Abstraction for Crash-Resilient Objects*

Artem Khyzha** and Ori Lahav(✉)

Tel Aviv University, Tel Aviv, Israel
artkhyzha@mail.tau.ac.il and orilahav@tau.ac.il

Abstract. We study abstraction for crash-resilient concurrent objects using non-volatile memory (NVM). We develop a library-correctness criterion that is sound for ensuring contextual refinement in this setting, thus allowing clients to reason about library behaviors in terms of their abstract specifications, and library developers to verify their implementations against the specifications abstracting away from particular client programs. As a semantic foundation we employ a recent NVM model, called Persistent Sequential Consistency, and extend its language and operational semantics with useful specification constructs. The proposed correctness criterion accounts for NVM-related interactions between client and library code due to explicit persist instructions, and for calling policies enforced by libraries. We illustrate our approach on two implementations and specifications of simple persistent objects with different prototypical durability guarantees. Our results provide the first approach to formal compositional reasoning under NVM.

Keywords: Non-volatile memory · Linearizability · Library abstraction

1 Introduction

Non-volatile memory, or NVM for short, is an emerging technology that enables byte addressable and high performant storage alongside with data persistency across system crashes. This combination of features allows researchers and practitioners to develop a variety of efficient crash-resilient data structures (see, e.g., [14,32]). Recently, NVM has started to become available in commodity architectures of manufacturers such as Intel and ARM [4,23], and formal (operational and declarative) models of these systems have been proposed [10,25,30].

Unfortunately, like other new technologies, NVM puts more burden on programmers. Indeed, to get close to the performance of DRAM, writes to the NVM are first kept in volatile (i.e., losing contents upon crashes) caches, and only later persist (i.e., propagate to the NVM), possibly not in the order in which they were

* This research was supported by the Israel Science Foundation (grants 1566/18 and 2005/17) and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811). Additionally, the first author was supported by the Blavatnik Family Foundation, and the second by the Alon Young Faculty Fellowship.

** Now at Arm Ltd, Cambridge, UK

I. Sergey (Ed.): ESOP 2022, LNCS 13240, pp. 262–289, 2022.
https://doi.org/10.1007/978-3-030-99336-8_10

issued. This results in counterintuitive behaviors even for sequential programs and requires careful management using barriers of different kinds, a.k.a. explicit persist instructions, for guaranteeing that the system recovers to a consistent state upon a failure. Combined with standard concurrency issues, programming on such machines is highly challenging.

To tackle the complexity and make NVM widely applicable, one would naturally want to draw on libraries encapsulating highly optimized concurrent crash-resilient data structures (a.k.a. persistent objects). This approach goes both ways: programmers should be able to reason about their code using abstract library specifications that hide the implementation details, and in turn, library developers should be able to verify "once and for all" their implementations against their specifications abstracting away from a particular client program. From a formal standpoint, this indispensable modularity requires us to have a so-called *(library) abstraction theorem*: a correctness condition that guarantees the soundness of client reasoning that assumes the specification instead of the implementation. Put differently, the abstraction theorem should allow one to establish *contextual refinement*, i.e., conclude that the specification reproduces the implementation's client-observable behaviors under any (valid) context. To the best of our knowledge, while several correctness criteria for persistent objects, akin to classical linearizability, have been proposed and have been established for multiple sophisticated implementations, none of them has been formally related to contextual refinement by an abstraction theorem of this kind.

In this paper we formulate and prove an abstraction theorem for concurrent programs utilizing non-volatile memory. We target the "Persistent Sequential Consistency" model of [25], or PSC, which enriches the standard sequentially consistent shared-memory with non-volatile storage using per-location FIFO buffers to account for delayed and out-of-order persistence of writes. PSC constitutes a relatively simple model that is very close to developer's informal understanding of NVM. While existing hardware does not implement PSC as is, [25] presented compiler mappings from PSC to x86 (based on its persistency model from [30]), which can be used to ensure PSC semantics on Intel machines. Directly supporting relaxed memory models is left for future work.

Auxiliary material. An extended version, including proofs of theorems stated in the paper, is available at https://arxiv.org/abs/2111.03881.

2 Key Challenges and Ideas

We outline the main challenges and the key ideas in our solutions. We keep the discussion informal, leaving the formal development to later sections.

2.1 Library Specifications

A choice of a formalism for specifying library behaviors is integral in stating a library abstraction theorem. For libraries of concurrent data structures (a.k.a. concurrent objects), a popular approach is to give specifications in terms of sequential objects with the help of the classical notion of linearizability [21], which

requires every sequence of method calls and returns that is possible to produce in a concurrent program to correspond to a sequence that can be generated by the sequential object. In this approach, a sequential object, represented by a set of sequences of pairs of method invocations and their associated responses, constitutes the library specification. Then, abstraction allows the client to reason about calls to a concurrent library as if they execute atomically on a single thread, or, equivalently, protected by a global lock [7,13].

For libraries of crash-resilient objects, there is more than one natural way of interpreting sequential specifications and adapting the linearizability definition, and no single notion of correctness w.r.t. sequential specifications captures all different options. A crash-resilient object may ensure that all methods completed by the moment of crash survive through it, or that some prefix of them does. It may also choose different possibilities for methods in progress at the moment of crash (whether they are allowed take their effect at some later point after the crash or not). Multiple adaptations of linearizability have been proposed, each relating crash-resilient objects to sequential specifications in a different way. This includes: strict linearizability [3], persistent atomicity [19], and durable linearizability and its buffered variant [24]. Among them, buffered durable linearizability, which allows for efficient implementations, ended up not being compositional, which means that it may happen that two (non-interacting) libraries are both correct, but their combination is not. In fact, since each of the different notions is useful for particular objects, one may naturally want to mix different correctness notions in a single client program. This would force the client to reason with several alternatives for interpreting sequential specifications, and to make sure that they compose well with one another.

To approach this variety, we believe it is necessary to follow a different approach, which is standard in concurrent program verification (see, e.g., [18,20,26]), and was applied before for deriving abstraction theorems in different contexts [8,16,17]. The idea is to take a library's specification to be just another library, where the latter is intended to have a simpler implementation. Then, we define a *library correctness condition* stating what it means for one library L to *refine* another library $L^\#$ (equivalently, for $L^\#$ to *abstract* L), and prove an abstraction theorem that ensures that when the library correctness condition is met, the behaviors of any client using L are contained in the behaviors of the client using $L^\#$. Such a theorem is only useful if the correctness condition avoids quantification over all possible clients, which would make the theorem trivial.

Using code for specifying libraries has several advantages over correctness notions based on sequential specifications. First, specifications and implementations are expressed and reasoned about in a unified framework, alleviating the need to interpret the use of sequential specifications by concurrent programs with system failures. Instead, the client of the theorem replaces complex library code with simpler specification code, and thus works with the semantics of a single language. Second, it enables a layered verification technique for library developers, allowing them to prove library correctness by introducing one or more intermediate implementations between L and $L^\#$. Finally, this formulation of

the abstraction theorem is compositional (a.k.a. local) by construction, meaning that objects can be specified and verified in isolation.

Now, "code as a specification" is only useful if the programming language is sufficiently expressive for desirable specifications. For concurrent objects, "atomic blocks", often included in theoretic programming languages, provide a handy specification construct. For NVM, one needs a way to govern the persistence similarly, offering intuitive specifications for libraries that simplify client reasoning. For that matter, viewing the out-of-order persistence of writes to different cache lines as the major source of counterintuitive behaviors in NVM, we propose a new specification construct, which we call *persistence blocks*. Roughly speaking, such blocks may only persist in their entirety, so that persistence blocks ensure an "all-or-nothing" persistency behaviors to the writes they protect.

For example, when recovering after a crash during a run of the tiny program $\dot{\mathtt{x}} := 1; \dot{\mathtt{y}} := 1$,[1] due to out-of-order persistence (writes to different cache lines are not guaranteed to persist in the order in which there were issued), we may reach any combination of values satisfying $\dot{\mathtt{x}} \in \{0,1\} \wedge \dot{\mathtt{y}} \in \{0,1\}$. In turn, if a persistence block is used, as in $\mathtt{beginPB}(\dot{\mathtt{x}}, \dot{\mathtt{y}}); \dot{\mathtt{x}} := 1; \dot{\mathtt{y}} := 1; \mathtt{endPB}(\dot{\mathtt{x}}, \dot{\mathtt{y}})$, then only $\dot{\mathtt{x}} = \dot{\mathtt{y}} = 0 \vee \dot{\mathtt{x}} = \dot{\mathtt{y}} = 1$ are possible upon recovery.

Our blocks are closely related to persistent transactions of the PMDK library [22] (but we avoid the term transaction, since persistence blocks do not ensure isolation when executed concurrently). In our technical development, we extend the PSC model with instructions for persistence blocks, and carefully construct their semantics (see §4.2) to allow the abstraction result. We believe that persistence blocks are a useful *specification construct* for various data structures, where data consistency naturally involves multiple locations (often, pointers) being in-sync with one another.

2.2 Client-Library Interaction Using Explicit Persist Instructions

The key to establishing a library abstraction theorem is in decomposing a program into two interacting sub-parts, a client and a library, and understanding the interactions between them. These interactions are usually defined in terms of *histories*, taken to be sequences of method invocations and responses, along with the values being passed. The library correctness condition (the premise of the abstraction theorem) requires that histories produced by using a library L are also produced by its specification $L^{\#}$ when both libraries are used by a certain "most general client" (MGC, for short) that concurrently invokes arbitrary methods of L an arbitrary number of times with every possible argument. The abstraction theorem ensures that if the library correctness condition holds, then L refines $L^{\#}$ for *any* client.

Thus, for the abstraction theorem to hold, one has to make sure that the interactions between any client and the library are fully captured in the history produced by the library when used by the MGC. In *crash-free* sequentially

[1] We use "overdots" to denote non-volatile variables. We assume that all variables are initialized to 0 and that $\dot{\mathtt{x}}$ and $\dot{\mathtt{y}}$ lie on different cache lines.

consistent shared memory semantics, this is ensured by the standard assumption that the client and the library manipulate disjoint set of memory locations. Indeed, this restriction guarantees that clients can communicate with libraries only via values passed to and returned from method invocations.

However, we observe that under NVM, mutual interactions between the client and the library go beyond passed values, even when assuming disjointness of memory locations, which makes the standard notion of a library history insufficient. As a simple example, consider an interface with just one method f, specified by $L^{\#} = [f \mapsto \texttt{sfence}; \texttt{return}]$. The `sfence` instruction, called "store fence", is an explicit persist instruction meant to be used in conjunction with optimized barriers called "flush-optimal" (denoted by `fo`). Its role is to guarantee the persistence of previous write instructions that are guarded by flush-optimal instructions. Concretely, under PSC (following x86), after a thread executes $\dot{\mathtt{x}} := 1; \texttt{fo}(\dot{\mathtt{x}}); \texttt{sfence}$, we know that the write of 1 to $\dot{x}$ has persisted (i.e., been propagated to the NVM), while without the `sfence`, it may still sit in the volatile part of the memory system.

In turn, consider an implementation L, given by $L = [f \mapsto \texttt{return}]$, that implements f by doing nothing. Clearly, L does not implement $L^{\#}$ correctly. Indeed, for the (sequential) client program $\dot{\mathtt{x}} := 1; \texttt{fo}(\dot{\mathtt{x}}); \texttt{call}(f); \dot{\mathtt{y}} := 1$ that uses $L^{\#}$, we have $\dot{\mathtt{y}} = 1 \implies \dot{\mathtt{x}} = 1$ as a global invariant: if the system has crashed and we have $\dot{\mathtt{y}} = 1$ in the NVM, then the `sfence` ensures that $\dot{\mathtt{x}} = 1$ is in the NVM as well. Nevertheless, due to out-of-order persistence, if we use L in this program, we may get $\dot{\mathtt{y}} = 1 \wedge \dot{\mathtt{x}} = 0$ after a crash. Now, the client and the libraries above mention disjoint locations, and the histories that L may produce for the MGC are exactly the histories that $L^{\#}$ produces (all well-formed sequences of "call" and "return"). Thus, when inspecting histories of L and of $L^{\#}$, we do not have sufficient information to observe the difference between them.

