$ is implemented by storing the left component in an $\mathsf{L}_{\mathsf{trans}}$ register and the right component in a $\mathsf{L}_{\mathsf{wreg}}$ register. The specification of $\mathsf{L}_{\mathsf{pair}}$ would need to track the persistence of each component separately, and hence a single set P of persisted events would not suffice.

Let us see how libraries can use these tags in *global well-formedness and consistency specifications*. The dilemma is, on the one hand, the specification of $\mathsf{L}_{\mathsf{trans}}$ needs to refer to events from other libraries, but on the other hand, it should not depend on other libraries to preserve encapsulation. Our idea is to *anonymize* these external events such that the global specification depends only on their relevant tags. A library should only rely on the tags it introduces itself, as well as the tags of the libraries it uses.

We now revisit several of our definitions to account for *tags* and *global specifications*. A library interface now additionally holds the tags it introduces as well as those it uses. For instance, the $\mathsf{L}_{\mathsf{trans}}$ library described above depends on no tag and introduces tags T and P^{tr}.

▶ **Definition 2.13** (Interfaces). *An* interface *is a tuple* $L = \langle \mathcal{M}, \mathcal{M}_{\text{c}}, \text{loc}, \textsc{Tags}_{\text{new}}, \textsc{Tags}_{\text{dep}} \rangle$, *where* $\mathcal{M}$, $\mathcal{M}_{\text{c}}$, *and* loc *are as in Def. 2.1,* $\textsc{Tags}_{\text{new}}$ *is the set of tags* L *introduces, and* $\textsc{Tags}_{\text{dep}}$ *is the set of tags* L *uses. The set of tags usable by* L *is* $\textsc{Tags}(L) \triangleq L.\textsc{Tags}_{\text{new}} \cup L.\textsc{Tags}_{\text{dep}}$.

We next define the notion of tagged method invocations (where a method invocation is associated with a set of tags). Hereafter, our notions of events, history (and so forth) use tagged method invocations (rather than methods invocations).

▶ **Definition 2.14.** *Given a library interface* L, *a* tagged method invocation *is of the form* $m(\vec{v})^{T}_{\mathsf{t}} : v_{\perp}$, *where the new component is a set of tags* $T \subseteq \textsc{Tags}(L)$.

A *global specification* of a library interface L is a set of histories with some anonymized events. These are formalized using a designated library interface, $\star_L$ (with a single method $\star$), which can be tagged with any tag from $\textsc{Tags}(L)$.

▶ **Definition 2.15.** *Given an interface* L, *the interface* $\star_L$ *is* $\langle \{\star\}, \emptyset, \emptyset, \emptyset, \textsc{Tags}(L) \rangle$.

Now, given any history $H \in \mathbf{Hist}(\{L\} \cup \Lambda)$, let $\pi_L(H) \in \mathbf{Hist}(\{L, \star_L\})$ denote the *anonymization* of H such that each non-L event e in H labelled with a method $m(\vec{v})^{T}_{\mathsf{t}} : v_{\perp}$ of $L' \in \Lambda$ is replaced with $\star^{T}_{\mathsf{t}}$ of $\star_L$ if $T \neq \emptyset$ and is discarded otherwise. It is then straightforward to extend the notion of libraries with global specifications as follows.

▶ **Definition 2.16.** *A* library specification L *is a tuple* $\langle L, \Lambda_{\text{tags}}, \mathcal{S}_{\mathsf{c}}, \mathcal{S}_{\mathsf{wf}}, \mathcal{T}_{\mathsf{c}}, \mathcal{T}_{\mathsf{wf}} \rangle$, *where* L, $\mathcal{S}_{\mathsf{c}}$ *and* $\mathcal{S}_{\mathsf{wf}}$ *are as in Def. 2.5;* $\mathcal{T}_{\mathsf{c}}$ *and* $\mathcal{T}_{\mathsf{wf}} \subseteq \mathbf{Hist}(\{L, \star_L\})$ *are the* globally consistent *and* globally well-formed *histories, respectively; and* Λ_{tags} *denotes the* tag-dependencies, *i.e. a*

collection of libraries that provide all tags that L *uses:* $L.\mathit{TAGS}_{\mathrm{dep}} \subseteq \bigcup_{\mathsf{L}' \in \Lambda_{\mathrm{tags}}} \mathsf{L}'.\mathit{TAGS}_{\mathrm{new}}$. *Both* $\mathcal{T}_{\mathsf{wf}}$ *and* $\mathcal{T}_{\mathsf{c}}$ *contain the empty history.*

In the context of a history, we write $\lfloor \mathrm{T} \rfloor$ for the set of events or calls tagged with the tag T (we consider a return event tagged the same way as its unique matching invocation).

For the $\mathsf{L}_{\mathsf{trans}}$ library, the *globally* well-formed set $\mathsf{L}_{\mathsf{trans}}.\mathcal{T}_{\mathsf{wf}}$ comprises histories H such that for each thread t, $E[\mathsf{t}]$ restricted to $\mathsf{PTBegin}$, PTEnd and events of the form T-tagged events is of the form described by the regular expression $(\mathsf{PTBegin}.\lfloor \mathrm{T} \rfloor^*.\mathsf{PTEnd})^*$. In particular, transaction nesting is disallowed in our simple $\mathsf{L}_{\mathsf{trans}}$ library.

To de ne global consistency, we need to know when two operations are part of the same transaction. Given a history H, we de ne the *same-transaction* relation, strans, relating pairs of $e, e' \in \lfloor \mathrm{T} \rfloor \cup \mathsf{PTEnd} \cup \mathsf{PTBegin}$ executed by the same thread t such that there is no $\mathsf{PTBegin}$ or PTEnd executed by t between them. The set $\mathsf{L}_{\mathsf{trans}}.\mathcal{T}_{\mathsf{c}}$ of globally consistent histories contains histories H such that $\forall (e, e') \in \mathrm{strans}, e \in \lfloor \mathrm{P}^{\mathsf{tr}} \rfloor \Leftrightarrow e' \in \lfloor \mathrm{P}^{\mathsf{tr}} \rfloor$, and all completed PTEnd calls are in $\lfloor \mathrm{P}^{\mathsf{tr}} \rfloor$. Since the PTEnd call is related to all events inside its transaction, this speci cation does express that (1) a transaction persist by the time the call to PTEnd nishes and (2) all events persist *atomically*.

Finally, we need to de ne the local consistency predicate $\mathsf{L}_{\mathsf{trans}}.\mathcal{S}_{\mathsf{c}}$ describing the behavior of the registers provided by $\mathsf{L}_{\mathsf{trans}}$. This is where the we de ne the concrete meaning of having persisted for these registers. Let S be the sequential speci cation of a register. Let $H \in \mathbf{Hist}(\mathsf{L}_{\mathsf{trans}})$ be a history decomposed into k eras as $H_1 \cdot \lightning \cdot H_2 \cdot \lightning \cdot \dots \cdot \lightning \cdot H_k$. Then $H \in \mathsf{L}_{\mathsf{trans}}.\mathcal{S}_{\mathsf{c}}$ i all events are tagged with T, and there exists a $\prec$-linearization H_ℓ of $\big((H_1 \cdot \lightning \cdot H_2 \cdot \lightning \cdot \dots \cdot \lightning \cdot H_{k-1}) \cap \lfloor \mathrm{P}^{\mathsf{tr}} \rfloor\big) \cdot H_k$ such that $H_\ell \in S$, where $\lfloor \mathrm{P}^{\mathsf{tr}} \rfloor$ is the set of events of H tagged with P^{tr}. In other words, a write operation is seen after a crash i it has persisted. The requirement that such operations must appear within transactions and the guarantee that they persist at the same time in a transaction are covered by the global speci cations.

2.7 Library Implementations

We have described how to *specify* persistent libraries in our framework, and next describe how to *implement* persistent libraries. This is formalized by the judgment $\Lambda \vdash I : \mathsf{L}$, stating that I *is a correct implementation of library* L *and only uses calls in the collection of libraries* Λ. As usual in such layered frameworks [12, 24], the base layer, which represents the primitives of the hardware, is speci ed as a library, keeping the framework uniform. This judgement can be composed vertically as follows, where $I[I_L]$ denotes replacing the calls to library L in I with their implementations given by I_L (which in turn calls libraries Λ'):

$$\frac{\Lambda, \mathsf{L} \vdash I : \mathsf{L}' \quad \Lambda' \vdash I_L : \mathsf{L}}{\Lambda, \Lambda' \vdash I[I_L] : \mathsf{L}'}$$

As we describe later, this judgment denotes *contextual re nement* and is impractical to prove directly. We de ne a stronger notion that is *compositional* and more practical to use.

▶ **De nition 2.17** (Implementation). *Given a collection* Λ *of libraries and a library* L*, an* implementation I of L over Λ *is a map,* $I : \mathsf{L}.\mathcal{M} \times \mathbf{Val}_\perp \longrightarrow \mathcal{P}(\mathbf{Hist}(\Lambda))$*, such that it is downward-closed: 1) if* $H \in I(m(\vec{v})_{\mathsf{t}}, v_\perp)$ *and* H' *is a pre x of* H*, then* $H' \in I(m(\vec{v}), \perp)$*; and 2) each* $I(m(\vec{v})_{\mathsf{t}}{:}v_\perp)$ *history only contain events by thread* t*.*

Intuitively, $I(m(\vec{v}), v_\perp)$ contains the histories corresponding to a call $m(\vec{v})$ with outcome $v_\perp$, where $v_\perp = \perp$ denotes that the call has not terminated yet and $v_\perp = v \in \mathbf{Val}$ denotes the

```
globals log := Q.new()
method PTNewReg() := alloc(1)
method PTRead(l) := read(l)
method PTWrite(l, v) :=
  Q.append(log, (l, v));
  write(l, v)
method PTBegin() := FENCE();
method PTEnd() :=
  Append(log, COMMITTED);
  FENCE()
```

```
method PTRecover() :=
  let w = Q.new() in
  while (x := Q.pop(log))
    if (x = COMMITTED)
      w = Q.new();
    else
      Q.append(w, x);
  while ((l, v) = Q.pop(log)) {
    write(l, v); }
```

Figure 1 Implementation of $\mathsf{L}_{\mathsf{trans}}$

return value. Downward-closure means that an implementation contains all partial histories. We use a concrete programming language to write these implementations; its syntax and semantics are standard and given in the technical appendix.

For example, the implementation of $\mathsf{L}_{\mathsf{trans}}$ over $\mathsf{L}_{\mathsf{wreg}}$ and $\mathsf{L}_{\mathsf{Queue}}$ is given in Fig. 1. The idea is to keep an undo-log as a persistent queue that tracks the values of the variables *before* the transaction begins. At the end of a transaction, and after all its writes have persisted, we write the sentinel value COMMITTED to the log to indicate that the transaction was completed successfully. After a crash, the recovery routine PTRecover returns the undo-log and undoes the operations of *incomplete* transactions by writing their previous values.

Histories and Implementations. An implementation I of L over Λ is correct if for all histories $H \in \mathbf{Hist}(\{\mathsf{L}\} \cup \Lambda')$ that use library L as well as those in Λ', and all histories H' obtained by replacing calls to L methods with their implementation in I, if H' is consistent, then so is H (it satisfies the L specification).

We define the action $H \cdot I$ of an implementation I on an abstract history H in a relational way: $H' \in H \cdot I$ when we can *match* each operation $m'(\vec{v})$ in H' with some operation $f(m'(\vec{v}))$ in H in such a way that the collection $f^{-1}(m(\vec{v})_{\mathsf{t}}{:}v_\perp)$ of operations corresponding to some call $m(\vec{v})_{\mathsf{t}}{:}v_\perp$ in H agrees with $I(m(\vec{v})_{\mathsf{t}}{:}v_\perp)$.

▶ **Definition 2.18.** *Let I be an implementation of L over Λ; let $H \in \mathbf{Hist}(\{\mathsf{L}\} \cup \Lambda')$ and $H' \in \mathbf{Hist}(\Lambda \cup \Lambda')$ be two histories. Given a map $f : \{1, \ldots, |H'|\} \to \{1, \ldots, |H|\}$, H' (I, f)-*matches *H if the following hold:*

1. *f is surjective;*
2. *for all invocations of H, if $m(\vec{v})_{\mathsf{t}} \notin \mathsf{L}.\mathcal{M}$, then $f(m(\vec{v})_{\mathsf{t}}) = m(\vec{v})_{\mathsf{t}}$;*
3. *for all threads t, if e_1 precedes e_2 in $H'[\mathsf{t}]$, then $f(e_1)$ precedes $f(e_2)$ in $H[\mathsf{t}]$;*
4. *for all calls $m(\vec{v})_{\mathsf{t}}{:}v_\perp$ of H, the set $f^{-1}(m(\vec{v})_{\mathsf{t}})$ corresponds to a substring H'_m of $H'[\mathsf{t}]$ and $H'_m \in I(m(\vec{v})_{\mathsf{t}}{:}v_\perp)$, where $v_\perp$ is the (optional) return value of $m(\vec{v})_{\mathsf{t}}$ in H.*

The action *of I on a history H is defined as $H \cdot I := \{H' \mid \exists f.\ H'\ (I, f)\text{-matches } H\}$.*

Condition 1 ensures that all events of the abstract history are matched with an implementation event; condition 2 ensures that the events that do not belong to the library being implemented (L) are left untouched, and condition 3 ensures that the thread-local order of events in the implementation agrees with the one in the specification. The last condition (4) states that the events corresponding to the implementation of a call $m(\vec{v})$ are consecutive in the history of the executing thread t, and correspond to the implementation I.

Well-formedness and Consistency. Recall that libraries specify both how they should be used (*well-formedness*), and what they guarantee if used correctly (*consistency*). Using

these specifications (expressed as sets of histories) to define implementation correctness is more subtle than one might expect. Specifically, if we view a program using a library L as a downward-closed set of histories in $\mathbf{Hist}(\mathsf{L})$, we cannot assume all its histories are in the set $\mathsf{L}.\mathcal{S}_{\mathsf{wf}}$ of well-formed histories, as the semantics of the program will contain *unreachable* traces (see [24]). To formalize reachability at a semantic level, we define *hereditary consistency*, stating that each step in the history was consistent, and thus the current state is reachable.

▶ **Definition 2.19** (Consistency). *History* $H \in \mathbf{Hist}(\Lambda)$ *is* consistent *if for all* $\mathsf{L} \in \Lambda$, $H[\mathsf{L}] \in \mathsf{L}.\mathcal{S}_c$ *and* $\pi_{\mathsf{L}}(H) \in \mathsf{L}.\mathcal{T}_c$. *It is* hereditarily consistent *if all* $H[1..k]$ *are consistent, for* $k \leq |H|$.

This definition uses the anonymization operator π_{L} defined in 2.6 to test that the history H follows the global consistency predicates of every $\mathsf{L} \in \Lambda$.

We further require that programs using libraries respect *encapsulation*, defined below, stating that locations obtained from a library constructor are only used by that library instance. Specifically, the first condition ensures that distinct constructor calls return distinct locations. The second condition ensures that a non-constructor call e of L uses locations that have been allocated by an earlier call c ($c \prec e$) to an L constructor.

▶ **Definition 2.20** (Encapsulation). *A history* $H \in \mathbf{Hist}(\Lambda)$ *is* encapsulated *if the following hold, where* C *denotes the set of calls to constructors in* H*:*

1. *for all* $c, c' \in C$, *if* $c \neq c'$, *then* $\mathrm{loc}(c) \cap \mathrm{loc}(c') = \emptyset$;
2. *for all* $e \in H \setminus C$, *if* $\mathrm{loc}(e) \neq \emptyset$, *then there exist* $c \in C$, $\mathsf{L} \in \Lambda$ *such that* $e, c \in \mathsf{L}.\mathcal{M}$, $c \prec e$ *and* $\mathrm{loc}(e) \subseteq \mathrm{loc}(c)$.

