Specifying and Verifying Persistent Libraries

Léo Stefanesco[1](✉), Azalea Raad[2], and Viktor Vafeiadis[1]

[1] MPI-SWS, Kaiserslautern, Germany
leo.stefanesco@mpi-sws.org
[2] Imperial College, London, United Kingdom

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 specifications ranging from hardware architectural specifications 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 specific to persistent memory compared to weak memory and how they are handled by our framework.

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 benefit 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 *flushes* 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

© The Author(s) 2024
S. Weirich (Ed.): ESOP 2024, LNCS 14577, pp. 185–211, 2024.
https://doi.org/10.1007/978-3-031-57267-8_8

thought of as the analogue of concurrent libraries for persistency. And just as concurrent libraries require a specification, so do persistent libraries.

The question naturally arises: what is the right specification for persistent libraries? Prior work has suggested a number of candidate definitions, such as *durable linearizability*, *buffered durable linearizability* [17], and *strict linearizability* [1], which are all extensions of the well-known correctness condition for concurrent data structures (i.e. linearizability [15]). In general, these definitions stipulate the existence of a total order among all executed library operations, a contiguous prefix of which is persisted upon a crash: the various definitions differ in exactly what this prefix should be, e.g. whether it is further constrained to include all fully executed operations.

Even though these specifications have a nice compositionality property, we argue that none of them are *the* right specification pattern for *every* persistent concurrent library. While for high-level persistent data structures, such as stacks and queues, a strong specification 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 flushes. Their semantics, recently formalized in [30,19,28] in the case of the Intel-x86 architecture and in [31,5] in the case of the ARMv8 architecture, quite clearly do not fit into the framework of the durable linearizability definitions. More generally, there are useful concurrent libraries (especially in the context of weak memory consistency) that are not linearizable [26]; it is, therefore, conceivable that making those libraries persistent will require weak specifications.

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 [6,24] provide a global persistency barrier that affects 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 flexible way of specifying persistent libraries, allowing each library to have a very different specification—be it durable linearizability or a more complex specification in the style of the hardware architecture persistency models. Further, to handle libraries that have a global effect (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 specifications of its constituent libraries without referring to their implementations (vertical compositionality). To achieve the latter, we define 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 first version of our framework over the classical notion of an execution *history* [15], 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) [23].

Subsequently, in §3 we generalise and extend our framework to handle weak consistency models such as x86-TSO [32] and RC11 [22], thereby allowing us to represent hardware persistency models such as Px86 [30] and PARMv8 [31], in our framework. To do so, we rebase our formal development over execution graphs using Yacovet [26] 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 Px86 [30], to general-purpose correctness conditions such as durable linearizability [17]. 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 first formal specifications of the FliT [35] and Mirror [10] libraries and establish the correctness of not only their implementations against their respective specifications, 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 [26], and leverages our 'semantic' approach in §3. Moreover, our proofs of these constructions are the *first* 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' specification (in addition to its consistency and persistency specifications) 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 first formal specifications of the FliT and Mirror libraries, verify their implementations against their specifications 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 specification. 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. [15], 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 infinite domains: **Meth** of method names, **Loc** of memory locations, **Tid** of thread identifiers, 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 identifiers, and v over values. An optional value $v_\bot \in \mathbf{Val}_\bot$ is either a value $v \in \mathbf{Val}$ or $\bot \notin \mathbf{Val}$.

2.1 Library Interfaces

A *library interface* declares a set of method invocations of the form $m(\boldsymbol{v})$. 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.

Definition 1. *A* library interface *L is a tuple $\langle \mathcal{M}, \mathcal{M}_\mathrm{c}, \mathrm{loc} \rangle$, where the set of method invocations $\mathcal{M}$ is a subset of $\mathcal{P}(\mathbf{Meth} \times \mathbf{Val}^*)$, $\mathcal{M}_\mathrm{c} \subseteq \mathcal{M}$ is the set of constructors, and* $\mathrm{loc} : \mathcal{M} \times \mathbf{Val}_\bot \to \mathcal{P}(\mathbf{Loc})$ *is the location function.*

Example 1 (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 define $\mathrm{loc}(\mathsf{QueueNew}(), x) = \mathrm{loc}(\mathsf{QueueEnq}(x, \_), \_) = \mathrm{loc}(\mathsf{QueueDeq}(x), \_) = \{x\}$.

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 $m(\boldsymbol{v})_{\mathsf{t}}$ with $m(\boldsymbol{v}) \in \bigcup_{L \in \Lambda} L.\mathcal{M}$ 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.