Generally speaking, the challenge stems from the fact that certain explicit persist instructions (`sfence` and other instructions whose implementation in the hardware contains an implicit store fence, such as RMWs in x86), which can be executed by the library, impose conditions on the persistence of writes performed by the client that ran earlier on the same processor.

We address this challenge in two ways. First, we can sidestep the problem by weakening the semantics of store fences, making them relative to a set of locations (those used by the library or those used by the client). To do so, we extend the programming language with a specification construct similar to a store fence, but only affecting a given set of locations, and we restrict its use by each component to mention only the locations it owns. The use of these localized instructions instead of store fences is sufficient to ensure that the interaction between client and library is fully captured in histories, and allows us to establish the expected abstraction theorem. Libraries that do not intend to provide a store fence functionality to their clients can readily replace store fences with their localized counterparts. Doing so gives more freedom to alternative implementations of the same specification, which may, e.g., use alternative persist instructions without the store fence functionality (such as CLFLUSH in [23]).

On the other hand, it is possible that in performance-critical systems, clients would like to rely on a store fence that is executed anyway by the library for the library's own needs. For that, the library developer needs to use a standard store fence in the library's specification rather than the localized counterpart, and the abstraction theorem has to handle store fences with their standard, non-localized semantics. To do so, we expose in histories not only method invocations and responses, but also store fences. Roughly speaking, it means that in addition to the standard requirement on values passed by method invocations and responses, for L to refine $L^{\#}$, we would also require that L performs a store fence whenever $L^{\#}$ does (which does not hold for the example above). Our notion of history in §5 is set to allow store fences (alongside with their weaker localized versions), and the abstraction theorem in §6 shows that these extended histories are expressive enough for defining the library-correctness condition.

2.3 Handling Calling Policies

The third challenge we address concerns abstraction for libraries that enforce certain calling policies on their clients.[2] For instance, a library implementing a lock may require that the calls of each thread for acquiring and releasing the lock perfectly interleave, and a library implementing a single-producer queue may require that only one thread is calling the enqueue method. In the context of NVM, libraries often demand that a distinguished *recovery method* is called after every crash before invoking any other method of the library. When the client uses the library in a way that violates the calling policy, the library developer ensures nothing, and the blame is assigned to the client.

In the presence of calling policies, the contextual refinement guaranteed by the library abstraction theorem, stating that all behaviors of a program $Pr[L]$ that uses L are also behaviors of the program $Pr[L^{\#}]$ that uses $L^{\#}$, is only applicable for a program Pr that respects the calling policy. An interesting compositionality question arises: Are we allowed to assume the library's specification when checking whether a program adheres to the calling policy (that is, require that $Pr[L^{\#}]$ adheres to the policy), or should this obligation be satisfied for the library's implementation (that is, require that $Pr[L]$ adheres to the policy)?

The latter option would limit the applicability of the abstraction theorem for client reasoning. Indeed, it may be the case that establishing that $Pr[L]$ adheres to the policy depends on the implementation L, whereas the abstraction theorem should allow reasoning without knowing the implementation *at all*. On the other hand, the former option seems circular, as it uses contextual refinement to establish its own precondition.

In this paper we show that requiring that $Pr[L^{\#}]$ adheres to the policy is actually sufficient for ensuring contextual refinement. Roughly speaking, our proof avoids circular reasoning by inspecting a *minimal* contextual refinement violation, for which we are able to establish policy adherence when using L, given

[2] This challenge is not particular to NVM, but, interestingly, to the best of our knowledge, it has not been addressed in previous work establishing abstraction theorems.

policy adherence when using $L^\sharp$. To the best of our knowledge, this is a novel argument in the context of library abstraction. It is akin to DRF (data-race freedom) guarantees in weak memory concurrency, where often programs are guaranteed to have strong semantics (usually, sequential consistency) provided that certain race-freedom conditions hold in all runs under the *strong* semantics.

We note that many library's calling policies are "structural", namely they only enforce certain ordering constraints on the clients that do not depend on the values returned by the library (in particular, "execute recovery first" is a structural policy). In these cases, policy adherence holds even for an over-approximation L_{stub} of L that returns arbitrary values. Certainly, however, this is not always the case. For example, a library L implementing standard list methods, cons and head, may require that head is only called on non-empty lists (like, e.g., pop_front in C++ that triggers undefined behavior if applied to an empty list [1]). Then, invoking head with the value returned from cons does adhere to the calling policy, but this is not the case for the over-approximated library L_{stub}, which allows cons to return the empty list.

3 NVM Programs: Syntax and Semantics

In this section we begin to present the formal settings for our results. As standard in memory models, it is convenient to break the operational semantics into: a *program* semantics (a.k.a. thread subsystem) and a *memory* semantics. We represent both components as labeled transition systems whose transition labels correspond to the operations they perform. We then consider the synchronized runs of the program and the memory, where program actions that interact with the memory are matched by actions executed by the memory system (see §4.1).

Next, we focus on the program part of the semantics, presenting both syntax (§3.1) and semantics (§3.2). We use the following standard notations.

Notation for finite sequences. For a finite alphabet Σ, we denote by Σ^* (respectively, Σ^+) the set of all (non-empty) sequences over Σ. We use ϵ to denote the empty sequence. The length of a sequence s is denoted by $|s|$. We often identify sequences with their underlying functions (whose domain is $\{1, \ldots, |s|\}$), and write $s(k)$ for the symbol at position $1 \leq k \leq |s|$ in s. We write $\sigma \in s$ if σ appears in s, that is if $s(k) = \sigma$ for some $1 \leq k \leq |s|$. We use "$\cdot$" for concatenating sequences, and identify symbols with sequences of length 1.

3.1 Program Syntax

The domains and metavariables used to range over them are as follows:

values	$v, u \in \mathsf{Val} = \{0, 1, 2, \ldots\}$
shared non-volatile variables	$\dot{x}, \dot{y} \in \mathsf{NVVar} = \{\dot{\mathtt{x}}, \dot{\mathtt{y}}, \ldots\}$
shared volatile variables	$\tilde{x}, \tilde{y} \in \mathsf{VVar} = \{\tilde{\mathtt{x}}, \tilde{\mathtt{y}}, \ldots\}$
shared variables	$x, y \in \mathsf{Var} = \mathsf{NVVar} \cup \mathsf{VVar}$
register names	$r \in \mathsf{Reg} = \{\mathtt{a}, \mathtt{b}, \ldots\}$
thread identifiers	$\tau, \pi \in \mathsf{Tid} = \{\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_\mathsf{N}\}$
method names	$f \in \mathsf{F} \qquad \mathsf{main} \notin \mathsf{F}$

Thus, there are three kinds of variables: shared non-volatile, shared volatile, and thread-local ones (called registers), which are also volatile. A distinguished name main is reserved for the starting point of the program execution.

For concreteness, we present a simple programming-language syntax. Its expressions and instructions are given by the following grammar:[3]

$$
\begin{aligned}
e ::= \quad & r \mid v \mid e + e \mid e = e \mid e \neq e \mid \ldots \\
\mathit{inst} ::= {} & r := e \mid \texttt{if } e \texttt{ goto } n_1 \mid \ldots \mid n_m \mid \texttt{havoc} \mid x := e \mid r := x \\
& \mid \texttt{fl}(\dot{x}) \mid \texttt{fo}(\dot{x}) \mid \texttt{sfence} \mid \texttt{call}(f) \mid \texttt{return} \\
& \mid \texttt{lsfence}(\dot{X}) \mid \texttt{beginPB}(\dot{X}) \mid \texttt{endPB}(\dot{X})
\end{aligned}
$$

Expressions are constructed with arithmetic and boolean operations over registers and values. Instructions consist of a local assignment $r := e$; a conditional $\texttt{if } e \texttt{ goto } n_1 \mid \ldots \mid n_m$ for non-deterministically jumping to a program counter from $\{n_1, \ldots, n_m\}$ when e evaluates to non-zero or, otherwise, skipping ($\texttt{goto } n_1 \mid \ldots \mid n_m$ can be encoded as $\texttt{if } 1 \texttt{ goto } n_1 \mid \ldots \mid n_m$); havoc for arbitrarily modifying all registers; a write to memory $x := e$; and a read from memory $r := x$. There are also explicit persist instructions: a *flush* instruction $\texttt{fl}(\dot{x})$ and its optimized version $\texttt{fo}(\dot{x})$, called *flush-optimal* (referred to as CLFLUSH and CLFLUSHOPT in [23]), as well as the store fence instruction sfence (see §2.2).

This standard instruction set is extended to support calling and specifying library methods. There is a call instruction $\texttt{call}(f)$ and a return instruction return. The novel specification constructs include the *local store fence* instruction $\texttt{lsfence}(\dot{X})$ that relaxes the semantics of sfence by only enforcing the persistence ordering for the given set $\dot{X}$ of variables (thus, $\texttt{lsfence}(\mathsf{NVVar})$ is equivalent to sfence); and instructions to begin and end a *persistence block*, $\texttt{beginPB}(\dot{X})$ and $\texttt{endPB}(\dot{X})$, respectively. The persistence block demarks the writes that need to persist simultaneously after the block ends, either non-deterministically or triggered by a flush on some variable in $\dot{X}$.

Next, we employ three syntactic categories:

- *Instruction sequences* represent the (sequential) implementation of each method (including main). Formally, an instruction sequence I is a function from a non-empty finite domain of the form $\{0, \ldots, n\}$ (representing the possible program counters) to the set of instructions. We say that an instruction sequence is *flat* if it does not include an instruction of the form $\texttt{call}(\_)$.
- *Sequential programs* consist of a "main" method accompanied with implementations of every method $f \in \mathsf{F}$. Formally, a sequential program S is a function assigning an instruction sequence to every $f \in \{\mathsf{main}\} \cup \mathsf{F}$. To avoid modeling a call stack and simplify the presentation, we require that $S(f)$ is a flat instruction sequence for every $f \in \mathsf{F}$.
- *Concurrent programs* are top-level parallel compositions of sequential programs, all accompanied by the same method implementations. Formally, a (concurrent) program Pr is a mapping assigning a sequential program to every $\tau \in \mathsf{Tid}$, with $Pr(\tau)(f) = Pr(\pi)(f)$ for every $\tau, \pi \in \mathsf{Tid}$ and $f \in \mathsf{F}$. Below, we write $Pr(f)$ for $Pr(\mathtt{T}_1)(f)$.

[3] In the extended version of this paper, we also include read-modify-write instructions.

3.2 Program Semantics

We give semantics to the syntactic objects using labeled transition systems.

Definition 1. A *labeled transition system* (LTS) is a tuple $A = \langle \Sigma, Q, q_{\mathsf{Init}}, T \rangle$, where Σ is a set of *transition labels*, Q is a set of *states*, $q_{\mathsf{Init}} \in Q$ is the *initial state*, and $T \subseteq Q \times \Sigma \times Q$ is a set of *transitions*. We often write $q \xrightarrow{\sigma} q'$ to denote a transition $\langle q, \sigma, q' \rangle$. We denote by $A.\mathbf{\Sigma}$, $A.\mathtt{Q}$, $A.\mathsf{q_{Init}}$, and $A.\mathtt{T}$ the components of an LTS A. We write $\xrightarrow{\sigma}_A$ for the relation $\{\langle q, q' \rangle \mid q \xrightarrow{\sigma} q' \in A.\mathtt{T}\}$ and $\rightarrow_A$ for $\bigcup_{\sigma \in \Sigma} \xrightarrow{\sigma}_A$. For a sequence $t \in A.\mathbf{\Sigma}^*$, we write $\xrightarrow{t}_A$ for the composition $\xrightarrow{t(1)}_A ; \ldots ; \xrightarrow{t(|t|)}_A$. A sequence $t \in A.\mathbf{\Sigma}^*$ such that $A.\mathsf{q_{Init}} \xrightarrow{t}_A q$ for some $q \in A.\mathtt{Q}$ is called a *trace* of A. We denote by $\mathsf{traces}(A)$ the set of all traces of A. A state $q \in A.\mathtt{Q}$ is called *reachable* in A if $A.\mathsf{q_{Init}} \xrightarrow{t}_A q$ for some $t \in \mathsf{traces}(A)$.