We can now define when a history of Λ is *immediately well-formed*: it must be encapsulated and be well-formed according to each library in Λ and all the tags it uses.

▶ **Definition 2.21.** *History* $H \in \mathbf{Hist}(\Lambda)$ *is* immediately well-formed *if the following hold:*

1. H *is encapsulated;*
2. $H[\mathsf{L}] \in \mathsf{L}.\mathcal{S}_{wf}$, *for all* $\mathsf{L} \in \Lambda$; *and*
3. $\pi_{\mathsf{L}}(H) \in \mathsf{L}.\mathcal{T}_{wf}$ *for all* $\mathsf{L} \in \mathsf{TagDep}(\Lambda)$, *where the immediate dependencies* $\mathsf{TagDep}(\Lambda)$ *is defined as* $\bigcup_{\mathsf{L} \in \Lambda} \{\mathsf{L}\} \cup \Lambda_{\mathrm{tags}}(\mathsf{L})$.

We finally have the notions required to define a *correct implementation*.

Implementation Correctness. As usual, an implementation is correct if all behaviors of the implementation are allowed by the specification. In our setting, this means that if a concrete history is *hereditarily consistent*, so should the abstract history. Moreover, assuming the abstract history is well-formed, all corresponding concrete histories should also be well-formed; this corresponds to the requirement that the library implementation uses its dependencies correctly, under the assumption that the program itself uses its libraries correctly.

▶ **Definition 2.22** (Correct implementation). *An implementation* I *of* L *over* Λ *is* correct, *written* $\Lambda \vdash I : \mathsf{L}$, *if for all collections* Λ', *all abstract histories* $H \in \mathbf{Hist}(\{\mathsf{L}\} \cup \Lambda')$ *and all concrete histories* $H' \in H \cdot I \subseteq \mathbf{Hist}(\Lambda \cup \Lambda')$, *the following hold:*

1. *if* H *is immediately well-formed, then* H' *is also immediately well-formed; and*
2. *if* H' *is immediately well-formed and hereditarily consistent, then* H *is consistent.*

This definition is similar to *contextual refinement* in that it quantifies over all contexts: it considers histories that use arbitrary libraries as well as those that concern I directly. We now present a more convenient, *compositional* method for proving an implementation correct, which allows one to only consider libraries and tags that are used by the implemented library.

2.8 Compositionally Proving Implementation Correctness

Recall that in this section we present our framework in a simpli ed sequentially consistent setting; later in 3 we generalize our framework to the weak memory setting. We introduce the notion of *compositional correctness*, simplifying the global correctness conditions in Def. 2.22. Speci cally, while Def. 2.22 considers histories with arbitrary libraries that may use tags introduced by L, our compositional condition requires one to prove that only those L methods that are L-tagged satisfy $\mathsf{L}.\mathcal{T}_c$.

▶ **De nition 2.23** (Compositional correctness). *An implementation I of L over Λ is* compositionally correct *if the following hold:*

1. *For all Λ', $H \in \mathbf{Hist}(\{\mathsf{L}\} \cup \Lambda)$ and $H' \in H \cdot I \subseteq \mathbf{Hist}(\Lambda \cup \Lambda')$, if H' is well-formed, then H is well-formed;*
2. *For all $H \in \mathbf{Hist}(\mathsf{L})$ and $H' \in H \cdot I \subseteq \mathbf{Hist}(\Lambda)$, if H' is well-formed and hereditarily consistent, then $H \in \mathsf{L}.\mathcal{S}_c \cap \mathsf{L}.\mathcal{T}_c$; and*
3. *For all $\mathsf{L}' \in \Lambda$, $H \in \mathbf{Hist}(\{\mathsf{L}, \mathsf{L}', \star_{\mathsf{L}'}\})$ and $H' \in H \cdot I$, if $\pi_{\mathsf{L}'}(H') \in \mathsf{L}'.\mathcal{T}_{wf} \cap \mathsf{L}'.\mathcal{T}_c$, then $\pi_{\mathsf{L}'}(H) \in \mathsf{L}'.\mathcal{T}_c$.*

The preservation of well-formedness (condition 1) does not change compared to its counterpart in Def. 2.22, as in practice this condition is easy to prove directly. Condition 2 requires one to prove that the implementation is correct *in isolation* (without Λ'). Condition 3 requires one to prove that global consistency requirements are maintained for all dependencies of the implementation. In practice, this corresponds to proving that those L operations tagged with existing tags in Λ obey the global speci cations associated with these tags. Intuitively, the onus is on the library that *uses* a tag for its methods to prove the associated global consistency predicate: we need not consider unknown methods tagged with tags in $\mathsf{L}.\textsc{Tags}_{\text{new}}$.

Finally, we show that it is su cient to show an implementation I is compositionally correct as it implies that I is correct.

▶ **Theorem 2.24** (Correctness). *If an implementation I of L over Λ is* compositionally correct *(Def. 2.23), then it is also* correct *(Def. 2.22).*

▶ **Example 2.25** (Transactional Library $\mathsf{L}_{\mathsf{trans}}$). Consider the implementation I_{trans} of $\mathsf{L}_{\mathsf{trans}}$ over $\Lambda = \{\mathsf{L}_{\mathsf{wreg}}, \mathsf{L}_{\mathsf{Queue}}\}$ given in Fig. 1, and let us assume we were to show that I_{trans} is compositionally correct. Our aim here is only to outline the proof obligations that must be discharged; later in 5 we give a full proof in the more general weak memory setting.

1. For the rst condition of compositional correctness, we must show I_{trans} preserves well-formedness: if the abstract history H is well-formed, then so is any corresponding concrete history $H' \in H \cdot I_{\mathsf{trans}}$. This is straightforward as the well-formedness conditions of $\mathsf{L}_{\mathsf{wreg}}$ and $\mathsf{L}_{\mathsf{Queue}}$ are trivial, and $\mathsf{L}_{\mathsf{trans}}$ does not use any existing tag.
2. For the second condition of compositional correctness, we must show that I_{trans} preserves consistency in the other direction: keeping the notations as above, assuming H' is consistent for Λ, then H is consistent as speci ed by $\mathsf{L}_{\mathsf{trans}}$. There are two parts to this obligation, as we also have to show that the $\mathsf{L}_{\mathsf{trans}}$ s operations tagged with T satisfy the global consistency predicate of the library.
3. The last condition holds vacuously as $\mathsf{L}_{\mathsf{trans}}$ does not use any existing tags.

▶ **Example 2.26** (A Client of $\mathsf{L}_{\mathsf{trans}}$). To see how the global consistency speci cations work, consider a simple min-max counter library, $\mathsf{L}_{\mathsf{mmcnt}}$, tracking the maximal and minimal integer it has been given. The $\mathsf{L}_{\mathsf{mmcnt}}$ is to be used within $\mathsf{L}_{\mathsf{trans}}$ transactions, and provides four

```
method mmNew() :=
 (PTNewReg(), PTNewReg())

method mmAdd(x, n) :=
  PTWrite(min(n, PTRead(x.1)))
  PTWrite(max(n, PTRead(x.2)))

method mmMin(x) :=
  PTRead(x.1)

method mmMax(x) :=
  PTRead(x.2)
```

Figure 2 Implementation I_{mmcnt} of $\mathsf{L}_{\mathsf{mmcnt}}$

methods: `mmNew`() to construct a min-max counter, `mmAdd`(x, n), to add integer n to the min-max counter, and `mmMin`(x) and `mmMax`(x) to read the respective values.

We present the I_{mmcnt} implementation over $\mathsf{L}_{\mathsf{trans}}$ in Fig. 2. The idea is simply to track two integers denoting the minimal and maximal values of the numbers that have been added. Interestingly, even though they are stored in $\mathsf{L}_{\mathsf{trans}}$ registers, the implementation does not begin or end transactions: this is the responsibility of the client to avoid nesting transactions. This is enforced by $\mathsf{L}_{\mathsf{mmcnt}}$ using a global well-formedness predicate. Moreover, the `mmAdd` operation is tagged with $\textsc{t}$ from the $\mathsf{L}_{\mathsf{trans}}$ library, ensuring that it behaves well w.r.t. transactions. A non-example is a version of I_{mmcnt} where the minimum is in a $\mathsf{L}_{\mathsf{trans}}$ register, but the max is in a normal $\mathsf{L}_{\mathsf{wreg}}$ register. This breaks the atomicity guarantee of transactions.

Formally, the interface L_{mmcnt} has four methods as above, where `mmNew` is the only constructor. The set of used tags is $\textsc{Tags}_{\mathrm{dep}} = \{\textsc{t}, \textsc{p}^{\mathsf{tr}}\}$, and all L_{mmcnt} methods are tagged with $\textsc{t}$ as they all use primitives from $\mathsf{L}_{\mathsf{trans}}$. The consistency predicate is de ned using the obvious sequential speci cation S_{mmcnt}, which states that calls to `mmMin` return the minimum of all integers previously given to `mmAdd` in the sequential history. We lift this to (concurrent) histories as follows. A history $H \in \mathbf{Hist}(\mathsf{L}_{\mathsf{mmcnt}})$ is in $\mathsf{L}_{\mathsf{mmcnt}}.\mathcal{S}_{\mathsf{c}}$ if there exists $E_\ell \in S_{\mathsf{mmcnt}}$ that is a $\prec$-linearization of $E_1[\textsc{p}^{\mathsf{tr}}] \cdot E_2[\textsc{p}^{\mathsf{tr}}] \cdots E_{n-1} \cdot E_n[\textsc{p}^{\mathsf{tr}}]$, where H constructs n eras decomposed as $H = E_1 \cdot \lightning \cdots \lightning \cdot E_n$ (recall that $E[\textsc{p}^{\mathsf{tr}}]$ denotes the sub-history with events tagged with $\textsc{p}^{\mathsf{tr}}$, that is, persisted events.). The global speci cation and well-formedness conditions of $\mathsf{L}_{\mathsf{mmcnt}}$ are trivial. Because $\mathsf{L}_{\mathsf{mmcnt}}$ uses tag $\textsc{t}$ of $\mathsf{L}_{\mathsf{trans}}$, a well-formed history of $\mathsf{L}_{\mathsf{mmcnt}}$ must satisfy $\mathsf{L}_{\mathsf{trans}}.\mathcal{T}_{\mathsf{wf}}$, which requires that all operations tagged with $\textsc{t}$ be inside transactions, and $\mathsf{L}_{\mathsf{trans}}.\mathcal{T}_{\mathsf{c}}$ guarantees that $\mathsf{L}_{\mathsf{mmcnt}}$ operations persist atomically in a transaction.

When proving that the implementation in Fig. 2 satis es $\mathsf{L}_{\mathsf{mmcnt}}$ using compositional correctness, one proof obligation is to show that, given histories $H \in \mathbf{Hist}(\{\mathsf{L}_{\mathsf{trans}}, \mathsf{L}_{\mathsf{mmcnt}}, \star_{\mathsf{L}_{\mathsf{trans}}}\})$ and $H' \in H \cdot I_{\mathsf{mmcnt}} \subseteq \mathbf{Hist}(\{\mathsf{L}_{\mathsf{trans}}, \star_{\mathsf{L}_{\mathsf{trans}}}\})$, if $\pi_{\mathsf{L}_{\mathsf{trans}}}(H') \in \mathsf{L}_{\mathsf{trans}}.\mathcal{T}_{\mathsf{c}}$, then $\pi_{\mathsf{L}_{\mathsf{trans}}}(H) \in \mathsf{L}_{\mathsf{trans}}.\mathcal{T}_{\mathsf{c}}$. This corresponds precisely to the fact that min-max counter operations persist atomically in a transaction, assuming the primitives it uses do as well.

2.9 Generic Durable Persistency Theorems

We consider another family of libraries with persistent reads/writes guaranteeing the following:

> if one replaces regular (volatile) reads/writes in a *linearizable* implementation with persistent ones, then the implementation obtained is *durably linearizable*.

We consider two such such libraries: FliT [31] and Mirror [9]. Thanks to our framework, we formalise the statement above for the rst time and prove it for both Flit and Mirror against a realistic consistency (concurrency) model (see 4).

3 A General Framework for Persistency and Consistency

We generalise our persistency speci cation framework from 2 to account for an arbitrary (potentially weak) memory consistency (concurrency) model (rather than sequential consistency (SC) which was hard-coded into our formalism in 2). As such, we need to revisit and generalise several of our de nitions.

3.1 Plain Executions and Executions

Plain Executions. Unlike in the SC setting of 2 where we represented an execution as a totally-ordered history (sequence) of events, in the general weak consistency setting, such a total execution order does not exist in general. As such, we model an execution as a pomset (partially ordered multiset) of events.

▶ **De nition 3.1.** *A* pomset *over the set X is a tuple $((E, \leq), \lambda)$ consisting of an ordered set E and a map $\lambda : E \to X$. We write $\mathcal{O}(X)$ for the set of non-empty pomsets over X. Two pomsets $((E, \leq), \lambda)$ and $((F, \leq), \mu)$ over X are identi ed if there exists an order-isomorphism $f : E \to F$ such that $\mu \circ f = \lambda$. The underlying set E of a pomset $P = ((E, \leq), \lambda)$ is denoted by $|P|$.*

Following the literature on weak consistency models where the execution of each instruction is modelled by a *single* event, we handle method calls di erently from 2: rather than having two distinct events for each method invocation and return, we model each method call as a single event $m(\vec{v})\!:\!v_\perp$, which is *incomplete* if $v_\perp = \perp$ and *complete* otherwise. As such, an *event* is either such a method call, or it is a crash event. Given a library interface L (as given in Def. 2.13), we can then model a *plain execution* of L as a pomset $((E, \mathrm{po}), \mathrm{lab})$, where E is a set of L events, po is the *program order*, and lab is the *label function*, associating each event with its label of the form $m(\vec{v})^T\!:\!v_\perp$ or $\lightning$. Moreover, incomplete method calls are maximal events in po unless their immediate successor is a crash event.

▶ **De nition 3.2** (Plain executions)**.** *A* plain execution *G of an interface L is a pomset $((E, \mathrm{po}), \lambda)$, where E is a set of events,* po *is the program order and $\lambda : E \to L.\mathcal{M} \cup \{\lightning\}$, such that the* po*-immediate successor of an incomplete method call $m(\vec{v})\!:\!\perp$ is a crash event.*

Given two plain executions, $G_1 = ((E_1, \mathrm{po}_1), \lambda_1)$ and $G_2 = ((E_2, \mathrm{po}_2), \lambda_2)$, their sequential composition *is $G_1; G_2 \triangleq ((\tilde{E}, \tilde{\mathrm{po}}), \tilde{\lambda})$, where $\tilde{E} \triangleq E_1 \amalg E_2$ is the disjoint sum[1] of events, $\tilde{\lambda}$ is the induced labelling, and $\tilde{\mathrm{po}}$ is the transitive closure of the following relation:*

$$\mathrm{po}_1 \cup \mathrm{po}_2 \cup \left\{(e_1, e_2) \in E_1 \times E_2 \,\middle|\, e_1 \text{ is labeled by a complete operation or } e_2 = \lightning\right\}$$

The set of plain executions is $\mathbf{PExec}(L)$*; for brevity, $((E, \mathrm{po}), \mathrm{lab})$ is often written as $\langle E, \mathrm{po}\rangle$.*

Note that the sequential composition of two plain executions preserves the invariant that the only possible immediate po-successors of an incomplete method call is a crash event.