Definition 2 (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.*

Definition 3 (Histories). *A Λ-*history *is a finite 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 $m(\boldsymbol{v})_{\mathsf{t}}$ in H, its *matching return* (when it exists) is the first event of the form $\mathbf{ret}(v)_{\mathsf{t}}$ that succeeds it (they share the same thread). A *call* is a pair $m(\boldsymbol{v})_{\mathsf{t}}{:}v_{\bot}$ of an invocation and either its matching return $v_{\bot} \in \mathbf{Val}$ (*complete call*) or $v_{\bot} = \bot$ (*incomplete call*).

A *library* (specification) 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.

Definition 4. *A* library specification *(or simply a* library*) L is a tuple $\langle L, \mathcal{S}_{\mathsf{c}} \rangle$, where L is a library interface, and $\mathcal{S}_{\mathsf{c}} \subseteq \mathbf{Hist}(L)$ denotes its set of* consistent *histories.*

2.3 Linearizability

Linearizability [15] is a standard way of specifying concurrent libraries that have a sequential specification S, denoting a set of finite sequences of complete calls. Given a sequential specification 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 sufficient to consider consistent executions because inconsistent executions are, by definition, guaranteed by the library to never happen. Happens-before is defined as follows.

Definition 5 (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 specification 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 effect. 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 define linearizability as follows [14].

Definition 6. *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 specification, S_{Queue}.

Example 2 (Sequential queue specification). 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 define 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 first 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 effect on the consistency of the history after the crash. We modify the definition 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 specification is *durable linearizability* [17], defined as follows.

Definition 7. *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 effect 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 $\mathsf{Px86}$ (the Intel-x86 persistency model) [30].

A more relaxed and realistic specification would consider two linearizations of the events: the standard *volatile* order $\prec$ 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) $\mathrm{mo}_x \subseteq\prec$ and $\mathrm{mo}_x \subseteq \mathrm{nvo}$. Given a history H, we can then combine these two linearizations by using $\prec$ 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 $\prec_i$ for $\prec \cap (H_i \times H_i)$ 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. $\prec_k$ 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 3. 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. $\prec$ 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 $\prec$ ($W_{\mathsf{t}_1}(x,1)$ 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 $\mathsf{Px86}$, 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 $\mathsf{L_{wreg}}$ as follows: given history H, write call c_w and PFENCE event c_f, if $c_w \prec_H c_f$, then $(c_w, c_f) \in$ nvo.

Example 4. Consider the history obtained from example 3 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 $\mathsf{L_{wreg}}$: 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 $\mathsf{L_{trans}}$ 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 $\mathsf{L_{trans}}$ methods are: $\mathsf{PTNewReg}$ to allocate a register that can be accessed (read/written) within a transaction; $\mathsf{PTBegin}$ and PTEnd to start and end a transaction, respectively; $\mathsf{PTRead(x)}$ and $\mathsf{PTWrite(x,\ v)}$ to read from and write to $\mathsf{L_{trans}}$ register x, respectively; and $\mathsf{PTRecover}$ 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 [30,31] 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 $\textsc{t}$ 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 $\textsc{p}^{\mathsf{tr}}$, denoting an operation that has abstractly persisted. The specification of $\mathsf{L}_{\mathsf{trans}}$ then guarantees that all operations tagged with $\textsc{t}$ inside a transaction persist atomically, in that either they are all tagged with $\textsc{p}^{\mathsf{tr}}$ of none of them are. Dually, using the well-formedness condition, $\mathsf{L}_{\mathsf{trans}}$ requires that all operations tagged with $\textsc{t}$ appear inside a transaction. Note that as the persistent queue and set libraries tag their operations with $\textsc{t}$, verifying their implementations incurs related proof obligations; we will revisit this later when we formalize the notion of library implementations.

Remark 1 (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 [19] 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 $\textsc{t}$ and $\textsc{p}^{\mathsf{tr}}$.