Next, we define the LTSs induced by instruction sequences, sequential programs, and concurrent programs. We will often identify the syntactic objects with the LTS they induce (e.g., when writing expressions like $S.\mathtt{Q}$ for a sequential program S). The transition labels of these LTSs feature *action labels*.

Definition 2. An *action label* takes one of the following forms: a read $\mathtt{R}(x, v)$, a write $\mathtt{W}(x, v)$, a flush $\mathtt{FL}(\dot{x})$, a flush-opt $\mathtt{FO}(\dot{x})$, an sfence $\mathtt{SF}$, a local sfence $\mathtt{LSF}(\dot{X})$, a start $\mathsf{beginPB}(\dot{X})$ or an end $\mathsf{endPB}(\dot{X})$ of a persistence block, a call $\mathtt{CALL}(f, \phi)$, or a return $\mathtt{RET}(f, \phi)$, where $x \in \mathsf{Var}$, $v \in \mathsf{Val}$, $\dot{x} \in \mathsf{NVVar}$, $\dot{X} \subseteq \mathsf{NVVar}$, $f \in \mathsf{F}$, and $\phi : \mathsf{Reg} \to \mathsf{Val}$. We denote by Lab the set of all action labels. The functions $\mathtt{typ}$ and $\mathtt{var}$ retrieve (when applicable) the type ($\mathtt{R}/\mathtt{W}/\ldots$) and variable ($x$ or $\dot{x}$) of an action label. We write $\mathtt{varset}(l)$ for the set of variables mentioned in l (e.g., $\mathtt{varset}(\mathtt{R}(x, v)) = \{x\}$, $\mathtt{varset}(\mathtt{LSF}(\dot{X})) = \dot{X}$, and $\mathtt{varset}(\mathtt{SF}) = \emptyset$).

Action labels represent the interactions that a program has with the memory.

Definition 3. The LTS induced by an instruction sequence I is given by:

- The transition labels are action labels, extended with ϵ for silent transitions.
- The states are pairs $\langle pc, \phi \rangle$ where $pc \in \mathbb{N}$, called *program counter*, stores the current instruction pointer inside the sequence, and $\phi : \mathsf{Reg} \to \mathsf{Val}$, called *local store*, records the values of the registers. We assume that local stores are extended to expressions in the obvious way.
- The initial state is $\langle 0, \phi_{\mathsf{Init}} \rangle$, where $\phi_{\mathsf{Init}} \stackrel{\text{def}}{=} \lambda r.\, 0$.
- The transitions are as follows:

$$\frac{I(pc) = r := e \quad \phi' = \phi[r \mapsto \phi(e)]}{\langle pc, \phi \rangle \xrightarrow{\epsilon}_I \langle pc + 1, \phi' \rangle} \qquad \frac{I(pc) = \mathtt{if}\ e\ \mathtt{goto}\ n_1 \mid \ldots \mid n_m \quad \phi(e) \neq 0 \implies pc' \in \{n_1, \ldots, n_m\} \quad \phi(e) = 0 \implies pc' = pc + 1}{\langle pc, \phi \rangle \xrightarrow{\epsilon}_I \langle pc', \phi \rangle} \qquad \frac{I(pc) = \mathtt{havoc}}{\langle pc, \phi \rangle \xrightarrow{\epsilon}_I \langle pc + 1, \phi' \rangle}$$

$$\frac{I(pc) = x := e \quad l = \mathtt{W}(x, \phi(e))}{\langle pc, \phi \rangle \xrightarrow{l}_I \langle pc + 1, \phi \rangle} \qquad \frac{I(pc) = r := x \quad l = \mathtt{R}(x, v) \quad \phi' = \phi[r \mapsto v]}{\langle pc, \phi \rangle \xrightarrow{l}_I \langle pc + 1, \phi' \rangle} \qquad \frac{I(pc) \in \left\{ \begin{array}{l} \mathtt{fl}(\_), \mathtt{fo}(\_), \\ \mathtt{sfence}, \mathtt{lsfence}(\_), \\ \mathtt{beginPB}(\_), \mathtt{endPB}(\_) \end{array} \right\} \quad l = \mathsf{matching\_label}(I(pc))}{\langle pc, \phi \rangle \xrightarrow{l}_I \langle pc + 1, \phi \rangle}$$

Recall that program semantics is separate from memory semantics, which is why the transitions above completely ignore the restrictions arising from the memory system. In particular, the write to memory $x := e$ only announces itself in the label. The read from memory $r := x$ loads an arbitrary value v into the destination register r, announcing that value in the read label. Other instructions act as no-ops, and simply announce themselves in the transition label, using the function matching_label that maps each instruction to its label ($\mathtt{fl}(\dot{x}) \mapsto \mathtt{FL}(\dot{x}), \mathtt{fo}(\dot{x}) \mapsto \mathtt{FO}(\dot{x})$, and so on).

Finally, $\mathtt{call}(f)$ and `return` instructions are not handled in this level, but receive special semantics at the level of sequential programs, as defined next.

Definition 4. The LTS induced by a sequential program S is given by:

- The transition labels are action labels, extended with ϵ for silent transitions.
- The states are tuples $q = \langle pc, \phi, pc_{\mathsf{s}}, f \rangle$, where:
  - $\langle pc, \phi \rangle$ is a state of the instruction sequence (see Def. 3) storing the state of the sequence currently running.
  - $pc_{\mathsf{s}} \in \mathbb{N} \cup \{\bot\}$, called *the stored program counter*, is used to remember the program position to jump to when the current instruction sequence returns, whereas $pc_{\mathsf{s}} = \bot$ means that the main method is currently running. (Recall that we assume that $S(f)$ is flat for every $f \in \mathsf{F}$, so we do not need to record the call stack.)
  - $f \in \mathsf{F} \cup \{\mathsf{main}\}$, called *the active method*, tracks the method that is currently running.

  We denote by $q.\mathtt{pc}$, $q.\phi$, $q.\mathtt{pc_s}$, and $q.\mathtt{f}$ the components of a state $q \in S.\mathtt{Q}$.
- The initial state is $\langle 0, \phi_{\mathsf{Init}}, \bot, \mathsf{main} \rangle$.
- The transitions are given by:

$$\textsc{normal}\quad \frac{l_\epsilon \in \mathsf{Lab} \cup \{\epsilon\} \qquad f \in \{\mathsf{main}\} \cup \mathsf{F} \qquad \langle pc, \phi \rangle \xrightarrow{l_\epsilon}_{S(f)} \langle pc', \phi' \rangle}{\langle pc, \phi, pc_{\mathsf{s}}, f \rangle \xrightarrow{l_\epsilon}_{S} \langle pc', \phi', pc_{\mathsf{s}}, f \rangle}$$

$$\textsc{call}\quad \frac{S(\mathsf{main})(pc) = \mathtt{call}(f) \qquad l = \mathtt{CALL}(f, \phi)}{\langle pc, \phi, \bot, \mathsf{main} \rangle \xrightarrow{l}_{S} \langle 0, \phi, pc + 1, f \rangle}$$

$$\textsc{return}\quad \frac{S(f)(pc) = \mathtt{return} \qquad l = \mathtt{RET}(f, \phi)}{\langle pc, \phi, pc_{\mathsf{s}}, f \rangle \xrightarrow{l}_{S} \langle pc_{\mathsf{s}}, \phi, \bot, \mathsf{main} \rangle}$$

$$\textsc{non-det-sfence}\quad \frac{l = \mathtt{SF}}{\langle pc, \phi, pc_{\mathsf{s}}, f \rangle \xrightarrow{l}_{S} \langle pc, \phi, pc_{\mathsf{s}}, f \rangle}$$

The NORMAL transition lifts the instruction-sequence transition to the level of sequential programs. Note that the transition applies for any method (main or other). The CALL transition passes control from the main method to some other method, jumping the program counter to the first instruction and storing the return point (pc+1). The RETURN transition passes control back using the stored return point. For simplicity, we do not have any argument passing mechanism and use the full register store for that matter. (If needed, each component may store the values it needs in the memory, and reload them later on.)

Finally, NON-DET-SFENCE is a non-standard transition that we find technically convenient to have. It allows the program to non-deterministically execute

an sfence at any point. Since, as will become apparent when presenting the memory system, sfences only restrict the possible behaviors, this transition is safe to include in the program semantics. It is particularly useful for simplifying the library correctness condition that only considers inclusion of sets of histories (see §5). For instance, switching the roles of L and $L^{\sharp}$ from §2.2, the library implementing f using `sfence` should be considered a refinement of the one that simply returns. For that, we allow the no-op specification to perform non-deterministic sfences that match the ones executed by the concrete implementation.

Finally, the LTS induced by a concurrent program is defined as follows.

Definition 5. The LTS induced by a (concurrent) program Pr is given by:

- The set of transition labels is given by $(\mathsf{Tid} \times (\mathsf{Lab} \cup \{\epsilon\})) \cup \{\lightning\}$. The functions on action labels (e.g., `typ`, `var`) are lifted to these labels in the obvious way.
- The states, denoted by $\overline{q}$, assign a state in $Pr(\tau).\mathtt{Q}$ to every $\tau \in \mathsf{Tid}$.
- The initial state is composed from the initial state of each thread: $\overline{q}_{\mathsf{Init}} \stackrel{\text{def}}{=} \langle Pr(\mathtt{T}_1).\mathtt{q}_{\mathsf{Init}}, \ldots, Pr(\mathtt{T}_{\mathsf{N}}).\mathtt{q}_{\mathsf{Init}} \rangle$.
- The transitions are interleaved thread transitions or crash transitions reinitializing the program state:

$$\text{NORMAL}\ \frac{l_\epsilon \in \mathsf{Lab} \cup \{\epsilon\} \qquad \overline{q}(\tau) \xrightarrow{l_\epsilon}_{Pr(\tau)} q'}{\overline{q} \xrightarrow{\tau, l_\epsilon}_{Pr} \overline{q}[\tau \mapsto q']} \qquad \text{CRASH}\ \frac{}{\overline{q} \xrightarrow{\lightning}_{Pr} \overline{q}_{\mathsf{Init}}}$$

4 The PSC Memory System

We present PSC ("Persistent Sequential Consistency"), the persistency model used as the memory system. We first introduce the model as it is in [25] (extended with standard volatile memory alongside with the non-volatile one), following its operational presentation as an LTS with non-deterministic memory-internal transitions that flush stores from the volatile part to the non-volatile part. In §4.1, we define the synchronization of programs with the PSC memory system. In §4.2, we present the extensions added in this paper that are useful for library abstraction. Finally, in §4.3, we establish certain separation properties of PSC that are essential in our proofs.

Roughly speaking, a state in PSC consists of a non-volatile memory (mapping from non-volatile variables to values) and a volatile memory (mapping from volatile variables to values). The volatile memory works just as a normal sequentially consistent memory, keeping track of the latest written value to every variable and returning that value for reads. Upon crash, the contents of the volatile memory is reset to its initial state. The non-volatile memory behaves observationally the same between crashes, but its contents survive crashes. To model delayed and out-of-order persistence of writes, write steps to non-volatile variables do not alter the non-volatile memory immediately when issued. Instead, writes first go to volatile per-variable persistence FIFO buffers, which maintain the writes to each variable that are yet to persist. Then, PSC non-deterministically takes *persist steps* that apply the oldest update from a persistence buffer in the

non-volatile memory. Reads from non-volatile variables retrieve the latest value in the relevant buffer, or the value from the non-volatile memory if that buffer is empty, thus providing standard sequentially consistent semantics in the absence of system crashes. Upon crash the buffers are reset to their initial (empty) state, but the contents of the non-volatile memory remains intact.