Executions. Recall from 2 that a method call C_1 happens before another C_2 in a history H ($C_1 \prec_H C_2$) if the response of e_1 precedes the invocation of C_2 in the totally-ordered history H. This captures the real-time ordering present under the strong SC model. However, under weaker consistency models (e.g. in modern multi-core processors) this notion of real-time ordering is not realistic. Indeed, the happens-before notion varies from one weak

[1] For example, de ne $X \amalg Y \coloneqq \{0\} \times X \cup \{1\} \times Y$.

model to another, and is typically de ned in terms of a *synchronizes-with* relation, which itself is also model-dependent. As such, we record the happens-before and synchronizes-with relations as primitive notions within the de nition of a library execution.

Note that executions additionally track the *tags* associated with method calls: events are labelled with *tagged* method calls $m(\vec{v})^T : v_\perp$ as well as crash events. Indeed, in general the tags are not observable by programs and belong to executions: e.g. the tag P^{Px86} (denoting that a write has persisted) pertains to phenomena that are external to the program.

▶ **De nition 3.3** (Library executions). *Given a library interface L, a* library L execution *is a tuple $\mathcal{G} = \langle E, \text{po}, \text{sw}, \text{hb}\rangle$ such that:*

- $\langle E, \text{po}\rangle$ *is a pomset labeled with tagged events of L;*
- $\text{sw} \subseteq E \times E$ *is the* synchronizes-with *relation;*
- hb *is the* happens-before *relation, which is a strict order with* $\text{po} \cup \text{sw} \subseteq \text{hb}$.

The set of library L executions is denoted $\mathbf{Exec}(L)$. *Given a library execution $\mathcal{G} = \langle E, \text{po}, \text{sw}, \text{hb}\rangle$, its underlying plain execution $\langle E, \text{po}\rangle$ is denoted by $|\mathcal{G}|$. An execution $\mathcal{G}$* re nes *a plain execution G, written $\mathcal{G} \sqsubset G$, when $|\mathcal{G}| = G$. This de nition is lifted to a collections Λ of libraries by allowing events to be labelled with any library in Λ.*

We often use the $\mathcal{G}$. pre x to project the components of $\mathcal{G}$, e.g. $\mathcal{G}.\text{sw}$.

3.2 Library Speci cations

Most of our de nitions pertaining library speci cations remain unchanged from 2, and the only de nition we need to adapt is the anonymization operation $\pi_L : \mathbf{Exec}(\{\mathsf{L}\} \cup \Lambda) \to \mathbf{Exec}(\{\mathsf{L}\} \cup \star_L)$ which now operates on decorated pomsets. To do this, we rst change the execution labelling: $\tilde{\mathcal{G}}$ is the same as $\mathcal{G}$, except that the labelling map, lab, of the underlying plain execution is replaced with $f \circ \text{lab}$, where the map f over events is de ned as:

$$f(\ell) = \begin{cases} \star^T & \text{if } \ell = m(\vec{v})^T : v_\perp \notin \mathsf{L}.\mathcal{M} \\ \ell & \text{if } \ell \in \mathsf{L}.\mathcal{M} \text{ or } \ell = \lightning \end{cases}$$

We then de ne $\pi_L(\mathcal{G})$ as the restriction of $\tilde{\mathcal{G}}$ to events which are not tagged with $\emptyset$.

As in Def. 2.16, a *library speci cation* is a tuple $\langle L, \Lambda_{\text{tags}}, \mathcal{S}_{\mathsf{c}}, \mathcal{S}_{\mathsf{wf}}, \mathcal{T}_{\mathsf{c}}, \mathcal{T}_{\mathsf{wf}}\rangle$, where L and Λ_{tags}, are as before, and $\mathcal{S}_{\mathsf{c}}$, $\mathcal{S}_{\mathsf{wf}}$, $\mathcal{T}_{\mathsf{c}}$ and $\mathcal{T}_{\mathsf{wf}}$ are sets of *executions* (not histories) as in Def. 3.3.

▶ **Example 3.4** (Px86). Our main example of a library with weak consistency and persistency is Px86 (the Intel-x86 consistency and persistency model [28]). Our speci cation below is a simple adaptation of [28]. Let us begin with the library interface L_{Px86}, introducing two new tags: D, denoting events that are *durable* in that they *can* persist; and P^{Px86}, denoting events that did persist. The *Px86 interface* is $\langle \mathcal{M}, \mathcal{M}_{\text{c}}, \text{loc}, \{\text{D}, \text{P}^{\mathsf{Px86}}\}, \emptyset\rangle$, where:

- $\mathcal{M}_{\text{c}} \triangleq \{\mathsf{alloc}()\}$;
- $\mathcal{M}_{\text{d}} \triangleq \mathcal{M}_{\text{c}} \cup \bigcup_{x \in \mathbf{Loc}} \mathcal{M}_{\text{d}}^x$, where $\mathcal{M}_{\text{d}}^x \triangleq \mathcal{W}^x \cup \mathcal{U}^x \cup \mathcal{FL}^x \cup \mathcal{FO}^x$; and:
  $\mathcal{W}^x \triangleq \{\mathsf{write}(x, v) \mid v \in \mathbf{Val}\}$ is the set of write events on location x;
  $\mathcal{U}^x \triangleq \{\mathsf{Upd}(x, v, v') \mid v, v' \in \mathbf{Val}\}$ is the set of read-modify-write operations on location x;
  $\mathcal{FL}^x \triangleq \{\mathsf{flush}(x)\}$ is the set of synchronous ush events on location x; and
  $\mathcal{FO}^x \triangleq \{\mathsf{flush}_{\mathsf{opt}}(x)\}$ is the set of delayed ush (ush-opt) events on location x.
- $\mathcal{M} \triangleq \mathcal{M}_{\text{c}} \cup \mathcal{M}_{\text{d}} \cup \mathcal{MF} \cup \mathcal{SF} \cup \bigcup_{x \in \mathbf{Loc}} \mathcal{R}^x$, where $\mathcal{MF} \triangleq \{\mathsf{mfence}\}$ is a memory fence invocation, $\mathcal{SF} \triangleq \{\mathsf{sfence}\}$ is a store fence, and $\mathcal{R}^x \triangleq \{\mathsf{read}(x)\}$ is a read from location x.

- $\forall x \in \mathbf{Loc}.\ \forall i \in \mathcal{M}^x_{\mathrm{d}} \cup \mathcal{R}^x.\ \mathrm{loc}(i, v_\perp) \triangleq \{x\}$, $\forall x.\ \mathrm{loc}(\mathsf{alloc}(), x) \triangleq \{x\}$ and otherwise $\mathrm{loc}(l, v_\perp) \triangleq \emptyset$.

Given an execution $\mathcal{G} \in \mathbf{Exec}(L_{\mathsf{Px86}})$ with $\mathcal{G} = \langle E, \mathrm{po}, \mathrm{sw}, \mathrm{hb}\rangle$, let $R \triangleq \bigcup_{x \in \mathbf{Loc}} R^x$ with $R^x \triangleq \left\{e \in E \,\middle|\, \mathrm{lab}(e) \in \mathcal{R}^x\right\}$, and let W^x, W, U^x, U, FL^x, FL, FO^x, FO, MF and SF be defined analogously. Let us also define the following sets:

$$
\begin{array}{ll}
D \triangleq \left\{e \in E \,\middle|\, \mathrm{lab}(e) \in \mathcal{M}_{\mathrm{d}}\right\} & \text{durable events} \\
D^x = \left\{e \in D \,\middle|\, \mathrm{loc}(e) = \{x\}\right\} & \text{durable events on location } x \\
\mathrm{hb}_{\mathsf{e}} = \mathrm{hb} \cap \left\{(a,b) \,\middle|\, (a,b) \notin \mathrm{po} \cup \mathrm{po}^{-1}\right\} & \mathrm{hb} \text{ between different threads} \\
\mathrm{eb} = \mathrm{po}; \lightning; \mathrm{po} & \text{the era-before relation} \\
\mathrm{se} = \{(e, e') \mid (e, e') \notin \mathrm{eb} \cup \mathrm{eb}^{-1}\} & \text{the same-era relation}
\end{array}
$$

Given a relation r on E, let $r_{\mathsf{se}} \triangleq r \cap \mathrm{se}$ Execution $\mathcal{G}$ is *Px86-consistent* if there exists a *reads-from* relation $\mathrm{rf} \subseteq W \times R$ such that rf^{-1} is total and functional (i.e. every read is related to exactly one write), a strict order tso that is total on $W \cup U$ and a strict order nvo on D such that $\mathrm{rf}, \mathrm{tso}, \mathrm{nvo} \subseteq \mathrm{eb} \cup \mathrm{se}$ and:

$$
\begin{array}{ll}
\mathrm{hb} \cup \mathrm{tso} \text{ is acyclic and } \mathrm{rf} \subseteq \mathrm{tso}_{\mathsf{se}} \cup \mathrm{po} & \text{(A1)} \\
\forall x \in \mathbf{Loc}.\ \forall (w,r) \in \mathrm{rf}_x.\ \forall w' \in W^x \cup U^x.\ (w', r) \in \mathrm{tso}_{\mathsf{se}} \cup \mathrm{po} \Rightarrow (w, w') \notin \mathrm{tso}_{\mathsf{se}} & \text{(A2)} \\
([E]; \mathrm{po}_{\mathsf{se}}; [MF \cup U]) \cup ([MF \cup U \cup R]; \mathrm{po}_{\mathsf{se}}; [E]) \subseteq \mathrm{tso}_{\mathsf{se}} & \text{(A3)} \\
([E]; \mathrm{po}_{\mathsf{se}}; [SF]) \cup ([SF]; \mathrm{po}_{\mathsf{se}}; [E \setminus R]) \subseteq \mathrm{tso}_{\mathsf{se}} & \text{(A4)} \\
[W \cup FL]; \mathrm{po}_{\mathsf{se}}; [W \cup FL] \subseteq \mathrm{tso}_{\mathsf{se}} & \text{(A5)} \\
\forall x \in \mathbf{Loc}.\ ([FL^x]; \mathrm{po}_{\mathsf{se}}; [FO^x]) \cup ([FO^x]; \mathrm{po}_{\mathsf{se}}; [FL^x]) \cup ([W^x]; \mathrm{po}_{\mathsf{se}}; [FO^x]) \subseteq \mathrm{tso}_{\mathsf{se}} & \text{(A6)} \\
\forall x \in \mathbf{Loc}.\ \mathrm{tso}_{\mathsf{se}}\lfloor_{D^x} \subseteq \mathrm{nvo}_{\mathsf{se}} & \text{(A7)} \\
\forall x \in \mathbf{Loc}.\ [D^x]; (\mathrm{tso} \cup \mathrm{hb}_{\mathsf{e}})_{\mathsf{se}}; [FO^x \cup FL^x] \subseteq \mathrm{nvo} & \text{(A8)} \\
([FL]; \mathrm{tso}_{\mathsf{se}}; [D]) \cup ([FO]; \mathrm{po}_{\mathsf{se}}; [MF \cup SF \cup U]; \mathrm{tso}_{\mathsf{se}}; [D]) \subseteq \mathrm{nvo}_{\mathsf{se}} & \text{(A9)} \\
\forall (w,r) \in \mathrm{rf} \cap \mathrm{eb}.\ w \in \lfloor \mathrm{P}^{\mathsf{Px86}} \rfloor \ \wedge\ ([\{w\}]; \mathrm{nvo}; [\lfloor \mathrm{P}^{\mathsf{Px86}} \rfloor \cap W^{\mathrm{loc}(w)}]; \mathrm{eb}; [\{r\}]) = \emptyset & \text{(New)}
\end{array}
$$

The specification of Px86 above is adapted from [28, Def. 4]. Specifically, axioms (A1) (A9) are directly imported from [28]. The main difference is that instead of considering *execution chains*, which are sequences of executions, we use our more general approach of a single execution with crash events. This generality is required for general data structures such as queues, as the pre-crash events cannot be summarized by a set of initial events, unlike in Px86 where the maximal write to each location captures the behavior of that location after a crash. This is reflected in (New), stating that a read r that reads from a write w in an earlier era, must do so from the maximal persisted such write on the same location, i.e. there should be no intervening persisted writes on the same location between w and r. Moreover, $\mathrm{rf}, \mathrm{tso}, \mathrm{nvo} \subseteq \mathrm{eb} \cup \mathrm{se}$ ensures that events in later eras are not rf-, tso- or nvo-related to those in earlier eras.

3.3 Library Implementations

We next describe how our general framework can be used to verify library implementations.

3.3.1 Semantic implementations

Analogously to 2, a semantic implementation in this general setting is a downward-closed map from method calls to sets of plain executions. This time, we define downward-closure with respect to the *prefix order* over plain executions.

Basic domains

$\mathsf{a} \in \mathbf{Reg}$	Registers
$v \in \mathbf{Val}$	Values
$\mathsf{t} \in \mathbf{Tid}$	Thread IDs
$m \in \mathbf{Meth}$	Method names

Expressions, sequential commands and programs

$$\begin{array}{rl} \mathbf{Exp} \ni \mathsf{e} ::= & v \mid \mathsf{a} \mid \mathsf{e} + \mathsf{e} \mid \cdots \\ \mathbf{Com} \ni \mathsf{C} ::= & \mathbf{skip} \mid \mathsf{a} := \mathsf{e} \mid \mathsf{a} := m(\mathsf{b}_1 \cdots \mathsf{b}_n) \mid \mathbf{continue} \\ & \mid \mathsf{C}; \mathsf{C} \mid \mathbf{if}\ (\mathsf{e})\ \mathbf{then}\ \mathsf{C}\ \mathbf{else}\ \mathsf{C} \mid \mathbf{while}\ (\mathsf{e})\ \mathbf{do}\ \mathsf{C} \\ & \mathbf{return}\ \ e \end{array}$$

$$\mathsf{P} \in \mathbf{Prog} \triangleq \mathbf{Tid} \xrightarrow{\text{fin}} \mathbf{Com}$$

Figure 3 A simple concurrent programming language

▶ **Definition 3.5.** *Given a collection Λ and plain executions $G, G' \in \mathbf{PExec}(\Lambda)$, the plain execution G is an* immediate prefix *of $G' \in \mathbf{PExec}(\Lambda)$, written $G \hookrightarrow_{im} G'$, if there exists a* po*-maximal event $e \in G'.E$ such that $G' \setminus \{e\} = G$. The* prefix order*, $\hookrightarrow$, is the transitive closure of $\hookrightarrow_{im}$. Both definitions are lifted naturally to library executions: $\mathcal{G} \hookrightarrow_{im} \mathcal{G}'$ if $|\mathcal{G}| \hookrightarrow_{im} |\mathcal{G}'|$, and $\hookrightarrow$ on library executions is the transitive closure of $\hookrightarrow_{im}$ on library executions. Both prefix relations are defined on executions by considering* hb*-maximal events.*

Intuitively, $G \hookrightarrow_{im} G'$ holds when G' can be obtained by adding a new event e to G, corresponding to a single step of program execution by one thread. When interpreting programs, we consider all their partial executions, and as such, any immediate prefix of an execution of a program will also be in its semantics. We can now define *library implementations* as indexed sets of executions that satisfy this downward-closure property.

▶ **Definition 3.6.** *An* implementation *I of library L over collection Λ is a map, $I : \mathsf{L}.\mathcal{M} \times \mathbf{Val}_\perp \longrightarrow \mathbf{PExec}(\Lambda)$ that is* downward-closed*: for all non-empty executions G and G', if $G \hookrightarrow G'$ and $G' \in I(m(\vec{v}){:}v_\perp)$, then $G \in I(m(\vec{v}){:}\perp)$, where we identify a method call with the pair of the invocation and the return value.*

3.3.2 A simple Concurrent Programming Language

In Fig. 3 we define a simple concurrent language for library implementations in our case studies (§ 5, § 4). The semantics of a program P, written $[\![\mathsf{P}]\!]$, is defined as a set of plain executions and is standard (see [24]). We write $[\![\mathsf{P}]\!]^v$ for the set of plain executions where P returns value v, and write $[\![\mathsf{P}]\!]^\perp$ for the set of plain executions where P has not yet returned.