Definition 8 (Interfaces). *An* interface *is a tuple* $L = \langle \mathcal{M}, \mathcal{M}_{\mathrm{c}}, \mathrm{loc}, \textsc{Tags}_{\mathrm{new}}, \textsc{Tags}_{\mathrm{dep}} \rangle$*, where* $\mathcal{M}$*,* $\mathcal{M}_{\mathrm{c}}$*, and* loc *are as in Def. 1,* $\textsc{Tags}_{\mathrm{new}}$ *is the set of tags* L *introduces, and* $\textsc{Tags}_{\mathrm{dep}}$ *is the set of tags* L *uses. The set of tags usable by* L *is* $\textsc{Tags}(L) \triangleq L.\textsc{Tags}_{\mathrm{new}} \cup L.\textsc{Tags}_{\mathrm{dep}}$*.*

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 9. *Given a library interface* L*, a* tagged method invocation *is of the form* $m(\boldsymbol{v})^{T}_{\mathsf{t}} : v_{\perp}$*, 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 10. *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 $m(\boldsymbol{v})^{T}_{\mathsf{t}} : v_{\perp}$ 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 11. *A* library specification L *is a tuple* $\langle L, \Lambda_{\mathrm{tags}}, \mathcal{S}_{\mathsf{c}}, \mathcal{S}_{\mathsf{wf}}, \mathcal{T}_{\mathsf{c}}, \mathcal{T}_{\mathsf{wf}} \rangle$*, where* L*,* $\mathcal{S}_{\mathsf{c}}$ *and* $\mathcal{S}_{\mathsf{wf}}$ *are as in Def. 4;* $\mathcal{T}_{\mathsf{c}}$ *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:* $L.\textsc{Tags}_{\mathrm{dep}} \subseteq \bigcup_{\mathsf{L}' \in \Lambda_{\mathrm{tags}}} \mathsf{L}'.\textsc{Tags}_{\mathrm{new}}$*. Both* $\mathcal{T}_{\mathsf{wf}}$ *and* $\mathcal{T}_{\mathsf{c}}$ *contain the empty history.*

In the context of a history, we write $\lfloor \textsc{t} \rfloor$ for the set of events or calls tagged with the tag $\textsc{t}$ (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 $\textsc{t}$-tagged events is of the form described by the regular expression $(\mathsf{PTBegin}.\lfloor\textsc{t}\rfloor^*.\mathsf{PTEnd})^*$. In particular, transaction nesting is disallowed in our simple $\mathsf{L}_{\mathsf{trans}}$ library.

To define global consistency, we need to know when two operations are part of the same transaction. Given a history H, we define the *same-transaction* relation, strans, relating pairs of $e, e' \in \lfloor\textsc{t}\rfloor \cup \mathsf{PTEnd} \cup \mathsf{PTBegin}$ 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 $\forall(e, e') \in \text{strans}, e \in \lfloor\textsc{p}^{\mathsf{tr}}\rfloor \Leftrightarrow e' \in \lfloor\textsc{p}^{\mathsf{tr}}\rfloor$, and all completed PTEnd calls are in $\lfloor\textsc{p}^{\mathsf{tr}}\rfloor$. Since the PTEnd call is related to all events inside its transaction, this specification does express that (1) a transaction persist by the time the call to PTEnd finishes and (2) all events persist *atomically.*

Finally, we need to define 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 define the concrete meaning of 'having persisted' for these registers. Let S be the sequential specification of a register. Let $H \in \mathbf{Hist}(\mathsf{L}_{\mathsf{trans}})$ be a history decomposed into k eras as $H_1 \cdot \lightning \cdot H_2 \cdot \lightning \cdot \cdots \cdot \lightning \cdot H_k$. Then $H \in \mathsf{L}_{\mathsf{trans}}.\mathcal{S}_{\mathsf{c}}$ iff all events are tagged with $\textsc{t}$, and there exists a $\prec$-linearization H_ℓ of $\big((H_1 \cdot \lightning \cdot H_2 \cdot \lightning \cdot \cdots \cdot \lightning \cdot H_{k-1}) \cap \lfloor\textsc{p}^{\mathsf{tr}}\rfloor\big) \cdot H_k$ such that $H_\ell \in S$, where $\lfloor\textsc{p}^{\mathsf{tr}}\rfloor$ is the set of events of H tagged with $\textsc{p}^{\mathsf{tr}}$. In other words, a write operation is seen after a crash iff 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 specifications.

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 [13,26], the base layer, which represents the primitives of the hardware, is specified 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 refinement* and is impractical to prove directly. We define a stronger notion that is *compositional* and more practical to use.