Explicit persist instructions can be used to control the persistence of writes. A "flush" barrier for a certain variable blocks the execution until the relevant persistence buffer is empty, thus forcing all previous writes to that variable to persist. Alternatively, a (cheaper) "flush-optimal" barrier for a certain variable enqueues a special marker in the persistence buffer of this variable accompanied by the thread identifier of the thread that issued the barrier. The effect of flush-optimal is delayed until the same thread performs an sfence, which blocks the execution until all flush-optimal markers of that thread are dequeued from all buffers. The fact that the persistence buffers are FIFO ensures that an sfence by some thread forces the persistence of all writes executed before a flush-optimal issued by the same thread.

Definition 6. PSC is the LTS defined as follows:

- The transition labels are given by $(\mathsf{Tid} \times \mathsf{Lab}) \cup \{\mathtt{per}, \lightning\}$. That is, a transition label can be a pair of the thread identifier and the action label of the operation, `per` denoting the internal propagation action, or $\lightning$ denoting a system crash.
- The states are tuples $M = \langle \dot{m}, \tilde{m}, P \rangle$, where:
  - $\dot{m} : \mathsf{NVVar} \to \mathsf{Val}$ is called the *non-volatile memory*.
  - $\tilde{m} : \mathsf{VVar} \to \mathsf{Val}$ is called the *volatile memory*.
  - $P : \mathsf{NVVar} \to \mathsf{PLBuff}$ is called the *persistence buffer*. Here, PLBuff denotes the set of all *per-location persistence buffers*, each of which is a finite sequence p of entries of the form $\mathtt{W}(v)$ for $v \in \mathsf{Val}$ (writes), or $\mathtt{FO}(\tau)$ for $\tau \in \mathsf{Tid}$ (flush optimal markers). The persistence buffer P assigns a per-location persistence buffer to every non-volatile variable.[4]

  We denote by $M.\dot{\mathtt{m}}$, $M.\tilde{\mathtt{m}}$, and $M.\mathtt{P}$ the components of a state $M \in \mathsf{PSC}.\mathtt{Q}$, and write $M[\mathtt{X} \mapsto Y]$ for the state obtained from M by setting $M.\mathtt{X}$ to Y.
- The initial state is $M_{\mathsf{Init}} \stackrel{\text{def}}{=} \langle \dot{m}_{\mathsf{Init}}, \tilde{m}_{\mathsf{Init}}, P_{\mathsf{Init}} \rangle$, where $\dot{m}_{\mathsf{Init}} \stackrel{\text{def}}{=} \lambda \dot{x}.\, 0$, $\tilde{m}_{\mathsf{Init}} \stackrel{\text{def}}{=} \lambda \tilde{x}.\, 0$, and $P_{\mathsf{Init}} \stackrel{\text{def}}{=} \lambda \dot{x}.\, \epsilon$.
- The transitions of PSC are presented in Fig. 1, using an auxiliary function for looking up the most recent value of a variable: we let $M(x)$ be $M.\tilde{\mathtt{m}}(x)$ for $x \in \mathsf{VVar}$, and, for $x \in \mathsf{NVVar}$, either the value v of the last write (rightmost) entry $M.\mathtt{P}(x)$ or, when there is no such entry, $M.\dot{\mathtt{m}}(x)$.

The transitions follow the intuitive account above. Those corresponding to program transitions are labeled with pairs in $\mathsf{Tid} \times \mathsf{Lab}$. For instance, a transition labeled with $\langle \tau, \mathtt{R}(x, v_{\mathtt{R}}) \rangle$ means that thread τ reads the value $v_{\mathtt{R}}$ from (volatile or non-volatile) shared variable x.

[4] We conservatively assume that writes persist at the location granularity, rather than at the cache-line granularity as happens in real machines.

$$\textsc{V-WRITE}\ \frac{l = \mathtt{W}(\tilde{x}, v) \quad \tilde{m}' = M.\tilde{\mathtt{m}}[\tilde{x} \mapsto v]}{M \xrightarrow{\tau,l}_{\mathsf{PSC}} M[\tilde{\mathtt{m}} \mapsto \tilde{m}']} \qquad \textsc{NV-WRITE}\ \frac{l = \mathtt{W}(\dot{x}, v) \quad p' = M.\mathtt{P}(\dot{x}) \cdot \mathtt{W}(v) \quad P' = M.\mathtt{P}[\dot{x} \mapsto p']}{M \xrightarrow{\tau,l}_{\mathsf{PSC}} M[\mathtt{P} \mapsto P']} \qquad \textsc{READ}\ \frac{l = \mathtt{R}(x, v) \quad M(x) = v}{M \xrightarrow{\tau,l}_{\mathsf{PSC}} M}$$

$$\textsc{FLUSH}\ \frac{l = \mathtt{FL}(\dot{x}) \quad M.\mathtt{P}(\dot{x}) = \epsilon}{M \xrightarrow{\tau,l}_{\mathsf{PSC}} M} \qquad \textsc{FLUSH-OPT}\ \frac{l = \mathtt{FO}(\dot{x}) \quad p' = M.\mathtt{P}(\dot{x}) \cdot \mathtt{FO}(\tau) \quad P' = M.\mathtt{P}[\dot{x} \mapsto p']}{M \xrightarrow{\tau,l}_{\mathsf{PSC}} M[\mathtt{P} \mapsto P']} \qquad \textsc{SFENCE}\ \frac{l = \mathtt{SF} \quad \forall \dot{x}.\, \mathtt{FO}(\tau) \notin M.\mathtt{P}(\dot{x})}{M \xrightarrow{\tau,l}_{\mathsf{PSC}} M}$$

$$\textsc{PERSIST-WRITE}\ \frac{l = \mathtt{per} \quad M.\mathtt{P}(\dot{x}) = \mathtt{W}(v) \cdot p \quad P' = M.\mathtt{P}[\dot{x} \mapsto p] \quad \dot{m}' = M.\dot{\mathtt{m}}[\dot{x} \mapsto v]}{M \xrightarrow{l}_{\mathsf{PSC}} M[\dot{\mathtt{m}} \mapsto \dot{m}', \mathtt{P} \mapsto P']} \qquad \textsc{PERSIST-FO}\ \frac{l = \mathtt{per} \quad M.\mathtt{P}(\dot{x}) = \mathtt{FO}(\tau) \cdot p \quad P' = M.\mathtt{P}[\dot{x} \mapsto p]}{M \xrightarrow{l}_{\mathsf{PSC}} M[\mathtt{P} \mapsto P']} \qquad \textsc{CRASH}\ \frac{l = \lightning}{M \xrightarrow{l}_{\mathsf{PSC}} M_{\mathsf{Init}}[\dot{\mathtt{m}} \mapsto M.\dot{\mathtt{m}}]}$$

Fig. 1. Transitions of PSC

4.1 Linking Programs and Memories

To give semantics of programs running under PSC, the thread system is synchronized with the PSC memory system. Formally, the synchronization of a program Pr with PSC, is another LTS, denoted by $Pr \bowtie \mathsf{PSC}$, defined as follows:

- The set of transition labels is $Pr.\mathbf{\Sigma} \cup \mathsf{PSC}.\mathbf{\Sigma}$, i.e., $(\mathsf{Tid} \times (\mathsf{Lab} \cup \{\epsilon\})) \cup \{\mathtt{per}, \lightning\}$.
- The states are pairs $\langle \overline{q}, M \rangle \in Pr.\mathtt{Q} \times \mathsf{PSC}.\mathtt{Q}$.
- The initial state is $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle$.
- The transitions are given by:

$$\textsc{SYNCHRONIZED}\ \frac{\alpha \in (\mathsf{Tid} \times \mathsf{Lab}) \cup \{\lightning\} \quad \overline{q} \xrightarrow{\alpha}_{Pr} \overline{q}' \quad M \xrightarrow{\alpha}_{\mathsf{PSC}} M'}{\langle \overline{q}, M \rangle \xrightarrow{\alpha}_{Pr \bowtie \mathsf{PSC}} \langle \overline{q}', M' \rangle} \qquad \textsc{PROGRAM-INTERNAL}\ \frac{\alpha \in \mathsf{Tid} \times \{\epsilon\} \quad \overline{q} \xrightarrow{\alpha}_{Pr} \overline{q}'}{\langle \overline{q}, M \rangle \xrightarrow{\alpha}_{Pr \bowtie \mathsf{PSC}} \langle \overline{q}', M \rangle} \qquad \textsc{MEMORY-INTERNAL}\ \frac{\alpha = \mathtt{per} \quad M \xrightarrow{\alpha}_{\mathsf{PSC}} M'}{\langle \overline{q}, M \rangle \xrightarrow{\alpha}_{Pr \bowtie \mathsf{PSC}} \langle \overline{q}, M' \rangle}$$

The above transitions are "synchronized transitions" of Pr and PSC, using the labels to decide what to synchronize on. Both the program and the memory take the same step for transition labels that are common to both LTSs, only the program steps for transition labels that are only program transitions, and only the memory steps for transition labels that are only memory transitions.

4.2 Extending PSC for Library Abstraction

We present the modifications of PSC for supporting the new specification constructs: localized sfences and persistence blocks. When referring to PSC in the sequel we mean the following revised version.

Local store fences. Localized sfences are straightforwardly supported by the following additional memory transition:

$$\textsc{LOCAL SFENCE}\ \frac{l = \mathtt{LSF}(\dot{X}) \quad \forall \dot{x} \in \dot{X}.\, \mathtt{FO}(\tau) \notin M.\mathtt{P}(\dot{x})}{M \xrightarrow{\tau,l}_{\mathsf{PSC}} M}$$

Here, instead of blocking until all $\texttt{FO}(\tau)$ entries are removed from all buffers, we only require that such entries are not present in buffers associated with variables from a certain set (mentioned in the action label and corresponding to the argument of the $\texttt{lsfence}(\dot{X})$ instruction).

Persistence blocks. We assume an infinite set $\mathsf{BlockID}$ of block identifiers that are non-deterministically allocated when blocks are opened. The state of the memory system keeps track of a mapping assigning the current open block identifier to every thread and non-volatile variable, or $\bot$ if the variable is not a part of an open block of the thread. When writing to non-volatile variables, the associated block identifiers are attached to the write entry in the per-location persistence buffer. In turn, the propagation from the buffers to the NVM ensures that blocks are propagated only after they are not open and only in their entirety. To do so, we generalize the persist step of PSC to allow simultaneous propagation of multiple entries from the buffers. To respect the per-variable FIFO order, the propagated entries should form a prefix of each buffer.

Formally, this requires the following modifications:

1. Write entries in buffers take the form $j{:}\texttt{W}(v)$ where $j \in \mathsf{BlockID} \cup \{\bot\}$ and $v \in \mathsf{Val}$ (instead of $\texttt{W}(v)$). A write entry of the form $\bot{:}\texttt{W}(v)$ means that the corresponding write was not a part of a persistence block.
2. States are extended to be quintuples $M = \langle \dot{m}, \tilde{m}, P, B, \mathit{Bid}\rangle$, where:
   - $B : \mathsf{Tid} \to \mathsf{NVVar} \to (\mathsf{BlockID} \cup \{\bot\})$ is called the *active-block mapping*. It assigns a block identifier (or $\bot$ if there is no active block) to every thread identifier and non-volatile variable.
   - $\mathit{Bid} \subseteq \mathsf{BlockID} \times \mathcal{P}(\mathsf{NVVar})$ is called the *block identifiers set*. It is used to store all persistence block identifiers occurring so far, each accompanied by the set of non-volatile variables that it protects.