There are two contexts in which the programming language is used: to define library implementations and to define a top-level program. A *syntactic* library implementation $\mathtt{I}$ of L over Λ is a program $\mathtt{I}(m(\vec{x}))$ with variables $\vec{x}$ for each method m of L. This defines a *semantic* implementation $[\![\mathtt{I}]\!]$ with $[\![\mathtt{I}]\!](m(\vec{v}){:}v_\perp) \triangleq [\![\mathtt{I}(m(\vec{x}))[\vec{x} := \vec{v}]]\!]^{v_\perp}$, where $[\vec{x} := \vec{v}]$ denotes the point-wise substitution of $\vec{x}$ with $\vec{v}$.

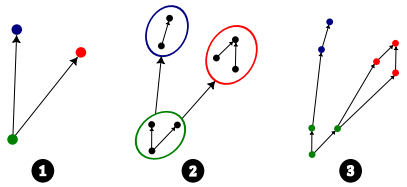
The semantics $[\![\mathsf{P}]\!]_{\mathsf{prog}}$ of a top-level program is different as we must account for crashes. This is done by restarting the program from scratch after each crash:

$$[\![\mathsf{P}]\!]^{v_\perp}_{\mathsf{prog}} \ = \ \left\{ G_1 \cdot \lightning \cdots \lightning \cdot G_n \,\middle|\, \forall i < n.\ G_i \in [\![\mathsf{P}]\!]^\perp \text{ and } G_n \in [\![\mathsf{P}]\!]^{v_\perp} \right\}$$

Formally, an execution of P is any number of partial executions interrupted by a crash, followed by a possibly complete execution.

3.3.3 Semantic Substitutions

We define the action of an implementation I of a library L over Λ on a plain execution $G \in \mathbf{PExec}(\mathsf{L})$, which captures what happens when a program that calls L is linked with the L implementation I. This operation, defined at the level of pomsets, is crucial to define what it means for the implementation to be correct.



- Picture 1 shows a 3-element pomset $P{=}((E,\leq),\lambda)$.
- In picture 2, each node $e \in E$ is now labelled with a pomset in the set $g(\lambda(e))$.
- Picture 3 shows the final result, where the inner pomsets have been inlined in P.

Figure 4 Substitution of labelled pomset P with g

As discussed in Def. 3.2, a plain execution is simply a pomset (Def. 3.1) labelled with method calls and crashes. Pomsets are endowed with a natural *substitution operation*[2]. Consider a pomset $P = ((E,\leq),\lambda)$ over a set X and a map $g : X \to \mathcal{O}(Y)$ associating labels in X with pomsets labelled over a set Y. Their substitution $P \cdot g \in \mathcal{O}(Y)$, depicted in Fig. 4, is obtained by replacing each event e in P with the pomset $g(\lambda(e))$. Formally, the carrier set of the pomset $P \cdot g$ is the disjoint union $F \triangleq \coprod_{e\in E} |g(\lambda(e))|$: each node e of P is replaced by the set of nodes of the pomset $g(\lambda(e))$. Elements of this set are of the form (e,f) with $f \in |g(\lambda(e))|$, and are ordered lexicographically as follows:

$$(e_1,f_1) \leq_{P\cdot g} (e_2,f_2) \iff (e_1 \leq_P e_2) \vee (e_1 = e_2 \wedge f_1 \leq_{g(\lambda(e_1))} f_2)$$

Finally, the label of an element (e,f) is $\mu_{\lambda(e)}(f) \in Y$, where we write $\mu_x : |g(x)| \to Y$ for the labeling map of the pomset $g(x)$.

Because a program is interpreted using a *set* of such pomsets, we also use the powerset $\mathcal{P}$ which maps a set X to the set $\{A \mid A \subseteq X\}$ of its subsets: $[\![e]\!]^{v_\perp} \in \mathcal{P}\mathcal{O}(\mathsf{calls}(\mathsf{L}))$, where $\mathsf{calls}(\mathsf{L})$ is the set of *method calls* (of the form $m(\vec{v}){:}v_\perp$) of library L. We extend the substitution operation to sets of pomsets: given a set $\mathbb{P} \in \mathcal{P}\mathcal{O}(X)$ of pomsets, and a map $g : X \to \mathcal{P}\mathcal{O}(Y)$, their substitution is as follows, where the substitution operation used on the right-hand side is the previous one, operating on pomsets, $\mathrm{Id}_S : S \to S$ is the identity map on the set S, and the map λ' has type $E \to \mathcal{O}(Y)$.

$$\mathbb{P} \cdot g \quad := \quad \bigcup_{((E,\leq),\lambda)\in\mathbb{P}} \left\{((E,\leq),\lambda') \cdot \mathrm{Id}_{\mathcal{O}(Y)} \mid \forall e \in E,\ \lambda'(e) \in g(\lambda(e))\right\}$$

Informally, a pomset P belongs to $\mathbb{P}\cdot g$ if it can be obtained from a pomset $Q = ((E,\leq),\lambda) \in \mathbb{P}$ by replacing each event in Q by some pomset in $g(\lambda(e)) \in \mathcal{P}\mathcal{O}(Y)$ — see Fig. 4.

Applying this general operation to plain executions and (semantic) implementations yields the desired operation because it preserves maximality of incomplete operations.

▶ **Proposition 3.7.** *The operation above lifts to an operation on plain executions:*

$$- \cdot I \ : \ \mathbf{PExec}(\mathsf{L}) \longrightarrow \mathcal{P}(\mathbf{PExec}(\Lambda))$$

Another way to formalize how to link an implementation with a program is syntactically: given a program P and a syntactic implementation $\mathtt{I}$, we can define $\mathsf{P} \cdot \mathtt{I}$ as the program where all calls to L methods are replaced with their source code given by $\mathtt{I}$. As expected, the two ways of linking with an implementation are compatible, in the following sense.

▶ **Proposition 3.8.** *Given a program* P *that uses* L *and a syntactic implementation* $\mathtt{I}$ *of* L*:*

$$[\![\mathsf{P}]\!]_{\mathsf{prog}} \cdot [\![\mathtt{I}]\!] = [\![\mathsf{P} \cdot \mathtt{I}]\!]_{\mathsf{prog}}$$

[2] This is indeed the "bind" operation of a monad structure on $\mathcal{O}$.

3.3.4 Implementation Correctness

We now have the notions required to lift the *correctness of an implementation* to the general weak memory setting. The main di erence with 2 is that we must consider the two levels of *plain executions*, which contain the events encountered by the program, and *executions*, which contain additional information that determine whether the execution is possible (consistent).

As an execution may contain calls from several libraries, given $\mathcal{G} \in \mathbf{Exec}(\Lambda)$, $\mathsf{L} \in \Lambda$ and a relation r on $\mathcal{G}.E$, we write $\mathcal{G}.E_{\mathsf{L}}$ for the $\mathcal{G}.E$ events labelled with L method calls, and we write r_{L} for $r \cap (\mathcal{G}.E_{\mathsf{L}} \times \mathcal{G}.E_{\mathsf{L}})$. Similarly, we write $\mathcal{G} \downharpoonright \mathsf{L}$ for $(\mathcal{G}.E_{\mathsf{L}}, \mathcal{G}.\mathrm{po}_{\mathsf{L}}, \mathcal{G}.\mathrm{sw}_{\mathsf{L}}, \mathcal{G}.\mathrm{hb}_{\mathsf{L}})$.

▶ **De nition 3.9.** *An execution $\mathcal{G} \in \mathbf{Exec}(\Lambda)$ is* consistent *if:*

1. $\mathcal{G}.\mathrm{sw} = \bigcup_{\mathsf{L} \in \Lambda} \mathcal{G}.\mathrm{sw}_{\mathsf{L}}$*;*
2. *for all $\mathsf{L} \in \Lambda$, $\mathcal{G} \downharpoonright \mathsf{L} \in \mathsf{L}.\mathcal{S}_c$; and*
3. *for all $\mathsf{L} \in \Lambda$, $\pi_{\mathsf{L}}(\mathcal{G}) \in \mathsf{L}.\mathcal{T}_c$.*

Execution $\mathcal{G}$ is hereditarily consistent *if either $\mathcal{G}$ is empty, or $\mathcal{G}$ is consistent and there exists $\mathcal{G}' \hookrightarrow_{im} \mathcal{G}$ such that $\mathcal{G}'$ is hereditarily consistent.*

In other words, an execution $\mathcal{G}$ is hereditarily consistent if there exists a sequence of consistent executions from the empty execution to $\mathcal{G}$, where each step corresponds to adding one event.

Encapsulation is de ned as in Def. 2.20, where the derived happens-before order $\prec$ is replaced with the primitive hb component of the execution.

▶ **De nition 3.10.** *An execution $\mathcal{G} \in \mathbf{Exec}(\Lambda)$ is* immediately well-formed *if:*

1. *$\mathcal{G}$ is encapsulated;*
2. *$\mathcal{G} \downharpoonright \mathsf{L} \in \mathsf{L}.\mathcal{S}_{wf}$ for all $\mathsf{L} \in \Lambda$;*
3. *$\pi_{\mathsf{L}}(\mathcal{G}) \in \mathsf{L}.\mathcal{T}_{wf}$ for all $\mathsf{L} \in \Lambda \cup \mathsf{L}.\Lambda_{\mathrm{tags}}$.*

An execution $\mathcal{G}$ is well-formed *if, for all $\mathcal{G}'' \hookrightarrow_{im} \mathcal{G}' \hookrightarrow \mathcal{G}$, if $\mathcal{G}''$ is consistent then $\mathcal{G}'$ is immediately well-formed.*

Intuitively, an implementation I of L over Λ is *correct* if 1) it transports well-formedness from the high-level execution to the low-level one as we expect library L to be used correctly; and 2) it transports consistency from the low-level execution to the high-level one, as we assume the underlying Λ libraries are correctly implemented.

▶ **De nition 3.11** (Correct implementation)**.** *An implementation I of Lover Λ is* correct*, written $\Lambda \vdash I : \mathsf{L}$ if, for all collections Λ' of libraries, all abstract plain executions $G \in \mathbf{PExec}(\{\mathsf{L}\} \cup \Lambda')$ and all concrete plain executions $G' \in G \cdot I \subseteq \mathbf{PExec}(\Lambda \cup \Lambda')$:*

1. *for all well-formed $\mathcal{G}$, if $\mathcal{G} \sqsubset G$, then there exists a well-formed $\mathcal{G}'$ such that $\mathcal{G}' \sqsubset G'$;*
2. *for all well-formed and hereditarily consistent $\mathcal{G}'$, if $\mathcal{G}' \sqsubset G'$, then there exists a hereditarily consistent $\mathcal{G}$ such that $\mathcal{G} \sqsubset G$.*

It may not be obvious that the above de nition is the right one. In 3.3.5 below we de ne the observable semantics of a closed program and prove that this de nition is indeed adequate to show that an implementation is correct.

3.3.5 Semantics of Programs

A program is *well-formed* if all its plain executions can be justified by well-formed executions:

$$\forall v_\bot \in \mathbf{Val}_\bot.\ \forall G \in [\![\mathsf{P}]\!]^{v_\bot}_{\mathsf{prog}}.\ \exists \mathcal{G} \sqsubset G.\quad \mathcal{G} \text{ is well-formed}$$

A program that is *not* well-formed is considered not to have a well-defined behavior, and thus we only consider the semantics of well-formed programs. Given a well-formed program P that uses libraries Λ, its behavior is defined as the values justified by a consistent execution:

$$\text{behaviors}(\mathsf{P}) \triangleq \{v \mid \exists G \in [\![\mathsf{P}]\!]^{v}_{\mathsf{prog}}.\ \exists \mathcal{G} \sqsubset G.\ \mathcal{G} \text{ is hereditarily consistent}\}$$

One justification for the definition of correctness for a semantic implementation is that it recovers the usual definition of correctness of a syntactic implementation [24].

▶ **Theorem 3.12.** *Let* P *be a well-formed program that uses* Λ*, and let* $\mathtt{I}$ *be a syntactic implementation of* $\mathsf{L} \in \Lambda$ *over* Λ' *such that* $[\![\mathtt{I}]\!]$ *is correct. Then the program* $\mathsf{P} \cdot \mathtt{I}$ *is well-formed and* $\text{behaviors}(\mathsf{P} \cdot \mathtt{I}) \subseteq \text{behaviors}(\mathsf{P})$.

Proof. We first prove $\mathsf{P} \cdot \mathtt{I}$ is well-formed. Let $G' \in [\![\mathsf{P} \cdot \mathtt{I}]\!]_{\mathsf{prog}}$. According to Proposition 3.8 there exists $G \in [\![\mathsf{P}]\!]_{\mathsf{prog}}$ such that $G' \in G \cdot [\![\mathtt{I}]\!]$. Because P is well-formed, there exists a well-formed $\mathcal{G}$ such that $\mathcal{G} \sqsubset G$, and thus we can directly conclude from Def. 3.11.

Take an arbitrary v such that there is a hereditarily consistent $\mathcal{G}'$ and $\mathcal{G}' \sqsubset G' \in [\![\mathsf{P} \cdot \mathtt{I}]\!]_{\mathsf{prog}}$. As above, we can obtain G such that $G' \in G \cdot [\![\mathtt{I}]\!]$, and conclude with the second implication of Def. 3.11. ◀

3.4 Compositionally Proving Implementation Correctness

As in 2.8, we introduce *compositional correctness*, simplifying the global correctness condition in Def. 3.11. This allows us to prove an implementation correct without reasoning about an arbitrary collection Λ' of other libraries. Our compositional correctness condition in this general weak memory (consistency) setting is inspired by the *local soundness* condition in [24].

3.4.1 Preliminaries

We begin with a few definitions that allow us to express *how* a low-level execution relates to a high-level one by *matching* a high-level method call with all low-level events that constitute its implementation. First, we define an operation to transport relations along maps.

▶ **Definition 3.13.** *Given two sets* X *and* Y*, an irreflexive relation* $r \subseteq X \times X$ *and a map* $f : X \to Y$*, the irreflexive relation* $f_*(r)$ *on* Y *is defined as follows:*

$$(y_1, y_2) \in f_*(r) \iff y_1 \neq y_2 \wedge \exists x_1 \in f^{-1}(y_1).\ \exists x_2 \in f^{-1}(y_2).\ (x_1, x_2) \in r$$

Using this operation, we can define *intentionally* when a low-level plain execution corresponds to linking a high-level plain execution with an implementation as follows.

▶ **Definition 3.14.** *Given plain executions* $G \in \mathbf{PExec}(\mathsf{L})$ *and* $G' \in \mathbf{PExec}(\Lambda)$ *and an implementation* I *of* L *over* Λ*, a map* $f : G'.E \to G.E$ *is a* plain matching *if:*

1. f *is surjective;*
2. $G.\text{po} = f_*(G'.\text{po})$*; and*
3. $\forall e \in G.E.\ G'\!\downarrow_{f^{-1}(e)} \in I(\text{lab}(e'))$

where $G\lfloor_X$*, with* $X \subseteq G.E$ *denotes the restriction of* G *to the set* X *of events.*

Intuitively, f denotes that the high-level method call e in G is implemented using the set of events $f^{-1}(e)$, and that the high-level program order is determined by that of the low-level execution. As captured by following proposition, such a plain matching *witnesses* the fact that a plain execution results from applying an implementation to a high-level execution.