Definition 12 (Implementation). *Given a collection* Λ *of libraries and a library* L*, an* implementation I of L over Λ *is a map,* $I \quad : \quad \mathsf{L}.\mathcal{M} \times \mathbf{Val}_\bot \quad \longrightarrow$

```
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); }
```

Fig. 1. Implementation of $\mathsf{L}_{\mathsf{trans}}$

$\mathcal{P}(\mathbf{Hist}(\Lambda))$*, such that it is downward-closed: 1) if* $H \in I(m(\boldsymbol{v})_{\mathsf{t}}, v_\perp)$ *and* H' *is a prefix of* H*, then* $H' \in I(m(\boldsymbol{v}), \perp)$*; and 2) each* $I(m(\boldsymbol{v})_{\mathsf{t}}{:}v_\perp)$ *history only contain events by thread* t*.*

Intuitively, $I(m(\boldsymbol{v}), v_\perp)$ contains the histories corresponding to a call $m(\boldsymbol{v})$ 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 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 appendix [34].

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 $m'(\boldsymbol{v})$ in H' with some operation $f(m'(\boldsymbol{v}))$ in H in such a way that the collection $f^{-1}(m(\boldsymbol{v})_{\mathsf{t}}{:}v_\perp)$ of operations corresponding to some call $m(\boldsymbol{v})_{\mathsf{t}}{:}v_\perp$ in H agrees with $I(m(\boldsymbol{v})_{\mathsf{t}}{:}v_\perp)$.

Definition 13. *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* $f : \{1, \dots, |H'|\} \to \{1, \dots, |H|\}$*,* H' (I, f)-matches H *if the following hold:*

1. f *is surjective;*

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

The action *of I on a history H is defined as follows:*

$$H \cdot I \triangleq \{H' \mid \exists f.\ H'\ (I, f)\text{-}matches\ H\}.$$

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 $m(\boldsymbol{v})$ 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 $\textbf{Hist}(\mathsf{L})$, 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 [26]). 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 14 (Consistency). *History $H \in \textbf{Hist}(\Lambda)$ is* consistent *if for all $\mathsf{L} \in \Lambda$, $H[\mathsf{L}] \in \mathsf{L}.\mathcal{S}_{\mathsf{c}}$ and $\pi_{\mathsf{L}}(H) \in \mathsf{L}.\mathcal{T}_{\mathsf{c}}$. 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 15 (Encapsulation). *A history $H \in \textbf{Hist}(\Lambda)$ 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 16. *History $H \in \mathbf{Hist}(\Lambda)$ is* immediately well-formed *if the following hold:*

1. *H is encapsulated;*
2. *$H[\mathsf{L}] \in \mathsf{L}.\mathcal{S}_{\mathit{wf}}$, for all $\mathsf{L} \in \Lambda$; and*
3. *$\pi_{\mathsf{L}}(H) \in \mathsf{L}.\mathcal{T}_{\mathit{wf}}$ 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 17 (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 simplified 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. 17. Specifically, while Def. 17 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 $\mathsf{L}.\mathcal{T}_{\mathsf{c}}$.

Definition 18 (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 $H \in \mathsf{L}.\mathcal{S}_{\mathsf{c}} \cap \mathsf{L}.\mathcal{T}_{\mathsf{c}}$; 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 $\pi_{\mathsf{L}'}(H') \in \mathsf{L}'.\mathcal{T}_{\mathsf{wf}} \cap \mathsf{L}'.\mathcal{T}_{\mathsf{c}}$, then $\pi_{\mathsf{L}'}(H) \in \mathsf{L}'.\mathcal{T}_{\mathsf{c}}$.*

The preservation of well-formedness (condition 1) does not change compared to its counterpart in Def. 17, 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 specifications 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 sufficient to show an implementation I is compositionally correct as it implies that I is correct.

Theorem 1 (Correctness). *If an implementation I of* L *over Λ is* compositionally correct *(Def. 18), then it is also* correct *(Def. 17).*