   We denote by $M.\texttt{B}$ and $M.\texttt{Bid}$ the additional components of a state M. We impose the following well-formedness conditions:
   - If $j{:}\texttt{W}(\_) \in M.\texttt{P}(\dot{x})$, then $\langle j, \{\dot{x}\} \cup \dot{X}\rangle \in M.\texttt{Bid}$ for some $\dot{X} \subseteq \mathsf{NVVar}$.
   - If $M.\texttt{B}(\tau)(\dot{x}) \neq \bot$, then $\langle M.\texttt{B}(\tau)(\dot{x}), \{\dot{x}\} \cup \dot{X}\rangle \in M.\texttt{Bid}$ for some $\dot{X} \subseteq \mathsf{NVVar}$.
3. The initial state is given by $M_{\mathsf{Init}} \stackrel{\text{def}}{=} \langle \dot{m}_{\mathsf{Init}}, \tilde{m}_{\mathsf{Init}}, P_{\mathsf{Init}}, B_{\mathsf{Init}}, \mathit{Bid}_{\mathsf{Init}}\rangle$, where $B_{\mathsf{Init}} \stackrel{\text{def}}{=} \lambda\tau.\, \lambda\dot{x}.\, \bot$, and $\mathit{Bid}_{\mathsf{Init}} \stackrel{\text{def}}{=} \emptyset$.
4. The NV-WRITE transition records the current active block in the added entry:

$$\textsc{nv-write}\ \frac{l = \texttt{W}(\dot{x}, v) \qquad p' = M.\texttt{P}(\dot{x}) \cdot M.\texttt{B}(\tau)(\dot{x}){:}\texttt{W}(v) \qquad P' = M.\texttt{P}[\dot{x} \mapsto p']}{M \xrightarrow{\tau, l}_{\mathsf{PSC}} M[\texttt{P} \mapsto P']}$$

5. The following two transitions for opening and closing blocks are added:

$$\textsc{beginPB}\ \frac{\begin{array}{c} l = \mathsf{beginPB}(\dot{X}) \\ \forall \dot{x} \in \dot{X}.\, M.\texttt{B}(\tau)(\dot{x}) = \bot \\ B' = M.\texttt{B}\left[\tau \mapsto \lambda\dot{x}.\, \begin{array}{l}\text{if } \dot{x} \in \dot{X} \text{ then } j \\ \text{else } M.\texttt{B}(\tau)(\dot{x})\end{array}\right] \\ \langle j, \_\rangle \notin M.\texttt{Bid} \qquad \mathit{Bid}' = M.\texttt{Bid} \cup \{\langle j, \dot{X}\rangle\} \end{array}}{M \xrightarrow{\tau, l}_{\mathsf{PSC}} M[\texttt{B} \mapsto B', \texttt{Bid} \mapsto \mathit{Bid}']}$$

$$\textsc{endPB}\ \frac{\begin{array}{c} l = \mathsf{endPB}(\dot{X}) \\ B' = M.\texttt{B}\left[\tau \mapsto \lambda\dot{x}.\, \begin{array}{l}\text{if } \dot{x} \in \dot{X} \text{ then } \bot \\ \text{else } M.\texttt{B}(\tau)(\dot{x})\end{array}\right] \end{array}}{M \xrightarrow{\tau, l}_{\mathsf{PSC}} M[\texttt{B} \mapsto B']}$$

Thus, opening a block allocates a fresh identifier and sets the active-block mapping accordingly. In turn, closing a block resets the relevant variables in the active-block mapping.

6. The following transition is used *instead* of PERSIST-WRITE and PERSIST-FO. It generalizes both PERSIST-WRITE and PERSIST-FO by simultaneously persisting several entries together (each $p_{\dot{x}}$ below stands for a *sequence* of entries).

$$\text{PERSIST}\ \frac{\begin{array}{c} l = \texttt{per} \qquad \forall \dot{x}.\, M.\texttt{P}(\dot{x}) = p_{\dot{x}} \cdot P'(\dot{x}) \\ \forall j.\, (\exists \dot{x}.\, j{:}\texttt{W}(\_) \in p_{\dot{x}}) \implies \forall \dot{x}.\, (\forall \tau.\, M.\texttt{B}(\tau)(\dot{x}) \neq j \wedge j{:}\texttt{W}(\_) \notin P'(\dot{x})) \\ \dot{m}' = \lambda \dot{x}. \begin{cases} v & \text{last write entry in } p_{\dot{x}} \text{ has value } v \\ M.\dot{\texttt{m}}(\dot{x}) & \text{there are no write entries in } p_{\dot{x}} \end{cases} \end{array}}{M \xrightarrow{l}_{\mathsf{PSC}} M[\dot{\texttt{m}} \mapsto \dot{m}', \texttt{P} \mapsto P']}$$

This step imposes two restrictions. First, the persisted entries from each buffer ($p_{\dot{x}}$) should form a prefix of that buffer, so that FIFO semantics is maintained. Second, to respect the persistence blocks, if some entry of a given block is persisted ($\exists \dot{x}.\, j{:}\texttt{W}(\_) \in p_{\dot{x}}$) then that block should not be currently active by any thread ($\forall \dot{x}, \tau.\, M.\texttt{B}(\tau)(\dot{x}) \neq j$) and no entries of that block should remain in the volatile buffers ($\forall \dot{x}.\, j{:}\texttt{W}(\_) \notin P'(\dot{x})$)).

We note that nested and interleaved blocks are allowed. The program on the right demonstrates such a case. Here, $\dot{\texttt{x}} = 1$ and $\dot{\texttt{y}} = 1$ must persist together; $\dot{\texttt{z}} = 1$ and $\dot{\texttt{w}} = 1$ must persist together; but these two pairs can persist independently of each other in any order. Thus, provided that the client and the library use blocks of their own locations, the block instructions by each component are invisible to the other.

```
beginPB(ẋ, ẏ);
ẋ := 1;
beginPB(ż, ẇ);
ż := 1; ẇ := 1;
endPB(ż, ẇ);
ẏ := 1;
endPB(ẋ, ẏ);
```

4.3 Separation Properties

To enable our library abstraction proof, the required key property of PSC, which we preserved in its extensions, is the ability to separate PSC states into disjoint parts (the library's part and the client's part) and capture each memory transition in terms of its effect on the two parts. Next, we formulate this property, which we will later use to prove library abstraction. In fact, our arguments for library abstraction rely only on the properties below, and never "unfold" the PSC-related definitions. This allows one to refine and extend PSC, as long as the separation properties are preserved.

The separation of PSC states is stated in terms of the following restriction operator relative to a set of variables. For persistence blocks to behave correctly, we need an auxiliary condition on this set: we say that a set $\dot{X} \subseteq \mathsf{NVVar}$ *separates a state* $M \in \mathsf{PSC}.\texttt{Q}$ if for every $\langle j, \dot{Y} \rangle \in M.\texttt{Bid}$, we have $\dot{Y} \subseteq \dot{X}$ or $\dot{Y} \subseteq \mathsf{NVVar} \backslash \dot{X}$.

Definition 7. The *restriction* of $M \in \mathsf{PSC}.\texttt{Q}$ onto a set $X \subseteq \mathsf{Var}$ such that $X \cap \mathsf{NVVar}$ separates M, denoted by $M|_X$, is the state $M' \in \mathsf{PSC}.\texttt{Q}$ given by:

- $M'.\dot{\texttt{m}}(\dot{x})$ is $M.\dot{\texttt{m}}(\dot{x})$ if $\dot{x} \in \mathsf{NVVar} \cap X$, or 0 otherwise.
- $M'.\tilde{\texttt{m}}(\tilde{x})$ is $M.\tilde{\texttt{m}}(\tilde{x})$ if $\tilde{x} \in \mathsf{VVar} \cap X$, or 0 otherwise.

- $M'.\mathtt{P}(\dot{x})$ is $M.\mathtt{P}(\dot{x})$ if $\dot{x} \in \mathsf{NVVar} \cap X$, or ϵ otherwise.
- For each $\tau \in \mathsf{Tid}$, $M'.\mathtt{B}(\tau)(\dot{x})$ is $M.\mathtt{B}(\tau)(\dot{x})$ if $\dot{x} \in \mathsf{NVVar} \cap X$, or $\bot$ otherwise.
- $M'.\mathtt{Bid} = \{\langle j, \dot{Y}\rangle \in M.\mathtt{Bid} \mid \dot{Y} \subseteq X\}$.

The next lemma states the separation property of PSC, providing a precise characterization of each PSC transition in terms of transitions on the restrictions $M|_X$ and $M|_{\mathsf{Var}\setminus X}$. A special case is needed for store fence transitions, since taking these transitions enforces conditions on *both* restrictions.

Lemma 1. Let $X \subseteq \mathsf{Var}$ such that $X \cap \mathsf{NVVar}$ separates a state M_1.

1. For every $\tau \in \mathsf{Tid}$ and $l \in \mathsf{Lab} \setminus \{\mathtt{SF}\}$ with $\mathtt{varset}(l) \subseteq X$,
   $M_1 \xrightarrow{\tau,l}_{\mathsf{PSC}} M_2 \iff (M_1|_X \xrightarrow{\tau,l}_{\mathsf{PSC}} M_2|_X \wedge M_1|_{\mathsf{Var}\setminus X} = M_2|_{\mathsf{Var}\setminus X})$
2. For every $\tau \in \mathsf{Tid}$,
   $M_1 \xrightarrow{\tau,\mathtt{SF}}_{\mathsf{PSC}} M_2 \iff (M_1|_X \xrightarrow{\tau,\mathtt{SF}}_{\mathsf{PSC}} M_2|_X \wedge M_1|_{\mathsf{Var}\setminus X} \xrightarrow{\tau,\mathtt{SF}}_{\mathsf{PSC}} M_2|_{\mathsf{Var}\setminus X})$
3. $M_1 \xrightarrow{\mathtt{per}}_{\mathsf{PSC}} M_2 \iff (M_1|_X \xrightarrow{\mathtt{per}}_{\mathsf{PSC}} M_2|_X \wedge M_1|_{\mathsf{Var}\setminus X} \xrightarrow{\mathtt{per}}_{\mathsf{PSC}} M_2|_{\mathsf{Var}\setminus X})$
4. $M_1 \xrightarrow{\lightning}_{\mathsf{PSC}} M_2 \iff (M_1|_X \xrightarrow{\lightning}_{\mathsf{PSC}} M_2|_X \wedge M_1|_{\mathsf{Var}\setminus X} \xrightarrow{\lightning}_{\mathsf{PSC}} M_2|_{\mathsf{Var}\setminus X})$

The proof of Lemma 1 proceeds by standard case analysis ranging over all possible transitions of PSC. Finally, the following operation is used below to compose a state from a client and a library components (see Lemma 2).

Definition 8. Let M_1, M_2 be states of PSC, and $X_1, X_2 \subseteq \mathsf{Var}$ such that $X_1 \cap X_2 = \emptyset$. The *merge of* M_1 *and* M_2 *w.r.t.* X_1 *and* X_2, denoted by $\langle M_1, X_1\rangle \uplus \langle M_2, X_2\rangle$, is the state $M \in \mathsf{PSC}.\mathtt{Q}$ defined by:

$$M.\dot{\mathtt{m}}(\dot{x}) = \begin{cases} M_1.\dot{\mathtt{m}}(\dot{x}) & \dot{x} \in X_1 \\ M_2.\dot{\mathtt{m}}(\dot{x}) & \dot{x} \in X_2 \\ 0 & \text{otherwise} \end{cases} \quad \begin{array}{l}\text{similar definitions}\\ \text{for } M.\tilde{\mathtt{m}},\, M.\mathtt{P},\, M.\mathtt{B}\end{array} \quad M.\mathtt{Bid} = \begin{array}{l}\{\langle j, \dot{Y}\rangle \in M_1.\mathtt{Bid} \mid \dot{Y} \subseteq X_1\} \cup \\ \{\langle j, \dot{Y}\rangle \in M_2.\mathtt{Bid} \mid \dot{Y} \subseteq X_2\}\end{array}$$

5 Libraries and Their Clients

We present the notions of libraries and clients, as well as the necessary definitions for stating the abstraction theorem: histories and most general clients.