▶ **Proposition 3.15.** *Given an implementation* I *of* L *over* Λ *and plain executions* $G \in \mathbf{PExec}(\mathsf{L})$ *and* $G' \in \mathbf{PExec}(\Lambda)$*, if* $G' \in G \cdot I$*, then there exists a matching* $f : G' \twoheadrightarrow G$*.*

3.4.2 A compositional criterion

We state our *compositional correctness* criterion as a lifting problem as follows. Given a *plain* matching $f : |\mathcal{G}'| \twoheadrightarrow G$, is it possible to find an execution $\mathcal{G}$ that refines G ($\mathcal{G} \sqsubset G$) such that $f : \mathcal{G}' \twoheadrightarrow \mathcal{G}$ is a matching at the level of executions, in the following sense?

▶ **Definition 3.16.** *Given executions* $\mathcal{G}$ *and* $\mathcal{G}'$ *and a map* f *such that* $f : |\mathcal{G}'| \twoheadrightarrow |\mathcal{G}|$*,* f *is a* refined matching*, written* $f : \mathcal{G}' \twoheadrightarrow \mathcal{G}$*, if the following hold:*

1. $\mathcal{G}$ *and* $\mathcal{G}'$ *are consistent;*
2. $\mathcal{G}.\mathrm{sw}^+ \subseteq f_*(\mathcal{G}'.\mathrm{hb})$*; and*
3. $\mathcal{G}.\mathrm{hb} = (f_*(\mathcal{G}'.\mathrm{hb} \setminus (\mathcal{G}'.\mathrm{sw} \cup \mathcal{G}'.\mathrm{po})^+) \cup \mathcal{G}.\mathrm{po} \cup \mathcal{G}.\mathrm{sw})^+$

Condition 1 captures the intuition that this refined notion of matching lives in an idealized world, where all executions have good properties. Condition 2 states that two high-level events synchronize when there exist two low-level events in their respective implementations that are related by happens-before. That is, $\mathcal{G}.\mathrm{sw}$ edges cannot appear out of thin air and must be justified by the implementation. Condition 3 states that the high-level happens-before order is determined by its own po and sw orders as well as the external happens-before order. The external order is computed by removing the $\mathcal{G}'.\mathrm{po}$ and $\mathcal{G}'.\mathrm{sw}$ contributions from $\mathcal{G}'.\mathrm{hb}$ and mapping the remaining $\mathcal{G}'.\mathrm{hb}$ to $\mathcal{G}$ using f_*. Intuitively, this external happens-before is a remnant of other libraries that are ignored by focussing on library L, and being compatible with it allows consistency not to depend on these other libraries.

We consider implementations that are *locally well-formed* in that they preserve well-formedness of local executions, i.e. executions that only contain events from either L for high-level executions or Λ for low-level ones.

▶ **Definition 3.17.** *An implementation* I *of* L *over* Λ *is* locally well-formed *if, for all plain executions* $G \in \mathbf{PExec}(\mathsf{L})$*, all* $G' \in G \cdot I \subseteq \mathbf{PExec}(\Lambda)$ *and all* $\mathcal{G}$*, if* $\mathcal{G} \sqsubset G$ *and* $\mathcal{G}$ *is well-formed, then there exists* $\mathcal{G}'$ *such that* $\mathcal{G}' \sqsubset G'$ *and* $\mathcal{G}'$ *is well-formed.*

We now state the *compositional correctness* criterion which comprises two parts: a local and a global condition. The local condition states that the lifting problem mentioned above always has a solution $\mathcal{G}$ such that $\mathcal{G}$ corresponds to an immediate prefix $\tilde{\mathcal{G}}'$ of the low-level execution $\mathcal{G}'$. This captures an induction on the property that we assume all low-level executions are hereditarily consistent. As empty executions are consistent, the base case holds, which means that we obtain a sequence of refined matchings, and in particular a witness that the high-level execution is locally consistent.

The global condition then ensures that global consistency predicates of the dependencies also hold. Consider a dependency $\mathsf{L}' \in \Lambda$ and executions $\mathcal{G}' \in \mathbf{Exec}(\Lambda \cup \{\star_{\mathsf{L}'}\})$ and $\mathcal{G} \in \mathbf{Exec}(\{\mathsf{L}, \mathsf{L}', \star_{\mathsf{L}'}\})$ with a plain matching between them. Restricting $\mathcal{G}$ to its L events induces a plain matching, which can be lifted to a refined matching because of the local condition.

As such, the global condition stipulates that the implementation preserve the global well-formedness and global consistency of $\mathcal{G}$ and $\mathcal{G}'$.

▶ **De nition 3.18** (Compositional correctness). *A well-formed implementation I of L over Λ is* compositionally correct *if the following two conditions hold.*

1. *Given executions $\tilde{\mathcal{G}}', \mathcal{G}' \in \mathbf{Exec}(\Lambda)$ and $\tilde{\mathcal{G}} \in \mathbf{Exec}(\mathsf{L})$ and a plain execution $G \in \mathbf{PExec}(\mathsf{L})$, **if** $\mathcal{G}$ is consistent and the following holds[3] (i.e. **if** f is a plain matching between $|\mathcal{G}'|$ and G, $\tilde{\mathcal{G}}'$ is an immediate pre x of $\mathcal{G}'$ and $\tilde{f}$ is obtained by restricting the domain of f to $\tilde{\mathcal{G}}.E$)*

$$\begin{array}{ccc} |\tilde{\mathcal{G}}'| & \hookrightarrow_{im} & |\mathcal{G}'| \\ {\scriptstyle \tilde{f}}\downarrow & & \downarrow{\scriptstyle f} \\ |\tilde{\mathcal{G}}| & \hookrightarrow^{=}_{im} & G \end{array} \qquad \begin{array}{ccc} \tilde{\mathcal{G}}' & \hookrightarrow_{im} & \mathcal{G}' \\ {\scriptstyle \tilde{f}}\downarrow & & \\ \tilde{\mathcal{G}} & & \end{array}$$

   ***then** there exists an execution $\mathcal{G}$ such that $\mathcal{G} \sqsubset G$ and the following holds[4]:*

$$\begin{array}{ccc} \tilde{\mathcal{G}}' & \hookrightarrow_{im} & \mathcal{G}' \\ {\scriptstyle \tilde{f}}\downarrow & & \downarrow{\scriptstyle f} \\ \tilde{\mathcal{G}} & \hookrightarrow^{=}_{im} & \mathcal{G} \end{array}$$

2. *For all $\mathsf{L}' \in \Lambda$, given executions $\mathcal{G}' \in \mathbf{Exec}(\Lambda \cup \{\star_{\mathsf{L}'}\})$ and $\mathcal{G} \in \mathbf{Exec}(\{\mathsf{L}, \mathsf{L}', \star_{\mathsf{L}'}\})$, consider $f : |\mathcal{G}'| \twoheadrightarrow |\mathcal{G}|$ such that for any event e not from L, $f^{-1}(e)$ is a singleton. Let $\mathcal{G}'_{\mathsf{L}'} \triangleq f^{-1}(\mathcal{G} \downharpoonright \mathsf{L}')$ and assume f restricts to $f_{\mathsf{L}'} : \mathcal{G}'_{\mathsf{L}'} \twoheadrightarrow \mathcal{G} \downharpoonright \mathsf{L}'$. Then the following must hold:*
   - *if $\pi_{\mathsf{L}'}(\mathcal{G}) \in \mathsf{L}'.\mathcal{T}_{wf}$, then $\pi_{\mathsf{L}'}(\mathcal{G}') \in \mathsf{L}'.\mathcal{T}_{wf}$; and*
   - *if $\pi_{\mathsf{L}'}(\mathcal{G}') \in \mathsf{L}'.\mathcal{T}_c$, then $\pi_{\mathsf{L}'}(\mathcal{G}) \in \mathsf{L}'.\mathcal{T}_c$.*

▶ **Theorem 3.19.** *Compositional correctness implies correctness.*

4 Case Study: Durable Linearizability with FliT and Mirror

We consider a family of libraries that provide a simple interface with persistent memory accesses (reads and writes), allowing one to convert any linearisable implementation to a durably linearisable one by replacing regular (volatile) accesses with persistent ones supplied by the library. Speci cally, we consider two such libraries FliT [31] and Mirror [9]; we specify them both in our framework, prove their implementations sound against their respective speci cations, and further prove their general result for converting data structures.

4.1 The FliT Library

FliT [31] is a persistent library that provides a simple interface very close to Px86, but with stronger persistency guarantees, which make it easier to implement durable data structures. Speci cally, a FliT object ℓ can be accessed via write and read methods, $\mathtt{wr}_\pi(\ell, v)$ and $\mathtt{rd}_\pi(\ell)$, as well as standard read-modify-write methods. Each write (resp. read) operation has two variants, denoted by the *type* $\pi \in \{\mathsf{p}, \mathsf{v}\}$. This type speci es if the write (resp. read) is *persistent* ($\pi = \mathsf{p}$) in that its e ects must be persisted, or *volatile* ($\pi = \mathsf{v}$) in that its

[3] The right diagram should be seen as sitting above the left one, where $\mathcal{G}$ being above G means that $\mathcal{G} \sqsubset G$.
[4] This amounts to completing the diagram sitting on top to obtain a cube.

```
method wr_π(ℓ, v) :
  if π = p then
    fetch-and-add( it-counter(ℓ), 1);
    write(ℓ, v);
    flush_opt(ℓ);
    fetch-and-add( it-counter(ℓ), −1);
  else
    sfence;
    write(ℓ, v);

method rd_π(ℓ) :
  local v = read(ℓ);
  if π = p ∧ it-counter(ℓ) > 0 then
    flush_opt(ℓ);
  return v;

method finishOp :
  sfence;
```

Figure 5 FliT library implementation in Px86

persistency has been optimised and o ers weaker guarantees. The default access type is persistent (p), and the volatile accesses may be used as optimizations when weaker guarantees su ce. Wei et al. [31] introduce a notion of *dependency* between di erent operations as follows. If a (persistent or volatile) write w depends on a persistent write w', then w' persists before w. If a persistent read r reads from a persistent write w, then r depends on w and thus w must be persisted upon reading if it has not already persisted. Though simple, FliT provides a strong guarantee as captured by a general result for correctly converting volatile data structures to persistent ones: if one replaces every memory access in the implementation of a *linearizable* data-structure with the corresponding persistent FliT access, then the resulting data structure is *durably linearizable*.

Compared to the original FliT development, our soundness proof is more formal and detailed: it is established against a formal speci cation (rather than an English description) and with respect to the formal Px86 model.

FliT Interface. The FliT interface uses the $\mathrm{P}^{\mathsf{Px86}}$ from Px86 and contains a single constructor, new, allocating a new FliT location, as well as three other methods below, the last two of which are durable:

- $\mathtt{rd}_\pi(\ell)$ with $\pi \in \{\mathsf{p}, \mathsf{v}\}$, for a π-read from ℓ;
- $\mathtt{wr}_\pi(\ell, v)$ with $\pi \in \{\mathsf{p}, \mathsf{v}\}$, denoting a π-write of value $v \in \mathbf{Val}$ to ℓ; and
- `finishOp`, which waits for previously executed operations to persist.

We write R and W respectively for the read and write events, and add the superscript π (e.g. R^{p}) to denote such events with the given persistency mode.

FliT Speci cation. We develop a formal speci cation of FliT in our framework, based on its original informal description. The correctness of FliT executions is described via a *dependency* relation that contains the program order and the total execution (linearization) order restricted to persistent write-read operations on the same location. Note that this dependency notion is stronger than the customary de nitions that use a rf relation (as in the Px86 speci cation) instead of lin, because a persistent read may not read directly from a persistent write w, but rather from another later (lin-after w) write.

▶ **De nition 4.1** (FliT execution Correctness). *A FliT execution $\mathcal{G}$ is correct if there exists a reads-from relation* rf *and a total order* lin $\supseteq \mathcal{G}.\mathrm{hb}$ *on $\mathcal{G}.E$ and an order* nvo *such that:*

1. *Each read event reads from the most recent previous write to the same location:*
   $\mathrm{rf} = \bigcup_{\ell \in \mathbf{Loc}} ([W_\ell]; \mathrm{lin}; [R_\ell]) \setminus (\mathrm{lin}; [W_\ell]; \mathrm{lin})$
2. *Reads return the value written by the write they read from:*
   $(w, r) \in \mathrm{rf} \Rightarrow \exists \ell, \pi, \pi', v.\ \mathrm{lab}(r) = \mathtt{rd}_{\pi'}(\ell) : v \wedge \mathrm{lab}(w) = \mathtt{wr}_\pi(\ell, v) : -$
3. *Persistent writes persist before every other later dependent write:*
   $[W^{\mathsf{p}}]; (\mathrm{po} \cup \bigcup_{\ell \in \mathbf{Loc}} [W^{\mathsf{p}}_\ell]; \mathrm{lin}; [R^{\mathsf{p}}_\ell])^+; [W] \subseteq \mathrm{nvo}$

4. *Persistent writes before a* `finishOp` *persist:*
   $dom([W^{\mathsf{P}}]; (\mathrm{po} \cup \bigcup_{\ell \in \mathbf{Loc}} [W^{\mathsf{P}}_\ell]; \mathrm{lin}; [R^{\mathsf{P}}_\ell])^+; [\texttt{finishOp}]) \subseteq \lfloor P^{Px86} \rfloor$
5. *And* nvo *is a persist order:* $dom(\mathrm{nvo}; \lfloor Ptag \rfloor) \subseteq \lfloor Ptag \rfloor$.

Px86 implementation of FliT. The implementation of FliT methods is given in Fig. 5. Whereas a naive implementation of this interface would have to issue a ush instruction both after persistent writes and in persistent reads, the implementation shown associates each location with a counter to avoid performing super uous ushes when reading from a location whose value has already persisted. Speci cally, a persistent write on ℓ increments its counter before writing to and ushing it, and decrements the counter afterwards. As such, persistent reads only need to issue a ush if the counter is positive (i.e. if there is a concurrent write that has not executed its ush yet).

▶ **Theorem 4.2.** *The implementation of FliT in Fig. 5 is correct.*

FliT and Durable Linearizability. Given a data structure implementation I, let $p(I)$ denote the implementation obtained from I by 1) replacing reads/writes in the implementation with their corresponding persistent FliT instructions, and 2) adding a call to `finishOp` right before the end of each method. We then show that given an implementation I, if I is linearizable, then $p(I)$ is *durably linearizable*[5]. We assume that all method implementations are single-threaded, i.e. all plain executions $I(m(\vec{v}))$ are totally ordered.

▶ **Theorem 4.3.** *If* $Px86 \vDash I : Lin(S)$, *then* $FliT \vDash p(I) : DurLin(S)$.

4.2 The Mirror Library

The Mirror [9] persistent library has similar goals to FliT. The main di erence between the two is that Mirror operations do not o er two variants, and their operations are implemented di erently from those of FliT. Speci cally, in Mirror each location has two copies: one in persistent memory to ensure durability, and one in volatile memory for fast access. As such, read operations are implemented as simple loads from volatile memory, while writes have a more involved implementation than those of FliT.

We present the Mirror speci cation and implementation in the technical appendix where we also prove that its implementation is correct against its speci cation. As with FliT, we further prove that Mirror can be used to convert linearizable data structures to durably linearizable ones, as described above.