Example 5 (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 first 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 specified 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 $\textsc{t}$ 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 6 (A Client of $\mathsf{L}_{\mathsf{trans}}$). To see how the global consistency specifications 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

```
method mmNew() :=                          method mmMin(x) :=
 (PTNewReg(), PTNewReg())                    PTRead(x.1)

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

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

$\mathsf{L}_{\mathsf{trans}}$ transactions, and provides four methods: $\mathtt{mmNew}()$ to construct a min-max counter, $\mathtt{mmAdd}(x, n)$, to add integer n to the min-max counter, and $\mathtt{mmMin}(x)$ and $\mathtt{mmMax}(x)$ 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 $\mathtt{mmAdd}$ 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 $\mathtt{mmNew}$ 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 defined using the obvious sequential specification S_{mmcnt}, which states that calls to $\mathtt{mmMin}$ return the minimum of all integers previously given to $\mathtt{mmAdd}$ 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 specification 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 Figure 2 satisfies $\mathsf{L}_{\mathsf{mmcnt}}$ 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 [35] and Mirror [10]. Thanks to our framework, we formalise the statement above for the first time and prove it for both Flit and Mirror against a realistic consistency (concurrency) model (see §4).

3 Generalization to weak-memory

This section sketches how we generalize the framework presented in the previous section to the weak memory, where events generated by the program are not totally ordered. For lack of space, the technical details, which largely follow that of the previous section, are relegated to the Appendix [34]. The purpose of this section is to give an idea of how *executions*, a standard tool in the semantics of weak memory, generalize the *histories* we used in the Overview section, and to give enough context for the case studies that follow.

Unlike the histories that we discussed in the previous section, in which events are totally ordered by a notion of time, events in executions are only partially ordered, reflecting that instructions executed in parallel are not naturally ordered. Formally, an execution is thus a set of events equipped with a partial order which represents the ordering between events from the same thread. This partial order, written po, for program-order, is depicted with black arrows in Fig. 3, where it orders minimally the initial event, and the two events of each thread according to the source code. Additional edges indicate, for each read-event returning the value v, the write-event that provided the value v: in that case, an rf-edge from the write-event to the read-event is added to the execution.

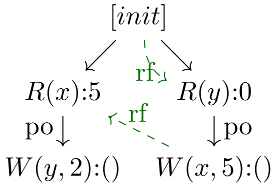

Fig. 3. An execution of the program P: $a = x;\ y = 2 \parallel a = y;\ x = 5$

To be able to reason about synchronization, the notion of happens-before needs to be adapted to this setting. It is defined using po and an additional type of edge, *synchronizes-with*, written sw, which denotes that two events synchronize with each other, and in particular that one happens before the other. Usually, sw $\subseteq$ rf, for example between a release-write and an acquire-read in the C11 memory model. Given these sw edges, the happens-before order they induce, which generalizes $\prec$ from the previous section is defined as the transitive closure $(\mathrm{po} \cup \mathrm{sw})^+$. This is not sufficient however, because we consider *partial* executions G where the focus is on a subset of the libraries in some unknown global execution G', that is: $G = G' \downharpoonright \mathsf{L}$. Therefore, external events (in G' but not in G) may induce happens-before relations between events of G, yet we want to specify library L without referring to any such execution G' that contains it. To solve this issue, we use the technique of [26], and we add a final type of edge to executions: hb, which corresponds to both the external and the internal synchronization. Because of the latter, it must contain the internal synchronization: $\mathrm{po} \cup \mathrm{sw} \subseteq \mathrm{hb}$.

To summarize, an execution is a tuple $\langle E, \mathrm{po}, \mathrm{rf}, \mathrm{sw}, \mathrm{hb} \rangle$ comprised of a set E of events, and of the relations we just described. A library specification is the same as in the previous section, *mutatis mutandis*. The sets of executions that are parts of specifications are defined using a formalism developed in the weak memory model literature. A set $\mathcal{S}$ of executions is described with conditions about relations built from po, rf, etc. Given a set V of events, we denote by $[V]$ the relation $V \times V$, and we denote by $R_1; R_2$ the standard composition of two relations R_1 and R_2. For example, if R denotes the set of read-events of an execution and W the set of write-events, the condition $[W]; \mathrm{rf}; [R] \subseteq \mathrm{sw}$ states that if there is a rf-edges between two events $e_1 \in W$ and $e_2 \in R$ of an execution, there must also be a sw synchronization edge between e_1 and e_2.

As in the previous section, the tag system allows the library specification to state which events must have been persisted in a valid execution. The semantics of a program is a set of executions that contain events from all the libraries used by the program; and whose happens-before order satisfy $\mathrm{hb} = (\mathrm{po} \cup \mathrm{sw})^+$, as there are no external synchronization in the executions of the whole program. The Appendix [34] details how our framework is defined in this more general setting.

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. Specifically, we consider two such libraries FliT [35] and Mirror [10]; we specify them both in our framework, prove their implementations sound against their respective specifications, and further prove their general result for converting data structures.

4.1 The FliT Library

FliT [35] 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. Specifically, a FliT object ℓ can be accessed via