Libraries. We take a library L to be a function assigning to method names in $dom(L) \subseteq \mathsf{F}$ flat instruction sequences representing the method bodies. In the context of some library L, we refer to the implementations of the methods in $\{\mathsf{main}\} \cup \mathsf{F} \setminus dom(L)$ in a program Pr as the *client of* L.

Client-library composition. We consider the common case where libraries and their clients never access the same shared variables. To formally define this restriction, we use the following notations for sets of locations used by instruction sequences, libraries, and their clients:

- $\mathsf{Var}(I)$ denotes the set of shared variables mentioned in an instruction sequence I (possibly as a part of a set $\dot{X}$ of variables, e.g., in $\mathtt{beginPB}(\dot{X})$).
- For a library L, $\mathsf{Var}(L) \stackrel{\text{def}}{=} \bigcup_{f \in dom(L)} \mathsf{Var}(L(f))$.
- For a program Pr and a set $F \subseteq \mathsf{F}$,
  $\mathsf{Var}(Pr \setminus F) \stackrel{\text{def}}{=} \bigcup_{\tau \in \mathsf{Tid}} \mathsf{Var}(Pr(\tau)(\mathsf{main})) \cup \bigcup_{f \in \mathsf{F}\setminus F} \mathsf{Var}(Pr(f))$.

Then, client-library composition is defined as follows.

Definition 9. A library L is *safe* for a program Pr if $\mathsf{Var}(L) \cap \mathsf{Var}(Pr \setminus dom(L)) = \emptyset$. When L is safe for Pr, we write $Pr[L]$ for the program obtained from Pr by setting $Pr(\tau)(f) = L(f)$ for every $\tau \in \mathsf{Tid}$ and $f \in dom(L)$.

Note that we always have $\mathsf{Var}(Pr[L] \setminus dom(L)) = \mathsf{Var}(Pr \setminus dom(L))$.

Histories. Histories record the interactions between libraries and clients. Formally, a *history* h of a library L is a sequence of transition labels representing a crash, a call to a method of L, a return from a method of L, or an sfence, i.e., labels from the set $\mathsf{HTLab}_{dom(L)}$, which is defined as follows:

$$\mathsf{Lab}_F \stackrel{\text{def}}{=} \{\mathtt{SF}\} \cup \{\mathtt{CALL}(f,\phi), \mathtt{RET}(f,\phi) \mid f \in F, \phi : \mathsf{Reg} \to \mathsf{Val}\}$$
$$\mathsf{HTLab}_F \stackrel{\text{def}}{=} (\mathsf{Tid} \times \mathsf{Lab}_F) \cup \{\lightning\}$$

Definition 10. Let t be a trace of $Pr \bowtie \mathsf{PSC}$ for some program Pr. The *history* induced by t w.r.t. a set $F \subseteq \mathsf{F}$, denoted by $\mathsf{H}_F(t)$, is the subsequence of t over HTLab_F consisting of (in the same order they appear in t): call and return labels $\langle \tau, \mathtt{CALL}(f,\phi)\rangle$ and $\langle \tau, \mathtt{RET}(f,\phi)\rangle$ with $f \in F$; $\mathtt{SF}$-labels $\langle \tau, \mathtt{SF}\rangle$; and crash labels. The notation $\mathsf{H}_F(t)$ is extended to sets of traces in the obvious way. The set of histories w.r.t. F of Pr, denoted by $\mathsf{H}_F(Pr)$, is given by $\mathsf{H}_F(\mathsf{traces}(Pr \bowtie \mathsf{PSC}))$. When $F = \mathsf{F}$ (i.e., the set of all method names), we simply write $\mathsf{H}(t)$ and $\mathsf{H}(Pr)$.

Most general clients. We encompass library calling policies (see §2.3) using the notion of a "most general client"—a non-deterministic client that invokes the library methods in the most general way allowed by the policy. Formally, a most general client MGC is given as a (concurrent) program. Adherence to the calling policy is defined as follows.

Definition 11. Let L be a library, and Pr and MGC be programs such that L is safe for both Pr and MGC. We say that Pr *correctly calls* L w.r.t. MGC if $\mathsf{H}_{dom(L)}(Pr[L]) \subseteq \mathsf{H}_{dom(L)}(MGC[L])$.

The policy of a library with no restrictions on its clients (beyond the separation of shared resources) is expressed by an MGC, called MGC_{free}, that repeatedly invokes arbitrary library methods with arbitrary initial stores. Often persistent objects include a recovery method meant to be executed after a crash before any other method is invoked. We call such a policy MGC_{rec}. Formally, MGC_{free} (for $dom(L) = \{f_1, \ldots, f_n\}$) and MGC_{rec} (for $dom(L) = \{f_1, \ldots, f_n\} \uplus \{\mathsf{recover}\}$) assign the following main method to each thread τ:

$MGC_{\mathsf{free}}(\tau)(\mathsf{main}) =$
```
BEGIN : havoc;
goto f1 | ... | fn | END;
f1 : call(f1); goto BEGIN;
...
fn : call(fn); goto BEGIN;
END :
```

$MGC_{\mathsf{rec}}(\tau)(\mathsf{main}) =$
```
a := CAS(x̃, 0, 1); if a = 0 goto REC; goto WAIT;
REC : call(recover); ỹ := 1; goto BEGIN;
WAIT : a := ỹ; if a = 0 goto WAIT; goto BEGIN;
BEGIN : ... rest of the code as in MGC_free ...
```

In MGC_{rec}, using a compare-and-swap, one thread performs the recovery. All other threads wait until recovery ends to start their method invocations.

6 The Library Abstraction Theorem

In this section we state and prove the library abstraction theorem. The premise of this theorem, the *library correctness condition*, is formulated as follows.

Definition 12. Let L and $L^{\#}$ be libraries, both safe for a program MGC. We say that L *refines* $L^{\#}$ w.r.t. MGC, denoted by $L \sqsubseteq_{MGC} L^{\#}$, if both libraries implement the same methods and $\mathsf{H}(MGC[L]) \subseteq \mathsf{H}(MGC[L^{\#}])$.

Next, the abstraction theorem states that $L \sqsubseteq_{MGC} L^{\#}$ ensures that any client adhering to the library's calling policy may safely use the implementation L while reasoning about possible behaviors in terms of the specification $L^{\#}$. Our notion of "a behavior" includes the generated histories, as well as the reachable states, by the composition of the program and the memory system. Including reachable states is intended to assist safety verification. Clearly, we cannot require that the program states match for threads that are currently executing a method of L. In addition, since L and $L^{\#}$ may update the memory differently (e.g., use different variables), we should only consider the variables of the client when inspecting the memory states. This leads us to the following statement.

Theorem 1 (Abstraction). *Suppose that $L \sqsubseteq_{MGC} L^{\#}$. Let MGC and Pr be programs such that both L and $L^{\#}$ are safe for MGC and Pr, and Pr correctly calls $L^{\#}$ w.r.t. MGC. If $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$, then there exist $t^{\#}$ and $\langle \overline{q}^{\#}, M^{\#} \rangle$ such that the following hold:*

- $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t^{\#}}_{Pr[L^{\#}] \bowtie \mathsf{PSC}} \langle \overline{q}^{\#}, M^{\#} \rangle$.
- $\mathsf{H}(t^{\#}) = \mathsf{H}(t)$.
- *For every* $\tau \in \mathsf{Tid}$, *if* $\overline{q}(\tau).\mathtt{f} \notin dom(L)$, *then* $\overline{q}^{\#}(\tau) = \overline{q}(\tau)$.
- $M^{\#}|_{\mathsf{Var}(Pr \setminus dom(L))} = M|_{\mathsf{Var}(Pr \setminus dom(L))}$ *(see Def. 7).*

Note that $L \sqsubseteq_{MGC} L^{\#}$ is necessary for the conclusion to hold: otherwise, MGC itself is a client that can observe behaviors of L that are impossible for $L^{\#}$. Following §2.3, we also note that policy adherence is required w.r.t. to $L^{\#}$.

To prove the abstraction theorem, the following key lemma is used multiple times (with different arguments). It allows us to compose the client's part from one trace with the library's part from another into one combined trace.

Lemma 2 (Composition). Let L and L' be libraries implementing the same set F of methods such that both are safe for a program Pr, and L is also safe for a program Pr'. Suppose that $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t_{\mathsf{cl}}}_{Pr[L'] \bowtie \mathsf{PSC}} \langle \overline{q}_{\mathsf{cl}}, M_{\mathsf{cl}} \rangle$, $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t_{\mathsf{lib}}}_{Pr'[L] \bowtie \mathsf{PSC}} \langle \overline{q}_{\mathsf{lib}}, M_{\mathsf{lib}} \rangle$, and $\mathsf{H}_F(t_{\mathsf{cl}}) = \mathsf{H}_F(t_{\mathsf{lib}})$. Then, there exists a trace t such that $\mathsf{H}(t) = \mathsf{H}(t_{\mathsf{cl}})$ and $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t}_{Pr[L] \bowtie \mathsf{PSC}} \langle \overline{q}, M \rangle$, for:

- $\overline{q} = \lambda\tau. \begin{cases} \langle \overline{q}_{\mathsf{lib}}(\tau).\mathtt{pc}, \overline{q}_{\mathsf{lib}}(\tau).\phi, \overline{q}_{\mathsf{cl}}(\tau).\mathtt{pc_s}, \overline{q}_{\mathsf{cl}}(\tau).\mathtt{f} \rangle & \overline{q}_{\mathsf{cl}}(\tau).\mathtt{f} \in F \\ \overline{q}_{\mathsf{cl}}(\tau) & \text{otherwise} \end{cases}$
- $M = \langle M_{\mathsf{cl}}|_{\mathsf{Var}(Pr \setminus F)}, \mathsf{Var}(Pr \setminus F) \rangle \uplus \langle M_{\mathsf{lib}}|_{\mathsf{Var}(L)}, \mathsf{Var}(L) \rangle$ (see Def. 8).

The proof of Lemma 2 is based on the inherent disjointness in client-library composition provided by a library safe for its client program, which we leverage in the following two ways.

Firstly, we extract *client-local* and *library-local* transition properties from all transitions of $Pr[L'] \bowtie \mathsf{PSC}$ and $Pr'[L] \bowtie \mathsf{PSC}$. Thus, when we consider a transition by $Pr[L'] \bowtie \mathsf{PSC}$ corresponding to an instruction outside of a method of L', we show that an analogous transition is possible with the same program state, but with memory state zeroing out locations used by the library L'. Similarly, when we consider a transition by $Pr'[L] \bowtie \mathsf{PSC}$ corresponding to an instruction in a method of L, we show that an analogous transition is possible with almost the same program state, except we alter its stored program counter, and with memory state zeroing out locations used by the client Pr'. The justifications for these steps follow by the ($\Rightarrow$) directions of Lemma 1.

Secondly, we compose the *client-local* transition properties Pr exhibits in t_{cl} and the *library-local* transition properties L exhibits in t_{lib} while constructing transitions of $Pr[L] \bowtie \mathsf{PSC}$ for a trace t. Knowing that L is safe for Pr, we consider client-local transition properties from t_{cl} corresponding to transitions we wish to recreate in t, and replace zeroed-out memory locations with locations of L. Dually, we consider library-local transition properties from t_{lib} corresponding to transitions we wish to recreate in t, and replace zeroed-out memory locations with locations of Pr. The ($\Leftarrow$) directions of Lemma 1 justify such transformations. For instance, non-$\mathtt{SF}$-transitions can be composed, provided that the client program preserves the library memory state, and vice versa; while crashes and $\mathtt{SF}$-transitions record an interaction between a client program and a library and therefore need to be performed in synchrony.