5 Case Study: Persistent Transactional Library

We revisit the $\mathsf{L}_{\mathsf{trans}}$ transactional library, develop its formal speci cation and verify its implementation (Fig. 1) against it. Recall the simple $\mathsf{L}_{\mathsf{trans}}$ implementation in Fig. 1 and that we do not allow for nested transactions. The implementation uses an *undo-log* which records the former values of persistent registers (locations) modi ed in a transaction. If, after a crash, the recovery mechanism detects a partially persisted transaction (i.e. the last entry in the undo log is not COMMITTED), then it can use the undo-log to restore registers to their former values. The implementation uses a durably linearizable queue library[6] Q, and assumes that

[5] The de nition here is the same as in 2, as hb-linearizations of the execution still yield sequential executions.
[6] For example, take any linearizable queue implementation and use the FliT library as described in 4.

it is *externally synchronized*: the user is responsible for ensuring no two transactions are executed in parallel. We formalize this using a global well-formedness condition.

Later in 5.2 we develop a wrapper library $\mathsf{L}_{\mathsf{Strans}}$ for $\mathsf{L}_{\mathsf{trans}}$ that additionally provides synchronization using locks and prove that our implementation of this library is correct. To do this, we need to make small modi cations to the structure of the speci cation: the speci cation in 2 requires that any transaction-aware operation (i.e. those tagged with T) be enclosed in calls to `PTBegin` and `PTEnd`. Since $\mathsf{L}_{\mathsf{Strans}}$ wraps the calls to `PTBegin` and `PTEnd`, the well-formedness condition needs to be generalized to allow operations tagged with T to appear between calls to operations that behave like `PTBegin` and `PTEnd`. To that end, we add two new tags B and E to denote such operations, respectively.

5.1 Speci cation

The $\mathsf{L}_{\mathsf{trans}}$ library provides four *tags*: 1) T for transaction-aware client operations; 2) P^{tr} for operations that have persisted using transactions; and 3) B, E for operations that begin and end transactions, respectively. We write $\mathcal{R}, \mathcal{W}, \mathcal{B}, \mathcal{E}, \mathcal{RC}$ respectively for the sets of events labeled with read, write, begin, end and recovery methods. As before, we write e.g. $\lfloor \mathrm{T} \rfloor$ for the set of events tagged with T. Note that while $\mathcal{B}$ denotes the set of the begin events in library $\mathsf{L}_{\mathsf{trans}}$, the $\lfloor \mathrm{B} \rfloor$ denotes the set of all events that are tagged with B, which includes $\mathcal{B}$ (of library $\mathsf{L}_{\mathsf{trans}}$) as well as events of *other* (non-$\mathsf{L}_{\mathsf{trans}}$) libraries that may be tagged with B; similarly for $\mathcal{E}$ and $\lfloor \mathrm{E} \rfloor$. As such, our local speci cations below (i.e. local well-formedness and consistency) are de ned in terms of $\mathcal{B}$ and $\mathcal{E}$, whereas our global speci cations are de ned in terms of $\lfloor \mathrm{B} \rfloor$ and $\lfloor \mathrm{E} \rfloor$. As before, for brevity we write e.g. $[\mathrm{T}]$ as a shorthand for the relation $[\lfloor \mathrm{T} \rfloor]$. We next de ne the same-transaction relation strans:

$$\mathsf{strans} \triangleq [\lfloor \mathrm{B} \rfloor \cup \lfloor \mathrm{E} \rfloor \cup \lfloor \mathrm{T} \rfloor]; (\mathsf{po} \cup \mathsf{po}^{-1}); [\lfloor \mathrm{B} \rfloor \cup \lfloor \mathrm{E} \rfloor \cup \lfloor \mathrm{T} \rfloor] \setminus ((\mathsf{po}; [\mathrm{E}]; \mathsf{po}) \cup (\mathsf{po}; [\mathrm{B}]; \mathsf{po}))$$

An execution is locally well-formed i the following hold:

1. A transaction must be opened before it is closed: $\mathcal{E} \subseteq \mathit{rng}([\mathcal{B}]; \mathsf{po})$
2. Transactions are not nested and are matching: $[\mathcal{E}]; \mathsf{po}; [\mathcal{E}] \subseteq [\mathcal{E}]; \mathsf{po}; [\mathcal{B}]; \mathsf{po}; [\mathcal{E}]$ and $[\mathcal{B}]; \mathsf{po}; [\mathcal{B}] \subseteq [\mathcal{B}]; \mathsf{po}; [\mathcal{E}]; \mathsf{po}; [\mathcal{B}]$
3. Transactions must be externally synchronized: $\mathcal{E} \times \mathcal{B} \subseteq \mathsf{hb} \cup \mathsf{hb}^{-1}$
4. The recovery routine must be called after a crash: $\lightning; \mathsf{hb}; \lfloor \mathrm{B} \rfloor \subseteq \lightning; \mathsf{hb}; [\mathcal{RC}]; \mathsf{hb}; \lfloor \mathrm{B} \rfloor$
5. Events are correctly tagged: $\mathcal{W} \cup \mathcal{R} \subseteq \lfloor \mathrm{T} \rfloor$

An execution is globally well-formed if client operations (T-tagged) are inside transactions:

6. $\lfloor \mathrm{T} \rfloor \subseteq \mathit{rng}([\mathrm{B}]; \mathsf{po})$
7. $[\mathrm{E}]; \mathsf{po}; [\mathrm{T}] \subseteq [\mathrm{E}]; \mathsf{po}; [\mathrm{B}]; \mathsf{po}; [\mathrm{T}]$

An execution is locally-consistent if there exists a reads-from relation rf such that:

8. rf relates writes to reads, $\mathsf{rf} \subseteq \mathcal{W} \times \mathcal{R}$, such that each read is related to exactly one write (i.e. rf^{-1} is total and functional).
9. Reads access the most recent write: $\mathsf{rf}^{-1}; \mathsf{hb} \subseteq \mathsf{hb}$
10. External reads (reading from a di erent transaction) read from persisted writes: $\mathit{dom}(\mathsf{rf} \setminus \mathsf{strans}) \subseteq \lfloor \mathrm{P}^{\mathsf{tr}} \rfloor$

An execution is globally-consistent if there exists an order nvo over $\lfloor \mathrm{T} \rfloor$ such that:

11. Transactions are nvo-ordered: $[\mathrm{E}]; \mathsf{hb}; [\mathrm{B}] \subseteq \mathsf{nvo}$

12. nvo is the persistance order: $dom(\mathrm{nvo}; [\mathrm{P}^{\mathsf{tr}}]) \subseteq \lfloor \mathrm{P}^{\mathsf{tr}} \rfloor$;
13. Either all or none of the events in a transaction persist (atomicity): $[\mathrm{P}^{\mathsf{tr}}]; \mathrm{strans}; [\mathrm{T}] \subseteq [\mathrm{P}^{\mathsf{tr}}]$
14. All events of a completed transaction (ones with an associated end event) persist: $\lfloor \mathrm{E} \rfloor^c \subseteq \lfloor \mathrm{P}^{\mathsf{tr}} \rfloor$, where $\lfloor \mathrm{E} \rfloor^c$ denotes the set of method calls tagged with E which have completed.

▶ **Theorem 5.1.** *The* L_{trans} *implementation in Fig. 1 over Px86 is correct.*

5.2 Vertical Library Composition: Adding Internal Synchronization

We next demonstrate how our framework can be used for *vertical library composition*, where an implementation of one library comprises calls to other libraries with non-trivial global speci cations. To this end, we develop $\mathsf{L}_{\mathsf{Strans}}$, a wrapper library around $\mathsf{L}_{\mathsf{trans}}$ that is meant to be simpler to use by providing synchronization internally: rather than the user ensuring synchronization for $\mathsf{L}_{\mathsf{trans}}$, one can use $\mathsf{L}_{\mathsf{Strans}}$ to prevent two transactions from executing in parallel. More formally, the well-formedness condition (3) of $\mathsf{L}_{\mathsf{trans}}$ becomes a correctness guarantee of $\mathsf{L}_{\mathsf{Strans}}$. We consider a simple implementation of $\mathsf{L}_{\mathsf{Strans}}$ that uses a global lock acquired at the beginning of each transaction and released at the end as shown below.

globals lock := L.new()

method LPTBegin() := L.acq(lock);PTBegin()
method LPTEnd() := PTEnd();L.rel(lock)

▶ **Theorem 5.2.** *The implementation of* L_{Strans} *above is correct.*

Using compositional correctness, the main proof obligation is condition 2 of Def. 3.18 stipulating that the implementation be well-formed, ensuring that $\mathsf{L}_{\mathsf{trans}}$ is used correctly by the $\mathsf{L}_{\mathsf{Strans}}$ implementation. This is straightforward as we can assume there exists an immediate pre x that is consistent (Def. 3.10). The existence of the hb-ordering of calls to `PTBegin` and `PTEnd` follows from the consistency of the global lock used by the implementation.

5.3 Horizontal Library Composition

We next demonstrate how our framework can be used for *horizontal library composition*, where a *client* program comprises calls to multiple libraries. To this end, we develop a simple library, $\mathsf{L}_{\mathsf{cntr}}$, providing a persistent counter to be used in *sequential* (single-threaded) settings. As such, if a client intends to use $\mathsf{L}_{\mathsf{cntr}}$ in concurrent settings, they must call its methods only within critical sections. The $\mathsf{L}_{\mathsf{cntr}}$ provides three operations to create (NewCounter), increment (CounterInc) and read a counter (CounterRead). The speci cation and implementation of $\mathsf{L}_{\mathsf{cntr}}$ are given in the technical appendix.

As $\mathsf{L}_{\mathsf{cntr}}$ uses the tags of $\mathsf{L}_{\mathsf{trans}}$, we de ne $\mathsf{L}_{\mathsf{cntr}}.\Lambda_{\mathrm{tags}} \triangleq \{\mathsf{L}_{\mathsf{trans}}\}$. The all the operations are tagged with T. As such, $\mathsf{L}_{\mathsf{cntr}}$ inherits the global well-formedness condition of $\mathsf{L}_{\mathsf{trans}}$, meaning that $\mathsf{L}_{\mathsf{cntr}}$ operations must be used within transactions (i.e. hb-between operations respectively tagged with B and E). Putting it all together, the following client code snippet uses $\mathsf{L}_{\mathsf{cntr}}$ in a correct way, even though $\mathsf{L}_{\mathsf{cntr}}$ has no knowledge of the existence of $\mathsf{L}_{\mathsf{Strans}}$.

```
c = NewCounter(); LPTBegin(); CounterInc(c); CounterInc(c); LPTEnd();
```

Speci cally, the above is an instance of horizontal library composition (as the client comprises calls to both $\mathsf{L}_{\mathsf{Strans}}$ and $\mathsf{L}_{\mathsf{cntr}}$), facilitated in our framework through global speci cations.

6 Conclusions, Related and Future Work

We presented a framework for specifying and verifying persistent libraries, and demonstrated its utility and generality by encoding existing correctness notions within it and proving the correctness of the FliT and Mirror libraries, as well as a persistent transactional library.

Related Work. The most closely related body of work to ours is [24]. However, while their framework can be used for specifying only the consistency guarantees of a library, ours can be used to specify both consistency and persistency guarantees. Existing literature includes several works on formal persistency models, both for hardware [23, 28, 29, 4, 5, 17, 27, 26] and software [3, 19, 10], as well as correctness conditions for persistent libraries such as durable linearizability [16]. As we showed in 3, such models can be speci ed in our framework.

Additionally, there are several works on implementing and verifying algorithms that operate on NVM. [8] and [32] respectively developed persistent queue and set implementations in Px86. [7] provided a formal correctness proof of the implementation in [32]. All three of [7, 32, 8] assume that the underlying concurrency model is SC [21], rather than that of Px86 (namely TSO). As we demonstrated in 4 5 we can use our framework to verify persistent implementations *modularly* while remaining faithful to the underlying concurrency model. [25, 2] have developed persistent program logics for verifying programs under Px86. [18] recently formalized the consistency and persistency semantics of the Linux ext4 le system, and developed a model-checking algorithm and tool for verifying the consistency and persistency behaviors of ext4 applications such as text editors.

Future Work. We believe our framework will pave the way for further work on verifying persistent libraries, whether manually (as done here), possibly with the assistance of an interactive theorem prover and/or program logics such as those of [6, 25, 2], or automatically via model checking. The work of [6] uses the framework of [24] to specify data structures in a program logic, and it would be natural to extend it to our framework for persistency. Existing work in the latter research direction, e.g. [11, 18], has so far only considered low-level properties, such as the absence of races or the preservation of user-supplied invariants. It has not yet considered higher-level functional correctness properties, such as durable linearizability and its variants. We believe our framework will be helpful in that regard. In a more theoretical direction, it would be interesting to understand how our compositional correctness theorem ts in general settings for abstract logical relations such as [15].

References

1 Aguilera, M.K., Frolund, S.: Strict linearizability and the power of aborting. Tech. Rep. HPL-2003-241 (2013)

2 Bila, E.V., Dongol, B., Lahav, O., Raad, A., Wickerson, J.: View-based owicki gries reasoning for persistent x86-tso. In: Sergey, I. (ed.) Programming Languages and Systems. pp. 234 261. Springer International Publishing, Cham (2022)

3 Chakrabarti, D.R., Boehm, H.J., Bhandari, K.: Atlas: Leveraging locks for non-volatile memory consistency. SIGPLAN Not. **49**(10), 433 452 (Oct 2014). https://doi.org/10.1145/2714064.2660224, http://doi.acm.org/10.1145/2714064.2660224

4 Cho, K., Lee, S.H., Raad, A., Kang, J.: Revamping hardware persistency models: View-based and axiomatic persistency models for Intel-X86 and Armv8. p. 16 31. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021)

5 Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B., Burger, D., Coetzee, D.: Better I/O through byte-addressable, persistent memory. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. pp. 133 146. SOSP 09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1629575.1629589, http://doi.acm.org/10.1145/1629575.1629589

6 Dang, H.H., Jung, J., Choi, J., Nguyen, D.T., Mansky, W., Kang, J., Dreyer, D.: Compass: Strong and compositional library speci cations in relaxed memory separation logic. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2022 (2022)

7 Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Wehrheim, H.: Verifying correctness of persistent concurrent data structures. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Formal Methods The Next 30 Years. pp. 179 195. Springer International Publishing, Cham (2019)

8 Friedman, M., Herlihy, M., Marathe, V., Petrank, E.: A persistent lock-free queue for non-volatile memory. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 28 40. PPoPP 18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3178487.3178490, https://doi.org/10.1145/3178487.3178490

9 Friedman, M., Petrank, E., Ramalhete, P.: Mirror: making lock-free data structures persistent. In: Freund, S.N., Yahav, E. (eds.) PLDI 21. pp. 1218 1232 (2021)

10 Gogte, V., Diestelhorst, S., Wang, W., Narayanasamy, S., Chen, P.M., Wenisch, T.F.: Persistency for synchronization-free regions. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 46 61. PLDI 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3192366.3192367, http://doi.acm.org/10.1145/3192366.3192367

11 Gorjiara, H., Xu, G.H., Demsky, B.: Yashme: Detecting persistency races. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. p. 830 845. ASPLOS 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3503222.3507766, https://doi.org/10.1145/3503222.3507766

12 Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X.N., Weng, S., Zhang, H., Guo, Y.: Deep speci cations and certi ed abstraction layers. In: POPL (2015)

13 Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)

14 Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463 492 (1990). https://doi.org/10.1145/78969.78972

15 Hermida, C., Reddy, U.S., Robinson, E.P.: Logical relations and parametricity a reynolds programme for category theory and programming languages. Electronic Notes in Theoretical Computer Science **303**, 149 180 (2014), proceedings of the Workshop on Algebra, Coalgebra and Topology (WACT 2013)

16 Izraelevitz, J., Mendes, H., Scott, M.L.: Linearizability of persistent memory objects under a full-system-crash failure model. In: Gavoille, C., Ilcinkas, D. (eds.) DISC. Lecture Notes in Computer Science, vol. 9888, pp. 313 327 (2016)