```
method $\texttt{wr}_\pi(\ell, v)$ :
  if $\pi = \mathsf{p}$ then
    fetch-and-add($\mathit{flit\text{-}counter}(\ell)$, 1);
    write($\ell, v$);
    flush$_{\mathsf{opt}}$($\ell$);
    fetch-and-add($\mathit{flit\text{-}counter}(\ell)$, −1);
  else
    sfence;
    write($\ell, v$);
```

```
method $\texttt{rd}_\pi(\ell)$ :
  local $v$ = read($\ell$);
  if $\pi = \mathsf{p} \wedge \mathit{flit\text{-}counter}(\ell) > 0$ then
    flush$_{\mathsf{opt}}$($\ell$);
  return $v$;

method finishOp :
  sfence;
```

Fig. 4. FliT library implementation in Px86

write and read methods, $\texttt{wr}_\pi(\ell, v)$ and $\texttt{rd}_\pi(\ell)$, 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 specifies if the write (resp. read) is *persistent* ($\pi = \mathsf{p}$) in that its effects must be persisted, or *volatile* ($\pi = \mathsf{v}$) in that its persistency has been optimised and offers weaker guarantees. The default access type is persistent (p), and the volatile accesses may be used as optimizations when weaker guarantees suffice. Wei et al. [35] introduce a notion of *dependency* between different 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 specification (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:

- $\texttt{rd}_\pi(\ell)$ with $\pi \in \{\mathsf{p}, \mathsf{v}\}$, for a π-read from ℓ;
- $\texttt{wr}_\pi(\ell, v)$ 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 Specification. We develop a formal specification 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 definitions that use a rf relation (as in the Px86 specification) 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.

Definition 19 (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:*
   $dom([W^{\mathsf{p}}]; (\mathrm{po} \cup \bigcup_{\ell \in \mathbf{Loc}}[W^{\mathsf{p}}_\ell]; \mathrm{lin}; [R^{\mathsf{p}}_\ell])^+; [\mathit{finishOp}]) \subseteq \lfloor \mathrm{P}^{\mathit{Px86}} \rfloor$
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. 4. Whereas a naive implementation of this interface would have to issue a flush instruction both after persistent writes and in persistent reads, the implementation shown associates each location with a counter to avoid performing superfluous flushes when reading from a location whose value has already persisted. Specifically, a persistent write on ℓ increments its counter before writing to and flushing it, and decrements the counter afterwards. As such, persistent reads only need to issue a flush if the counter is positive (i.e. if there is a concurrent write that has not executed its flush yet).

Theorem 2. *The implementation of FliT in Fig. 4 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*[3]. We assume that all method implementations are single-threaded, i.e. all plain executions $I(m(\boldsymbol{v}))$ are totally ordered.

Theorem 3. *If* $\mathit{Px86} \vDash I : \mathit{Lin}(S)$*, then* $\mathit{FliT} \vDash p(I) : \mathit{DurLin}(S)$*.*

4.2 The Mirror Library

The Mirror [10] persistent library has similar goals to FliT. The main difference between the two is that Mirror operations do not offer two variants, and their operations are implemented differently from those of FliT. Specifically, in Mirror each location has two copies: one in persistent memory to ensure durability,

[3] The definition here is the same as in §2, as hb-linearizations of the execution still yield sequential executions.

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 specification and implementation in the technical appendix where we also prove that its implementation is correct against its specification. 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 specification 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) modified 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[4] Q, and assumes that 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 modifications to the structure of the specification: the specification in §2 requires that any 'transaction-aware operation' (i.e. those tagged with $\textsc{t}$) 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 $\textsc{t}$ to appear between calls to operations that behave like `PTBegin` and `PTEnd`. To that end, we add two new tags $\textsc{b}$ and $\textsc{e}$ to denote such operations, respectively.

5.1 Specification