We use these two ideas in proving Lemma 2 by induction on the sum of lengths of t_{cl} and t_{lib}, and use their local transition properties to justify composing them in synchrony. For the base case, we can simply take $t = \epsilon$. For the induction step, we consider the last labels in t_{cl} and t_{lib}, as well as the cases when one of the traces is empty. When $t_{\mathsf{cl}} = \_ \cdot \alpha_{\mathsf{cl}}$ and $t_{\mathsf{lib}} = \_ \cdot \alpha_{\mathsf{lib}}$, we use t' from the induction hypothesis for t_{cl} and t_{lib} with the last action removed from either or both of them, and let $t = t' \cdot \alpha_{\mathsf{cl}}$ or $t = t' \cdot \alpha_{\mathsf{lib}}$.

Then, the abstraction theorem is proved as follows.

Proof outline for Thm. 1. It suffices to show $\mathsf{H}(Pr[L]) \subseteq \mathsf{H}(Pr[L^\#])$; then the claim follows using Lemma 2 by letting $L := L^\#$, $L' := L$, $Pr := Pr$, and $Pr' := Pr$. Suppose otherwise, and let h be a shortest history in $\mathsf{H}(Pr[L]) \setminus \mathsf{H}(Pr[L^\#])$. Let t be a shortest trace in $\mathsf{traces}(Pr[L] \bowtie \mathsf{PSC})$ with $\mathsf{H}(t) = h$. Consider the last transition label α in t. The minimality of h and t ensures that α must be a return transition label for some $f \in dom(L)$. Indeed, otherwise, we can show that α is enabled in the end of a corresponding trace of $Pr[L^\#] \bowtie \mathsf{PSC}$, which contradicts the fact that $h \notin \mathsf{H}(Pr[L^\#])$. (The full argument here requires applying Lemma 2 with $L := L^\#$, $L' := L$, $Pr := Pr$, and $Pr' := Pr$.)

Now, using the fact that Pr correctly calls $L^\#$ w.r.t. MGC, we again apply Lemma 2 with $L := L$, $L' := L^\#$, $Pr := MGC$, and $Pr' := Pr$, and derive that α is enabled in the end of a corresponding trace of $MGC[L] \bowtie \mathsf{PSC}$. Then,

$L \sqsubseteq_{MGC} L^{\#}$ ensures that $\mathsf{H}_{dom(L)}(t) \in \mathsf{H}_{dom(L)}(MGC[L^{\#}])$. Using Lemma 2 for the last time (applied with $L := L^{\#}$, $L' := L$, $Pr := Pr$, and $Pr' := MGC$), we obtain that $h = \mathsf{H}(t) \in \mathsf{H}(Pr[L^{\#}])$, which contradicts our assumption. □

The following corollary of Thm. 1 states that, like classical linearizability, our correctness condition is compositional (a.k.a. local), meaning that a library consisting of several (non-interacting) libraries can be abstracted by considering each sub-library separately. Formally, the composition of libraries $L_1, \ldots, L_n$ with pairwise disjoint sets of declared methods, denoted by $L_1 \uplus \ldots \uplus L_n$, is defined to be the library obtained by taking the union of $L_1, \ldots, L_n$. Compositionality is formulated as follows.

Corollary 1 (Compositionality). The following two conditions together imply that $L_1 \uplus \ldots \uplus L_n \sqsubseteq_{MGC} L_1^{\#} \uplus \ldots \uplus L_n^{\#}$:

1. $\mathsf{Var}(L_1), \ldots, \mathsf{Var}(L_n), \mathsf{Var}(L_1^{\#}), \ldots, \mathsf{Var}(L_n^{\#}), \mathsf{Var}(MGC \setminus dom(L_1 \uplus \ldots \uplus L_n))$ are pairwise disjoint.
2. For all i, $L_i \sqsubseteq_{MGC_i} L_i^{\#}$ for $MGC_i = MGC[L_1^{\#} \uplus \ldots \uplus L_{i-1}^{\#} \uplus L_{i+1}^{\#} \uplus \ldots \uplus L_n^{\#}]$.

To end this section, we provide a simple lemma that allows one to establish $L \sqsubseteq_{MGC} L^{\#}$ by applying standard simulation arguments for *crashless* traces (with observable transitions being those that induce history labels). For that matter, we require a simulation relation on non-volatile memories generated by $MGC[L] \bowtie \mathsf{PSC}$ and $MGC[L^{\#}] \bowtie \mathsf{PSC}$ that holds for the very initial memory and preserved during crashless executions.

Lemma 3. A trace t is $\dot{m}_0$-*to*-$\dot{m}$ if $\langle \overline{q}_{\mathsf{Init}}, M_{\mathsf{Init}}[\dot{\mathtt{m}} \mapsto \dot{m}_0] \rangle \xrightarrow{t}_{Pr \bowtie \mathsf{PSC}} \langle \overline{q}, M[\dot{\mathtt{m}} \mapsto \dot{m}] \rangle$ for some $\overline{q}$ and M. Suppose that some relation R on $\mathsf{NVVar} \to \mathsf{Val}$ satisfies:

- $\langle \dot{m}_{\mathsf{Init}}, \dot{m}_{\mathsf{Init}} \rangle \in R$.
- If $\langle \dot{m}_0, \dot{m}_0^{\#} \rangle \in R$, then for every $\dot{m}_0$-to-$\dot{m}$ crashless trace t of $MGC[L] \bowtie \mathsf{PSC}$, there exist a non-volatile memory $\dot{m}^{\#}$ and an $\dot{m}_0^{\#}$-to-$\dot{m}^{\#}$ crashless trace $t^{\#}$ of $MGC[L^{\#}] \bowtie \mathsf{PSC}$, such that $\langle \dot{m}, \dot{m}^{\#} \rangle \in R$ and $\mathsf{H}(t) = \mathsf{H}(t^{\#})$.

Then, assuming $dom(L) = dom(L^{\#})$, we have that $L \sqsubseteq_{MGC} L^{\#}$.

Furthermore, if $MGC[L^{\#}]$ has no $\mathtt{fo}(\cdot)$ and `sfence` instructions, then $MGC[L^{\#}] \bowtie \mathsf{PSC}$ can take non-deterministic sfence steps (see §3) when $MGC[L] \bowtie \mathsf{PSC}$ takes `SF-` steps, so store fences can be ignored when checking $\mathsf{H}(t) = \mathsf{H}(t^{\#})$.

7 An Application: Persistent Pairs

We illustrate the use of the library abstraction theorem for a simple concurrent and persistent data structure—a pair of values that supports write and read operations. We present two specifications and an implementation for each specification. Both specifications ensure atomicity (i.e., linearizability if the system does not crash), and "data consistency" (reads return values written by a single write invocation), but they differ in their persistency guarantees. For the concurrency aspect, the implementations follow the sequence lock (seqlock, for short)

mechanism, which uses a version counter along with the pair and allows readers to avoid blocking [6]. For durability, the implementations employ different techniques: one uses a "redo log" and the other is based on "checkpoints".

A durable pair. The first specification, a library we denote by $L^{\#}_{\text{pair}}$, consists of three methods: write for writing the two values of the pair, read for reading the pair, and recover for recovering from a crash. The specification is as follows:[5]

```
write:
LOCK: if CAS(l̃, 0, 1)
   goto LOCK;
beginPB(ẋ1, ẋ2);
ẋ1 := a1; ẋ2 := a2;
endPB(ẋ1, ẋ2);
fl(ẋ1);
UNLOCK: l̃ := 0;
return;
```

```
read :
LOCK: if CAS(l̃, 0, 1)
   goto LOCK;
a1 := ẋ1; a2 := ẋ2;
UNLOCK: l̃ := 0;
return;

recover :
return;
```

A volatile lock ($\tilde{\mathtt{l}}$) is used to ensure atomicity. For durability, writes use persistence blocks, which ensure that the two parts of the pair persist simultaneously. After the block is ended, $\mathtt{fl}(\dot{\mathtt{x}}_1)$ (equivalent here to $\mathtt{fl}(\dot{\mathtt{x}}_2)$ due to the persistence block) ensures that the block persists. If the system crashes after a write completed, the written values are guaranteed to survive the crash. Thus, there is nothing to be done at recovery. Nevertheless, aiming to allow implementations, the library policy requires that recovery is executed after every crash before other methods are invoked (MGC_{rec} in §5).

Next, we present an implementation of $L^{\#}_{\text{pair}}$, which we denote by L_{pair}. We write $x := y$ instead of a read of y (to some fresh register) followed by a write to x. We also omit some necessary register bookkeeping: since histories record the whole register store in call/return labels, strictly speaking, implementations must unroll changes to registers not used to pass return values.

```
write:
LOCK: if CAS(l̃, 0, 1)
   goto LOCK;
ẋ1^new := a1; fo(ẋ1^new); ẋ2^new := a2; fo(ẋ2^new);
sfence;
ṡ := ṡ + 1; fl(ṡ);
ẋ1 := a1; fo(ẋ1); ẋ2 := a2; fo(ẋ2);
sfence;
ṡ := ṡ + 1;
UNLOCK: l̃ := 0;
return;
```

```
read :
BEGIN: a := ṡ;
if odd(a)
   goto BEGIN;
a1 := ẋ1; a2 := ẋ2;
if ṡ ≠ a
   goto BEGIN;
return;
```

```
recover :
if even(ṡ)
   goto END;
ẋ1 := ẋ1^new; fo(ẋ1);
ẋ2 := ẋ2^new; fo(ẋ2);
sfence;
END: ṡ := 0;
return;
```

Ignoring crashes, atomicity is guaranteed here using a seqlock. As for persistency, observe first that writing directly to the NVM is wrong since we cannot control the non-deterministic propagation: if a crash occurs during the execution of write, it is possible that only one part of the pair has persisted, and the recovery method will not have sufficient information for reinitializing the pair correctly. Instead, write first records its "job" in $\langle \dot{\mathtt{x}}_1^{\mathtt{new}}, \dot{\mathtt{x}}_2^{\mathtt{new}} \rangle$. Then, if a crash happens and

[5] Our simplified language has no mechanism for argument passing. We assume that write receives arguments (read returns results) via designated registers, $\mathtt{a}_1$ and $\mathtt{a}_2$.

the write was in the middle of updating $\langle \dot{x}_1, \dot{x}_2 \rangle$ (as identified via observing an odd version number), the recovery will complete the job of the writer. We note that the (rather extensive) use of flushes (or flush-optimals followed by an sfence) is necessary here in order to restrict the out-of-order persistence. The final write to $\dot{s}$ in write does not have to be explicitly persisted. Indeed, if a crash happens between this write and its persistence, recovery will redo the (idempotent) job.

Theorem 2. $L_{\text{pair}} \sqsubseteq_{MGC_{\text{rec}}} L^{\#}_{\text{pair}}$.

Our proof sketch uses Lemma 3, letting $\langle \dot{m}, \dot{m}^{\#} \rangle \in R$ if the following hold:

- If $\dot{m}(\dot{s})$ is even, then $\dot{m}(\dot{x}_1) = \dot{m}^{\#}(\dot{x}_1)$ and $\dot{m}(\dot{x}_2) = \dot{m}^{\#}(\dot{x}_2)$.
- If $\dot{m}(\dot{s})$ is odd, then $\dot{m}(\dot{x}_1^{\text{new}}) = \dot{m}^{\#}(\dot{x}_1)$ and $\dot{m}(\dot{x}_2^{\text{new}}) = \dot{m}^{\#}(\dot{x}_2)$.