17 Khyzha, A., Lahav, O.: Taming x86-tso persistency. Proc. ACM Program. Lang. **5**(POPL), 1 29 (2021)

18 Kokologiannakis, M., Kaysin, I., Raad, A., Vafeiadis, V.: Persevere: Persistency semantics for veri cation under ext4. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434324, https://doi.org/10.1145/3434324

19 Kolli, A., Gogte, V., Saidi, A., Diestelhorst, S., Chen, P.M., Narayanasamy, S., Wenisch, T.F.: Language-level persistency. In: Proceedings of the 44th Annual International Symposium on Computer Architecture. pp. 481 493. ISCA 17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3079856.3080229, http://doi.acm.org/10.1145/3079856.3080229

20 Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in c/c++11. SIGPLAN Not. **52**(6), 618 632 (jun 2017). https://doi.org/10.1145/3140587.3062352, https://doi.org/10.1145/3140587.3062352

21 Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690 691 (Sep 1979). https://doi.org/10.1109/TC.1979.1675439, http://dx.doi.org/10.1109/TC.1979.1675439

22 Pelley, S., Chen, P.M., Wenisch, T.F.: Memory persistency. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture. pp. 265 276. ISCA 14, IEEE Press, Piscataway, NJ, USA (2014), http://dl.acm.org/citation.cfm?id=2665671.2665712

23 Pelley, S., Chen, P.M., Wenisch, T.F.: Memory persistency. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture. p. 265 276. ISCA 14, IEEE Press (2014)

24 Raad, A., Doko, M., Ro i , L., Lahav, O., Vafeiadis, V.: On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models. POPL (2019)

25 Raad, A., Lahav, O., Vafeiadis, V.: Persistent owicki-gries reasoning: A program logic for reasoning about persistent programs on intel-x86. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428219, https://doi.org/10.1145/3428219

26 Raad, A., Maranget, L., Vafeiadis, V.: Extending intel-x86 consistency and persistency: formalising the semantics of intel-x86 memory types and non-temporal stores. Proc. ACM Program. Lang. **6**(POPL), 1 31 (2022)

27 Raad, A., Vafeiadis, V.: Persistence semantics for weak memory: Integrating epoch persistency with the tso memory model. Proc. ACM Program. Lang. **2**(OOPSLA), 137:1 137:27 (Oct 2018). https://doi.org/10.1145/3276507, http://doi.acm.org/10.1145/3276507

28 Raad, A., Wickerson, J., Neiger, G., Vafeiadis, V.: Persistency semantics of the intel-x86 architecture. Proc. ACM Program. Lang. **4**(POPL), 11:1 11:31 (2020)

29 Raad, A., Wickerson, J., Vafeiadis, V.: Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models. Proc. ACM Program. Lang. **3**(OOPSLA), 135:1 135:27 (Oct 2019)

30 Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: X86-TSO: A rigorous and usable programmer s model for x86 multiprocessors. Commun. ACM **53**(7), 89 97 (Jul 2010). https://doi.org/10.1145/1785414.1785443, http://doi.acm.org/10.1145/1785414.1785443

31 Wei, Y., Ben-David, N., Friedman, M., Blelloch, G.E., Petrank, E.: Flit: a library for simple and e cient persistent algorithms. In: Lee, J., Agrawal, K., Spear, M.F. (eds.) PPoPP 22. pp. 309 321 (2022)

32 Zuriel, Y., Friedman, M., She , G., Cohen, N., Petrank, E.: E cient lock-free durable sets.

Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360554, https://doi.org/10.1145/3360554

A Compositional soundness

We prove Theorem 3.12, the adequacy of the compositional correctness criterion. We first prove the following lemma:

▶ **Lemma A.1.** *Consider an implementation I of L on top of Λ and let $\mathcal{G}'$ be a consistent and closed execution of Λ', Λ. and G' be a plain execution of Λ', L, such that $|\mathcal{G}'| \in G \cdot I$. Let $f : |\mathcal{G}'| \to G$ be the map given by Prop 3.15. Let G_L be the restriction of G to events in L and $\mathcal{G}'_L \coloneqq \mathcal{G}'\restriction_{f^{-1}(G_L.E)}$ be the corresponding restriction of $\mathcal{G}'$. Let $f_L : |\mathcal{G}'_L| \twoheadrightarrow G_L$ be the corresponding restriction of f.*

If there exists a lifting $f_L : \mathcal{G}'_L \twoheadrightarrow \mathcal{G}_L$ with $\mathcal{G}_L \sqsubset G_L$, then the plain execution G' can be lifted to a closed and locally consistent execution $\mathcal{G}' \sqsubset G'$ such that $\mathcal{G}'.\mathrm{hb} \subseteq f_*(\mathcal{G}.\mathrm{hb})$. In particular, $(\mathrm{po}' \cup \mathrm{sw}')^+$ is acyclic.

Proof. First, let us define an execution $\mathcal{G}$ refining G using $\mathcal{G}'_L$, and $\mathcal{G}$. Let sw be $(f_*(\mathcal{G}'.\mathrm{sw})'_\Lambda) \cup \mathcal{G}_L.\mathrm{sw}$ (we use implicitly the inclusion $G_L \to G$). Define

$$\mathcal{G} \coloneqq \langle G, \mathrm{sw}, (\mathrm{po} \cup \mathrm{sw})^+ \rangle$$

To prove that $\mathcal{G}'_L$ is well-formed according to the specification of the library L', it suffices to prove that $\mathcal{G}'.\mathrm{hb} \downharpoonright \mathsf{L}'$ is acyclic for $\mathsf{L}' \in \Lambda' \cup \{\mathsf{L}\}$. As we assume that $\mathcal{G}'$ is consistent, it suffices to prove that $\mathcal{G}'.\mathrm{hb} = (\mathrm{po}' \cup \mathrm{sw}')^+$ is acyclic. We assume by contradiction that there exists a cycle $x_1, \dots, x_n = x_1$ in this relation; the contradition will be the existence of a cycle in $\mathcal{G}_L.\mathrm{hb}$.

Because f is surjective, for all i, there are three possible cases (in this proof, primed relations are components of $\mathcal{G}'$ and unprimed are components of $\mathcal{G}$):

1. $x_i \xrightarrow{\mathrm{po}} x_{i+1}$, and then there exist y'_i and y_{i+1} in G' such that $y'_i \xrightarrow{\mathrm{po}'} y_{i+1}$ and $f(y'_i) \neq f(y_{i+1})$;
2. $x_i \xrightarrow{f_*(\mathrm{sw}') \cap \mathsf{L}^c} x_{i+1}$, then $x_i, x_{i+1} \notin \mathsf{L}$, and there exist y'_i and y_{i+1} in G' such that $y'_i \xrightarrow{\mathrm{sw}'} y_{i+1}$;
3. $x_i \xrightarrow{\mathrm{sw}_L} x_{i+1}$, then $x_i, x_{i+1} \in \mathsf{L}$, and there exist y'_i and y_{i+1} in G'_L such that $f(y'_i) \xrightarrow{\mathrm{sw}_L} f(y_{i+1})$.

Moreover, the ys can be chosen such that $f(y'_i) = f(y_i) = x_i$. If none of the x_i are in L, it is easy to construct a cycle in G, and so we can assume without loss of generality that $x_1 \in \mathsf{L}$.

Consider now the subsequence $x_{\sigma(j)}$ of the elements of (x_i) which are in L. Then the $z_j \coloneqq y_{\sigma(j)}$ and the $z_j \coloneqq y'_{\sigma(j)}$ are in G_L, and, for each j, there are three cases:

1. $z'_j \xrightarrow{\mathrm{po}'} z_{j+1}$,
2. $z'_j \xrightarrow{f^{-1}(\mathrm{sw}_L)} z_{j+1}$,
3. $z'_j \xrightarrow{\mathrm{hb}' \setminus (\mathrm{sw}' \cup \mathrm{po}')} z_{j+1}$

Thus, the sequence $f(z_j)$ is a cycle in $\mathcal{G}'_L$ for the relation $f_*(\mathcal{G}'.\mathrm{hb} \setminus (\mathrm{sw}' \cup \mathrm{po}')) \cup \mathrm{po} \cup \mathrm{sw}$, which contradicts the fact that $\mathcal{G}_L$ is well-formed. ◀

We can now prove Theorem 3.12:

Proof. We consider a well-formed implementation I of L on top of Λ, as well as a collection Λ' of libraries and plain executions $G \in \mathbf{PExec}(\{\mathsf{L}\} \cup \Lambda)$ and $G' \in G \cdot I \subseteq \mathbf{PExec}(\Lambda \cup \Lambda')$. The

rst part of Def. 3.11 follows immediately from the fact that I is well-formed. Let us then prove the second part, and consider $\mathcal{G}' \sqsubset G'$ well-formed, consistent and closed. We need to nd $\mathcal{G} \sqsubset G$ which is consistent and closed. Since $\mathcal{G}'$ is hereditarily consistent, there exists a sequence

$$\emptyset = \mathcal{G}'_0 \hookrightarrow_{im} \mathcal{G}'_1 \hookrightarrow_{im} \cdots \hookrightarrow_{im} \mathcal{G}'_n = \mathcal{G}'$$

of consistent closed executions of $\Lambda \cup \Lambda'$. As above, by considering the images of these subexecutions under f, we obtain

$$\emptyset = G_0 \hookrightarrow^{=}_{im} G_1 \hookrightarrow^{=}_{im} \cdots \hookrightarrow^{=}_{im} G_n = G$$

with $f_i : |\mathcal{G}_i| \twoheadrightarrow G_i$. Further, by taking the restrictions of G_i to the library $\mathcal{G}$, we obtain a sequence of $f_i^{\mathsf{L}} : |\mathcal{G}_i'^{\mathsf{L}}| \twoheadrightarrow G_i^{\mathsf{L}}$. The rst plain matching f_0^{L} lifts trivially to a re ned matching, and applying the assumption that I is sound n times, we lift all the f_i^{L} to re ned mathchings, and we get a sequence of consistent executions:

$$\emptyset = \mathcal{G}_0^{\mathsf{L}} \hookrightarrow^{=}_{im} \mathcal{G}_1^{\mathsf{L}} \hookrightarrow^{=}_{im} \cdots \hookrightarrow^{=}_{im} \mathcal{G}_n^{\mathsf{L}} = \mathcal{G}^{\mathsf{L}}$$

Applying the lemma above, we get a sequence of consistent and closed executions $\mathcal{G}_i \sqsubset G_i$, which are related by $\hookrightarrow^{=}_{im}$ because of the restriction to sw in the de nition of re ned matching and the restriction on po in the de nition of plain matching. This concludes the proof. ◀

B Correctness of the FliT Implementation

▶ **Theorem 4.2.** *The implementation of FliT in Fig. 5 is correct.*

Proof. We use our modularity theorem. Let $\mathcal{G}' = \langle E', \mathrm{po}', \mathrm{sw}', \mathrm{hb}' \rangle$ be a consistent Px86 execution, and let rf' and tso be witnesses. $G = \langle E, \mathrm{po} \rangle$ be a plain FliT execution and let $f : |\mathcal{G}'| \twoheadrightarrow G$ be a plain matching. We then de ne $\mathrm{lin} \triangleq f_*(\mathrm{tso}'_M)$, where tso_M denotes the restriction of tso to events on location ℓ. The order lin is total, because a read po-after a write to the same location is also ordered by tso thanks to the fetch-and-add operation after the write. We now prove that the relation lin satis es the four properties in the speci cation of FliT.

Proof of (1) *and* (2). The lin-maximal write to the location is the tso' maximal write to that location before a crash, or the nvo'-maximal such write before a crash.

Proof of (3). First, notice that $\mathrm{po} \subseteq \mathrm{nvo}$, because of the ush and the barrier. Moreover, $[W^{\mathsf{p}}_\ell]; f_*(\mathrm{tso}'_M); [R^{\mathsf{p}}_\ell]; \mathrm{po}'; [W] \subseteq \mathrm{nvo}$: write w, r, w' for the two writes and the read which are mentioned in this expression. There are several cases. First, notice that if the two writes are executed on on the same thread, then this follows from the rst point. There are now two cases remaining, depending on whether the read operation goes though its fast path, or whether it executes the optimized ush.

1. *Fast path.* If the read operation reads 0 in the it-counter, then the write it read must have been mo'-after or equal to the decreasing FAA operation executed by the write operation which was read. In particular, this means that any write which is po'-after that read is tso'-after, and therefore nvo'-after, the decreasing FAA, which is nvo'-after the rst write.
2. *Slow path.* Any write which is tso'-after the read operation is nvo-after its optimized ush, which is tso'-after the write operation.

Then, the nal write executes either a fence or an RMW before its linearization point, so that $(r, w') \in \mathrm{nvo}$.

Proof of (4). Follows from the properties of ushes in Px86. ◀

In this proof, it is useful to change the causal structures of executions to make incomplete events maximal in order to create a crashless execution from a chain. Given a plain execution $\langle E, \mathrm{po}\rangle$ and a set of events $E' \subseteq E$, we de ne $\mathsf{detached}(E', \langle E, \mathrm{po}\rangle) \triangleq \langle E, \mathrm{po} \setminus (E' \times (E \setminus E'))\rangle$, i.e. E' are made po-maximal in $\mathsf{detached}(E', \langle E, \mathrm{po}\rangle)$.

▶ **Theorem 4.3.** *If* $\mathit{Px86} \vDash I : \mathit{Lin}(S)$*, then* $\mathit{FliT} \vDash p(I) : \mathit{DurLin}(S)$*.*

Proof. Consider a plain execution G of $\mathsf{DurLin}(S)$, and a corresponding consistent program execution $\mathcal{G}'$ such that $|\mathcal{G}'| \in G \cdot p(I)$. Let rf' be a witness that $\mathcal{G}'$ is a consistent Px86 execution. There exists a plain matching $f : |\mathcal{G}'| \twoheadrightarrow \bar{G}$.

To use the fact that I implements $\mathsf{Lin}(S)$, we construct an execution (without crashes) of I and a corresponding plain execution of $\mathsf{Lin}(S)$. Consider $\bar{\mathcal{G}}'$ de ned as follows.

1. First, detach all the $f^{-1}(m(\vec{v}, \bot))$ in $\mathcal{G}'$, giving the plain execution $\mathcal{G}'^d$;
2. Second, de ne $\mathcal{G}'^p \triangleq \mathcal{G}'^d \cap \downarrow_{\mathrm{po}} \lfloor \mathrm{P}^{\mathsf{Px86}} \rfloor$, where $\downarrow_{\mathrm{po}} \lfloor \mathrm{P}^{\mathsf{Px86}} \rfloor$ is the set of events of $\mathcal{G}'^d$ which are po-before a persisted Px86 event;
3. Remove all `finishOp` events, and relabel all p-writes and p-reads with the corresponding Px86 labels;
4. Finally, we get $\bar{\mathcal{G}}'$ by removing all crash events.

Similarly, we de ne $\bar{G}$ by removing all crash events and all events e of labeled with $m(\vec{v}, \bot)$ such that $f^{-1}(v) \cap \lfloor \mathrm{P}^{\mathsf{Px86}} \rfloor = \emptyset$ from $\mathcal{G}'$.

▷ Claim B.1. $\bar{G}$ is a plain execution of $\mathsf{Lin}(S)$, $\bar{\mathcal{G}}'$ is an execution of Px86 and $|\bar{\mathcal{G}}| \in \bar{G} \cdot I$, with $\bar{f} := f \cap \bar{E} : |\bar{\mathcal{G}}'| \twoheadrightarrow \bar{G}$.