The $\mathsf{L}_{\mathsf{trans}}$ library provides four *tags*: 1) $\textsc{t}$ for transaction-aware 'client' operations; 2) $\textsc{p}^{\mathsf{tr}}$ for operations that have persisted using transactions; and 3) $\textsc{b}, \textsc{e}$ 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. $\lfloor \textsc{t} \rfloor$ for the set of events tagged with $\textsc{t}$. Note that while $\mathcal{B}$ denotes the set of the begin events in library $\mathsf{L}_{\mathsf{trans}}$, the $\lfloor \textsc{b} \rfloor$ denotes the set of all events that are tagged with $\textsc{b}$, 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 $\textsc{b}$; similarly for $\mathcal{E}$ and $\lfloor \textsc{e} \rfloor$. As such, our local specifications below (i.e. local well-formedness

[4] For example, take any linearizable queue implementation and use the FliT library as described in §4.

and consistency) are defined in terms of $\mathcal{B}$ and $\mathcal{E}$, whereas our global specifications are defined in terms of $\lfloor \textsc{b} \rfloor$ and $\lfloor \textsc{e} \rfloor$. As before, for brevity we write e.g. $[\textsc{t}]$ as a shorthand for the relation $[\lfloor \textsc{t} \rfloor]$. We next define the 'same-transaction' relation strans:

$$\mathrm{strans} \triangleq [\lfloor \textsc{b}\rfloor \cup \lfloor \textsc{e}\rfloor \cup \lfloor \textsc{t}\rfloor]; (\mathrm{po} \cup \mathrm{po}^{-1}); [\lfloor \textsc{b}\rfloor \cup \lfloor \textsc{e}\rfloor \cup \lfloor \textsc{t}\rfloor] \setminus ((\mathrm{po}; [\textsc{e}]; \mathrm{po}) \cup (\mathrm{po}; [\textsc{b}]; \mathrm{po}))$$

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

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

An execution is globally well-formed if client operations are inside transactions:

6. $\lfloor \textsc{t} \rfloor \subseteq \mathit{rng}([\textsc{b}]; \mathrm{po})$
7. $[\textsc{e}]; \mathrm{po}; [\textsc{t}] \subseteq [\textsc{e}]; \mathrm{po}; [\textsc{b}]; \mathrm{po}; [\textsc{t}]$

An execution is locally-consistent if there exists a relation rf satisfying:

8. rf relates writes to reads, $\mathrm{rf} \subseteq \mathcal{W} \times \mathcal{R}$, 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: $\mathrm{rf}^{-1}; \mathrm{hb} \subseteq \mathrm{hb}$
10. External reads (reading from a different transaction) read from persisted writes: $\mathit{dom}(\mathrm{rf} \setminus \mathrm{strans}) \subseteq \lfloor \textsc{p}^{\mathsf{tr}} \rfloor$

An execution is globally-consistent if there exists an order nvo over $\lfloor \textsc{t} \rfloor$ satisfying:

11. Transactions are nvo-ordered: $[\textsc{e}]; \mathrm{hb}; [\textsc{b}] \subseteq \mathrm{nvo}$
12. nvo is the persistance order: $\mathit{dom}(\mathrm{nvo}; [\textsc{p}^{\mathsf{tr}}]) \subseteq \lfloor \textsc{p}^{\mathsf{tr}} \rfloor$;
13. Either all the events or none of the events in a transaction persist (atomicity): $[\textsc{p}^{\mathsf{tr}}]; \mathrm{strans}; [\textsc{t}] \subseteq [\textsc{p}^{\mathsf{tr}}]$
14. All events of a completed transaction (ones with an associated end event) persist: $\lfloor \textsc{e} \rfloor^{c} \subseteq \lfloor \textsc{p}^{\mathsf{tr}} \rfloor$, where $\lfloor \textsc{e} \rfloor^{c}$ denotes the set of method calls tagged with $\textsc{e}$ which have completed.

Theorem 4. *The $\mathsf{L}_{\mathit{trans}}$ implementation in Fig. 1 over $\mathsf{\mathit{Px86}}$ 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 specifications. 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. *The implementation of* $\mathsf{L}_{\mathsf{Strans}}$ *above is correct.*

Using compositional correctness, the main proof obligation is the condition 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 prefix that is consistent. 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: If a client uses $\mathsf{L}_{\mathsf{cntr}}$ in concurrent settings, it must call its methods within critical sections. The $\mathsf{L}_{\mathsf{cntr}}$ provides three operations to create (NewCounter), increment (CounterInc) and read a counter (CounterRead). The specification and implementation of $\mathsf{L}_{\mathsf{cntr}}$ are given in [34]

As $\mathsf{L}_{\mathsf{cntr}}$ uses the tags of $\mathsf{L}_{\mathsf{trans}}$, we define $\mathsf{L}_{\mathsf{cntr}}.\Lambda_{\text{tags}} \triangleq \{\mathsf{L}_{\mathsf{trans}}\}$. 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();
```

Specifically, 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 specifications.

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 [26]. 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. More generally, our tag system extends the expressivity of [26] with support for *global* effects such as some types of fences.