Proof of claim. The rst two parts are obvious. The last part follows from the fact that, by construction, all $\bar{f}^{-1}(m(\vec{v}, \bot))$ are po-pre xes of the $f^{-1}(m(\vec{v}, \bot))$, and that I is pre x-closed. For $f^{-1}(m(\vec{v}, v))$ it follows from the fact that, if `finishOp` has nished executing, then all writes of the method have persisted. ◀

Write $\overline{\mathrm{rf}}'$ the restriction of rf' to $\bar{\mathcal{G}}'$.

▷ Claim B.2. The relation $\overline{\mathrm{rf}}$ de ned above is surjective on reads, and $\bar{\mathcal{G}}'$ is consistent.

Proof of claim. If a read event e is in $\bar{\mathcal{G}}'$, then in $\mathcal{G}'$:

$$e' : \mathtt{wr}_p(x, v) \xrightarrow{\mathrm{rf}} e : \mathtt{rd}_p(x, v) \xrightarrow{\mathrm{po}} e''$$

, with $e'' \in \lfloor \mathrm{P}^{\mathsf{Px86}} \rfloor$, and because of the speci cation of FliT e' is a write which is lin-before the read e, therefore e depends on e'' by transitivity and $e' \in \lfloor \mathrm{P}^{\mathsf{Px86}} \rfloor$, and thus $e' \in \bar{\mathcal{G}}'$.

Because FliT de nes SC executions, $\bar{\mathcal{G}}'$ is also consistent for Px86-consistency (namely TSO). ◀

We can use the fact that I is assumed to implement the library $\mathsf{Lin}(S)$ to obtain a correct execution $\bar{\mathcal{G}} \sqsubset \bar{G}$. That is, there is a total order lin which extends $\bar{\mathcal{G}}'.\mathrm{hb}$ and induces a sequence $r \in S$. De ne $e \in \mathcal{G} \iff e \in r$; and otherwise use the same rf, sw and so forth from $\bar{\mathcal{G}}$. ◀

```
1   method CAS(v_addr, expected, newval) {
2     p_addr = COMPUTE_P_ADDR(
            v_addr);
3     while (true) {
4       p_seq = p_addr->seq;
5       p_val = p_addr->val;
6       p_seq_again = p_addr->seq;
7
8       v_seq = v_addr->seq;
9       v_val = v_addr->val;
10      v_seq_again = v_addr->seq;
11
12      if (p_seq != p_seq_again ||
13          v_seq != v_seq_again)
14        continue;
15      if (p_seq == v_seq + 1) {
16        FLUSH(p_addr); FENCE();
17        DWCAS(v_addr, {v_val, v_seq},
18                      {p_val, p_seq});
19        continue;
20      }
21      if (p_seq != v_seq) continue;
22
23      if (p_val != expected) {
24        expected = p_val; return false;
25      }
26      before = {p_val, p_seq};
27      after = {vewval, p_seq+1};
28      res = DWCAS(p_addr, before, after);
29      FLUSH(p_addr); FENCE();
30      if (res) {
31        DWCAS(v_addr, before, after);
32      } else {
33        if (before.val == expected)
            continue;
34        DWCAS(v_addr, {v_val, v_seq},
            before);
35      }
36      return res;
37    }
38  }
39
40  method wr(v_addr, val) {
41    p_addr = COMPUTE_P_ADDR(
          v_addr);
42    while (!CAS(v_addr, *p_addr, val)) {}
43  }
44
45  method rd(v_addr) { return *v_addr; }
```

Figure 6 The Mirror implementation in Px86

C Correctness of Mirror

We consider the Mirror library [9], whose interface is that of persistent registers. Besides the constructor, `new`(), which allocates a new register, it provides two kinds of durable invocations: `wr`(ℓ, v), which writes the value v at location ℓ, and `cas`(ℓ, v_1, v_2), which atomically replaces the value at location ℓ by v_2 if it is equal to v_1 and does nothing otherwise; and one other non-durable invocation: `rd`(ℓ), which returns the value of the register at location ℓ.

As with FliT, the Mirror implementation avoids issuing a flush on reads, albeit using a different approach: it keeps two copies of the register contents: one in NVM and one in volatile memory. Reads access only the volatile copy, whereas writes and CASes first write the value to NVM, flush it, and then also write it to the volatile copy. To ensure lock-freedom, the implementation uses sequence numbers and a double-word (128 bits) compare-and-swap (DWCAS) to atomically update a pair of a value and sequence number (see Fig. 6), which is available on Intel-x86.

C.1 Specification of the library

Mirror provides sequentially consistent registers with the guarantee that completed writes persist, and that the persistence order agrees with the sequential order of the operations.

▶ **Definition C.1.** *An execution $\mathcal{G}$ of the library Mirror is correct if there exists an order* nvo *and a total order* lin *on $\mathcal{G}.E$ agreeing with $\mathcal{G}.$*po *and $\mathcal{G}.$*hb *($\mathcal{G}.\text{po} \cup \mathcal{G}.\text{hb} \subseteq \text{lin}$) such that:*

1. $\text{sw} = \bigcup_{\ell \in \mathbf{Loc}} ([W_\ell]; \text{lin}; [R_\ell]) \setminus (\text{lin}; [W_\ell]; \text{lin})$*, where W_ℓ is the set of all writes and CASes at location ℓ, and R_ℓ is the set of all reads and CASes at location ℓ;*
2. *if $(w, r) \in \text{sw}$, then $\text{loc}(w) = \text{loc}(r)$ and the value read by r is the value written by w;*
3. $[W]; (\text{po} \cup \text{sw})^+; [W] \subseteq \text{nvo}$*, where W is the set of all writes in $\mathcal{G}$;*

4. $dom(\mathrm{nvo}; [P^{\mathsf{Px86}}]) \subseteq \lfloor P^{\mathsf{Px86}} \rfloor$; *and*
5. $\lfloor P^{\mathsf{Px86}} \rfloor = \{w \in W \mid w \textit{ is a complete operation }\}$.

▶ **Theorem C.2.** *The implementation in Fig. 6 is a correct implementation of the* Mirror *library.*

Proof. Suppose given a Px86 execution $\mathcal{G}$ and a plain execution G' of Mirror such that $|\mathcal{G}| \twoheadrightarrow G'$ and such that $\mathcal{G}$ is consistent.

Operations are persisted. De ne the $\mathrm{P}^{\mathsf{Px86}}$-tagged events to be the events e in G' labeled with a `cas` operation such that, in $f^{-1}(e)$, the DWCAS at line 28 has succeeded and persisted. Clearly, because of the ush and the fence at line 29, all operations of the form $\mathtt{cas}(\ell, v_1, v_2, \mathsf{true})$ are tagged with $\mathrm{P}^{\mathsf{Px86}}$.

De ning the sw **relation**. Given an event e labeled with an operation $\mathtt{rd}(\ell, v)$ in G', we de ne the originating write of e, $u(e) \in G'$, as follows: consider r, the corresponding read in $\mathcal{G}$, and w the unique event such that $(w, r) \in \mathcal{G}.\mathrm{rf}$. If w corresponds to the CAS operations of line 31, then de ne $u(e) \triangleq f(w)$. Otherwise, let r' be the read of `p_addr` which precedes it (either line 5 or line 28 when the CAS is unsuccessful) and let w' be the write such that $(w', r') \in \mathcal{G}.\mathrm{rf}$ and de ne $u(e) \triangleq f(w')$.

Now, consider a CAS event $e \in G'$. If it returns false, de ne $u(e) \triangleq f(\mathrm{rf}^{-1}(r))$, where r is the read of `p_addr->val` in line 5. Otherwise, the DWCAS of line 28 succeeds, and we de ne $u(e) \triangleq f(\mathrm{rf}^{-1}(u))$, where u is the event corresponding to that DWCAS.

Finally, we de ne $\mathrm{sw}' \triangleq u^{-1}$, which is, by construction, the inverse of a function and is surjective on reads and CASes. Because the value written in `v_addr->val` is the same as the value read from the source of the rf edge, the value which is read matches the one which is written.

De ning the lin **order**.

We de ne lin by projecting along f a linearization of tso of the linearization points of the operation implementations: for `cas`, it is the DWCAS of lines 28 if control- ow reaches them, and otherwise the read of `p_addr->val` in line 5. Clearly it agrees with po'.

It remains to prove that the relation sw' de ned above reads the most recent write, and that it agrees with the $\mathrm{nvo}' = f_*^{\forall}(\mathrm{nvo})$ order, where $f_*^{\forall}(()r) = \{(x, y) \mid \forall u \in f^{-1}(x),\ \forall v \in f^{-1}(y),\ (u, v) \in r\}$. The rst conjunct follows from the fact that all values are written using a Px86 DWCAS, and that all writes that are in P_i have their corresponding writes to `p_addr` persisted. The second conjunct follows from the fact that the linearization points of all persistent operations are writes to the same location, and thus their tso order agrees with nvo.

The relation $[W]; (\mathrm{po}' \cup \mathrm{sw}')^+; [W]$ **is included in** nvo'.

We rst note that given $(r, w) \in \mathrm{po}' \cap (R \times W)$, the reads in $f^{-1}(r)$ are tso-before the writes in $f^{-1}(w)$. Also, given $(w, r) \in \mathrm{sw}' \setminus \mathrm{po}$, the read r' of `v_addr` in $f^{-1}(r)$ is tso-before $\mathrm{rf}^{-1}(r')$ since they belong to di erent threads. In all paths, r is po-after a ush and a fence which is tso-after the write to `p_addr`.

Now, let $(w, w') \in [W]; (\mathrm{po}' \cup \mathrm{sw}')^*; [W]$, there exists a sequence of events which are related by the two relations we just discussed. Therefore the write to `p_addr` in $f^{-1}(w')$ is tso-after a fence, which is tso-after a ush of `p_addr` which is tso-after the persistent write in $f^{-1}(w)$. As such, $(w, w') \in \mathrm{nvo}' = f_*^{\forall}(\mathrm{nvo})$. ◀

C.2 Using Mirror to Enforce Durable Linearizability

As with FliT, Mirror can be used to transform a linearizable data-structure into a durably linearizable one. Given an implementation I over Px86 using reads, writes and CASes, let

$m(I)$ be the implementation over Mirror which replaces the Px86 calls with their corresponding Mirror calls.

▶ **Theorem C.3.** *If* $\mathit{Px86} \vdash I : \mathit{Lin}(S)$*, then* $\mathit{Mirror} \vdash m(I) : \mathit{DurLin}(S)$*.*

The proof is similar to the corresponding theorem of FliT, noting that Mirror provides a stronger speci cation than FliT.

D Proof of Transaction library

D.1 Proof of $\mathsf{L}_{\mathsf{trans}}$

We use a module Q which provides a durably linearizable queue. One simple solution to implement this module is to take any linearizable queue (e.g. the one proved in [24]) and use the results of 4 to obtain a durably linearizable queue. We prove the implementation I_{trans}, presented in Figure 1 of the transaction library.

▶ **Theorem D.1.** I_{trans} *implements the library speci cation* $\mathsf{L}_{\mathit{trans}}$*. Formally,* $\mathsf{L}_{\mathit{Px86}}, \mathsf{L}_{\mathsf{Queue}} \vdash I_{\mathsf{trans}} : \mathsf{L}_{\mathit{trans}}$*.*

Proof sketch. The idea is to de ne nvo on the events of $\mathsf{L}_{\mathsf{trans}}$ to be hb induced by the implementation-level execution graph. By well-formedness, we know all events are in a critical section, and we declare that an event has tag P^{tr} if the corresponding appending of COMMITTED has persisted.

In the absence of a crash, condition (7) follows from the fact that, since beginnings and ends of critical sections are externally synchronized, all reads and writes of PT registers are related by hb. In case of a crash, if a read r reads from a write w with a crash in-between, we know by de nition that $w \in \lfloor \mathrm{P}^{\mathsf{tr}} \rfloor$, and, according the well-formedness condition of $\mathsf{L}_{\mathsf{trans}}$, that a call to recovery is hb-between the crash and r. Since $w \in \lfloor \mathrm{P}^{\mathsf{tr}} \rfloor$, the COMMITTED message has been written to the log and thus the write has not been undone by the recovery. Symmetrically, we also know that all later writes to the register have not been persisted, and thus the recovery procedures has undone these writes.

Condition (8) holds for a similar reason: a read from another section is hb-after the end of the section that wrote into the register. Conditions (11) and (10) which are treated later imply that the write has persisted.

Global correctness. Since this library introduces it own tags, we can assume that the operations tagged with its own tags are its own operations.

Condition (10) by de nition of nvo above.

Condition (11): a write is persisted if the COMMITTED message of the corresponding critical section has been persisted. Therefore either all writes or none of the writes of a section persist.

Condition (12): If `PTEnd` has nished, the sfence instruction has nished and the COMMITTED message has persisted. ◀

D.2 Proof of $\mathsf{L}_{\mathsf{Strans}}$

We consider a (volatile) lock library with the following speci cation: An execution $\mathcal{G}$ is correct if the events are totally ordered by hb in such a way that the induced word is of the form $(\texttt{lock} \cdot \texttt{unlock})^* \cdot \texttt{lock}^?$.

▶ **Theorem D.2.** *The implementation* I_{LPT} *of* $\mathsf{L}_{\mathit{Strans}}$ *is correct:* $\mathsf{L}_{\mathsf{Lock}}, \mathsf{L}_{\mathit{trans}} \vdash I_{\mathsf{LPT}} : \mathsf{L}_{\mathit{Strans}}$*.*

Proof sketch. The salient part of the proof is establishing that the library TC is used according to its well-formedness specification. According to Definition 3.10, we consider an implementation execution $\mathcal{G}$ which has an immediate prefix $\bar{\mathcal{G}} \hookrightarrow_{im} \mathcal{G}$ which is correct. The important case is when the event added in $\mathcal{G}$ compared to $\bar{\mathcal{G}}$ is a call to `PTBegin` or `PTEnd` which is part of the implementation of a call to `LPTBegin` or `LPTEnd` respectively. In that case, the correctness of $\bar{\mathcal{G}}$ restricted to the Lock library implies that two LPT critical sections are related by hb. Therefore, the calls to the PT library are externally synchronized.

The rest of the proof consists in using unchanged the corresponding properties of the PT library: We consider a correct execution $\mathcal{G} \in \mathbf{Exec}(\mathsf{L}_{\mathsf{Lock}}, \mathsf{L}_{\mathsf{trans}}, \star_{\mathsf{L}_{\mathsf{trans}}})$ and a corresponding locally correct execution $\mathcal{G} \in \mathbf{Exec}(\mathsf{L}_{\mathsf{Strans}}, \mathsf{L}_{\mathsf{trans}}, \star_{\mathsf{L}_{\mathsf{trans}}})$ and we need to prove that $\mathcal{G}$ is correct with respect to the global specification of $\mathsf{L}_{\mathsf{trans}}$. This is easy to see that this follows from the correctness of $\mathcal{G}$ with respect to the same global specification. ◀

D.3 Counter

method NewCounter() = PTAlloc()

method CounterInc(c) = let v = Read(c) in PTWrite(v+1)

method CounterRead(c) = PTRead(c)

An execution is $\mathsf{L}_{\mathsf{cntr}}$-correct if each call to CounterRead(c) returns the number of CounterInc(c) which are hb-before it. The proof of correctness is simple, for example by using condition (7) of the specification of $\mathsf{L}_{\mathsf{trans}}$.

Note that in the proof, the implementation graph $\mathcal{G}$ we consider is *not* well-formed, since there are no begin/end calls! However, we know that for any well-formed context in which $\mathsf{L}_{\mathsf{cntr}}$ is used, the library $\mathsf{L}_{\mathsf{trans}}$ will also be used according to well-formedness, and as such we can use the fact that $\mathcal{G}$ is correct.