Existing literature includes several works on formal persistency models, both for hardware [25,30,31,5,6,19,29,28] and software [4,21,11], as well as correctness conditions for persistent libraries such as durable linearizability [17]. As we showed in §3, such models can be specified in our framework.

There have been works [33] to specify libraries using an operational approach instead of the declarative approach that we advocate for here. While it is not generic in the memory model, it support weak memory, with a fragment of the C++ 11 memory model, and supports synchronization that is internal and external to the library. Another framework for formalizing behavior of concurrent objects in the presence of weak memory is [18], which is more syntactic as our framework: they use a process calculus, which allows them to handle callbacks between the library and the client. Extending our framework, which is more semantic, to handle this setting would probably require shifting from executions/histories to something similar to game semantics.

Additionally, there are several works on implementing and verifying algorithms that operate on NVM. [9] and [36] respectively developed persistent queue and set implementations in Px86. [8] provided a formal correctness proof of the implementation in [36]. All three of [8,36,9] assume that the underlying concurrency model is SC [23], 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. [27,2] have developed persistent program logics for verifying programs under Px86. [20] recently formalized the consistency and persistency semantics of the Linux ext4 file system, and developed a model-checking algorithm and tool for verifying the consistency and persistency behaviors of ext4 applications such as text editors.

Recently, and independently to this work, Bodenmüller *et al* [3] have proved the correctness of the Flit library under TSO. They used an operational approach, and modeled the libraries and the memory and persistency models operationally using automata, and proved a simulation result using KIV a specialized proof assistant. As for this paper, they proved that a linearizable library using Flit becomes durably linearizable.

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 [7,27,2], or automatically via model checking. The work of [7] uses the framework of [26] 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. [12,20], 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 fits in general settings for abstract logical relations such as [16].

Acknowledgments

We thank the anonymous reviewers for their feedback. This work has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349). Raad is funded by a UKRI fellowship MR/V024299/1, by the EPSRC grant EP/X037029/1 and by VeTSS.

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. Bodenmüller, S., Derrick, J., Dongol, B., Schellhorn, G., Wehrheim, H.: A fully verified persistency library. In: Dimitrova, R., Lahav, O., Wolff, S. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 26–47. Springer Nature Switzerland, Cham (2024)
4. 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
5. 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)
6. 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
7. Dang, H.H., Jung, J., Choi, J., Nguyen, D.T., Mansky, W., Kang, J., Dreyer, D.: Compass: Strong and compositional library specifications in relaxed memory separation logic. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2022 (2022)
8. 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)
9. 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

10. 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)
11. 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
12. 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
13. Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X.N., Weng, S., Zhang, H., Guo, Y.: Deep specifications and certified abstraction layers. In: POPL (2015)
14. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
15. 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
16. 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)
17. 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)
18. Jagadeesan, R., Petri, G., Pitcher, C., Riely, J.: Quarantining weakness. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. pp. 492–511. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
19. Khyzha, A., Lahav, O.: Taming x86-tso persistency. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021)
20. Kokologiannakis, M., Kaysin, I., Raad, A., Vafeiadis, V.: Persevere: Persistency semantics for verification under ext4. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434324, https://doi.org/10.1145/3434324
21. 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
22. 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
23. 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
24. 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

25. 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)
26. Raad, A., Doko, M., Rožić, L., Lahav, O., Vafeiadis, V.: On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models. POPL (2019)
27. 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`
28. 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)
29. 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`
30. 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)
31. 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)
32. 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`
33. Singh, A.K., Lahav, O.: An operational approach to library abstraction under relaxed memory concurrency. Proc. ACM Program. Lang. **7**(POPL) (jan 2023). `https://doi.org/10.1145/3571246`, `https://doi.org/10.1145/3571246`
34. Stefanesco, L., Raad, A., Vafeiadis, V.: Specifying and verifying persistent libraries (with appendix). CoRR **abs/2306.01614** (2023). `https://doi.org/10.48550/ARXIV.2306.01614`, `https://doi.org/10.48550/arXiv.2306.01614`
35. Wei, Y., Ben-David, N., Friedman, M., Blelloch, G.E., Petrank, E.: Flit: a library for simple and efficient persistent algorithms. In: Lee, J., Agrawal, K., Spear, M.F. (eds.) PPoPP '22. pp. 309–321 (2022)
36. Zuriel, Y., Friedman, M., Sheffi, G., Cohen, N., Petrank, E.: Efficient lock-free durable sets. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). `https://doi.org/10.1145/3360554`, `https://doi.org/10.1145/3360554`

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.