Using the abstraction theorem, we obtain that for a program Pr that uses L_{pair} correctly (i.e., calls recovery first after every crash), for every state $\langle \overline{q}, M \rangle$ that is reachable in $Pr[L_{\text{pair}}] \bowtie \mathsf{PSC}$, there exists a state $\langle \overline{q}^{\#}, M^{\#} \rangle$ reachable in $Pr[L^{\#}_{\text{pair}}] \bowtie \mathsf{PSC}$ and indistinguishable from $\langle \overline{q}, M \rangle$ from the client perspective.

A buffered durable pair. A second specification, denoted by $L^{\#}_{\text{bpair}}$, allows for "buffered" behaviors, which enable faster implementations by weakening persistency guarantees [24]. Instead of requiring operations to persist before returning, it only requires that operations are "persistently ordered" before returning.

write:
```
LOCK: if CAS(l̃, 0, 1)
   goto LOCK;
beginPB(ẋ1, ẋ2);
ẋ1 := a1; ẋ2 := a2;
endPB(ẋ1, ẋ2);
UNLOCK: l̃ := 0;
return;
```

read:
```
LOCK: if CAS(l̃, 0, 1)
   goto LOCK;
a1 := ẋ1; a2 := ẋ2;
UNLOCK: l̃ := 0;
return;
```

recover:
```
return;
```

sync:
```
fl(ẋ1);
return;
```

Compared to $L^{\#}_{\text{pair}}$, the explicit flush instruction $\texttt{fl}(\dot{x}_1)$ from the write method is omitted, which means that a crash after a completed write may take the pair back to its state before the write. Thus, the state after a crash need not necessarily be fully up-to-date. An additional method, called sync, can used to ensure that previous writes have persisted. Without sync, an implementation could simply ignore persistency and store the pair in the volatile memory, which corresponds to an execution of $L^{\#}_{\text{bpair}}$ in which the persistency buffers are never being flushed.

An implementation can be obtained as follows:

write:
```
LOCK: if CAS(l̃, 0, 1)
   goto LOCK;
s̃ := s̃ + 1;
x̃1 := a1; x̃2 := a2;
s̃ := s̃ + 1;
UNLOCK: l̃ := 0;
return;
```

read:
```
BEGIN: a := s̃;
if odd(a) goto BEGIN;
a1 := x̃1; a2 := x̃2;
if s̃ ≠ a goto BEGIN;
return;
```

recover:
```
if ḟ = 1 goto PREV;
x̃1 := ẋ1^next; x̃2 := ẋ2^next;
return;
PREV: x̃1 := ẋ1^prev; x̃2 := ẋ2^prev;
ḟ := 0; fl(ḟ);
return;
```

sync:
```
LOCK: if CAS(l̃, 0, 1)
   goto LOCK;
a1 := x̃1; a2 := x̃2;
ẋ1^prev := ẋ1^next; fo(ẋ1^prev);
ẋ2^prev := ẋ2^next; fo(ẋ2^prev);
sfence;
ḟ := 1; fl(ḟ);
NEXT: ẋ1^next := a1; fo(ẋ1^next);
ẋ2^next := a2; fo(ẋ2^next);
sfence;
ḟ := 0; fl(ḟ);
UNLOCK: l̃ := 0;
return;
```

This implementation exploits the freedom allowed by the specification. Writes and reads again employ a seqlock, but this time they only use volatile variables. In turn, sync sets a "checkpoint", and recovery rolls the state back to the latest complete checkpoint. For that matter, a non-volatile flag $\dot{\mathtt{f}}$ is used to detect crashes during the setting the checkpoint $\langle \dot{\mathtt{x}}_1^{\mathtt{next}}, \dot{\mathtt{x}}_2^{\mathtt{next}} \rangle$. Thus, before storing the checkpoint, the previous checkpoint is stored in the non-volatile variables $\langle \dot{\mathtt{x}}_1^{\mathtt{prev}}, \dot{\mathtt{x}}_2^{\mathtt{prev}} \rangle$. Upon recovery, given the value of the flag, we know if we can restore the state from the current stored checkpoint, or, if a crash happened during the store of this checkpoint (which means that sync did not return), set the pair to the previous stored one.

Theorem 3. $L_{\text{bpair}} \sqsubseteq_{MGC_{\text{rec}}} L^{\#}_{\text{bpair}}$.

Our proof sketch uses Lemma 3, letting $\langle \dot{m}, \dot{m}^{\#} \rangle \in R$ if the following hold:

- If $\dot{m}(\dot{\mathtt{f}}) = 0$, then $\dot{m}(\dot{\mathtt{x}}_1^{\mathtt{next}}) = \dot{m}^{\#}(\dot{\mathtt{x}}_1)$ and $\dot{m}(\dot{\mathtt{x}}_2^{\mathtt{next}}) = \dot{m}^{\#}(\dot{\mathtt{x}}_2)$.
- If $\dot{m}(\dot{\mathtt{f}}) = 1$, then $\dot{m}(\dot{\mathtt{x}}_1^{\mathtt{prev}}) = \dot{m}^{\#}(\dot{\mathtt{x}}_1)$ and $\dot{m}(\dot{\mathtt{x}}_2^{\mathtt{prev}}) = \dot{m}^{\#}(\dot{\mathtt{x}}_2)$.

8 Related and Future Work

Library abstraction theorems. Previous work has developed library abstraction theorems for crashless shared memory concurrency. First, [13] formalized the intuition that standard linearizability as defined in [21] corresponds to contextual refinement (and also proved a completeness result: the converse also holds provided that threads have other means of interaction besides the library). Later, [7] refined and formulated this result using history inclusion instead of linearizability, which is closer to our formalization. Other abstraction results account for liveness [16], resource-transferring programs [17], and x86-TSO [8]. Our composition lemma (Lemma 2) is inspired by [8], which addresses a challenge that is close to the challenge posed by store fence instructions in NVM, where actions of the client and the library affect each other even if they access to distinct locations. To do so, the notion of a history is extended to expose events that correspond to the flushing certain entries from the x86-TSO store buffers, which is close to what we do to handle store fences. Our alternative approach to this problem, i.e., introducing a relaxed version of the store fence, is novel.

While our framework is operational, library abstraction was also studied before for declarative shared memory concurrency semantics, particularly in the context of the C11 weak memory model [5,28].

Linearizability notions for persistent objects. Different approaches for adapting the standard linearizability criterion that is based on crash-free sequential specifications [21] were proposed before [3,19,24], but were not formally related to contextual refinement. Since methods like recover and sync (see §7) are meaningless in crash-free sequential specifications, they require an ad-hoc external treatment in these linearizability adaptations. The variety of approaches to interpret crash-free sequential specifications for crash-resilient concurrent objects

makes it hard, in particular, to combine libraries with different linearizability guarantees in a single program.

In turn, these existing notions are typically expressible in the refinement framework that we employ. For example, in the *crashless* setting, by wrapping each method of a sequential implementation S of some object inside a global lock, one obtains an abstract library $L^{\#}_{S}$ for that object that corresponds to the conditions imposed by standard linearizability [7] (a library L is linearizable w.r.t. S iff every crashless history induced by a trace of $MGC[L]$ is also induced by some trace of $MGC[L^{\#}_{S}]$). Now, when crashes are involved, by wrapping each method of S inside a global lock *and a persistence block* followed by an explicit flush instruction (like $L^{\#}_{\text{pair}}$ in §7), one obtains an abstract library $L^{\#}_{S\lightning}$ that corresponds to the conditions imposed by strict linearizability of [3] (L is strictly linearizable w.r.t. S iff $L \sqsubseteq_{MGC} L^{\#}_{S\lightning}$). Thus, our results can be used to derive contextual refinement (using $L^{\#}_{S\lightning}$ as a specification) from strictly linearizable objects. We note that while the original definition of strict linearizability was for a model with per-processor failure, what we consider here is its application for full system crashes.

Durable linearizability [24] weakens strict linearizability by allowing methods that were active during a crash to take their effect at any later point in the execution (or never), instead of requiring that the effect of such methods is visible immediately after the crash (or never). This weakening aims to allow lazy recovery for large structures, where either the recovery procedure is executed in parallel to other methods after a crash, or the methods themselves participate in recovering the data structure when they are further executed. This notion can be also expressible as an abstract implementation in our language. For this matter, every update method in the specification would: first record its task in a work-set; remove the task from the work-set; flush the updated work-set; and perform the task like in $L^{\#}_{S\lightning}$ described above. In turn, every query method may choose to complete any task it finds in the work-set, since the method performing such a task has crashed during its invocation. For persistent pairs (see §7), this is illustrated by the specification below. The non-volatile variable $\dot{w}$ is the multiset holding the work-set with atomic add and remove operations, and $\tilde{\mathtt{l}}_{\mathtt{rw}}$ is an abstract multiple-readers-single-writer lock used to resolve races on the work-set.

write:	read:
`LOCK1:` acquire $\tilde{\mathtt{l}}_{\mathtt{rw}}$ as a reader;	`goto` {`LOCK1`, `BEGIN`};
add $\langle \mathtt{a}_1, \mathtt{a}_2 \rangle$ to $\dot{w}$;	`LOCK1:` acquire $\tilde{\mathtt{l}}_{\mathtt{rw}}$ as a writer;
remove $\langle \mathtt{a}_1, \mathtt{a}_2 \rangle$ from $\dot{w}$;	pick some $\langle \mathtt{a}_1, \mathtt{a}_2 \rangle \in \dot{w}$;
$\mathtt{fl}(\dot{w})$;	remove $\langle \mathtt{a}_1, \mathtt{a}_2 \rangle$ from $\dot{w}$;
`UNLOCK1:` release $\tilde{\mathtt{l}}_{\mathtt{rw}}$;	$\mathtt{fl}(\dot{w})$;
... continue as in write of $L^{\#}_{\text{pair}}$ (§7) ...	... write $\langle \mathtt{a}_1, \mathtt{a}_2 \rangle$ to $\langle \mathtt{x}, \mathtt{y} \rangle$ as in write of $L^{\#}_{\text{pair}}$ (§7) ...
recover :	`UNLOCK1:` release $\tilde{\mathtt{l}}_{\mathtt{rw}}$;
`return`;	`BEGIN:` ... continue as in read of $L^{\#}_{\text{pair}}$ (§7) ...

A "buffered" version of strict linearizability, which only requires the existence of a prefix of the completed invocations to be observed after a crash, is also naturally derived by considering $L^{\#}_{S\lightning \mathrm{b}}$ which is obtained from a sequential implementation S by wrapping each method of S inside a global lock and a per-

sistence block (*without* an explicit flush instruction) and ensuring that there is a single non-volatile variable that is written to by all library methods (introducing such a variable if needed).[6]

An alternative operational characterization of durable linearizability using Input/Output automata was developed in [12] and used to formally establish this property for the persistent queue of [14] by providing a full-blown simulation proof using the KIV proof assistant.[7] Nevertheless, this work does not relate the proved correctness criterion to contextual refinement.

Persistency models. The underlying model we assume is PSC by [25], a strengthening of Px86 [30] that formalizes the Intel-x86 persistency. The paper [25] provided compiler mappings that ensure PSC semantics on machines guaranteeing Px86 semantics. We extended the general semantic framework with libraries, and extended PSC with local store fences and persistence blocks.

Future work. Future work includes extending our proof method and results for weaker persistency models, such as persistent x86-TSO [30] and ARM [10]; handling random access shared memory with allocations and deallocations (instead of the simplified shared variables model we employ); and lifting the strict condition that libraries and clients live in disjoint address spaces by allowing them to transfer ownership of certain locations (as was done in [17] for standard volatile memory).

In addition, extending and adapting methods for refinement verification under volatile memory is needed in order to provide library developers with means to validate our library-correctness conditions. Such methods may include automated checking by approximation [7], layered interactive verification in the style of [20,27], and formal logics as the one in [26]. Similarly, developing formal methods and tools that allow using library specifications for client reasoning is left for future work, including decidable reachability analysis [2], program logics [29], and principled testing [15]. Finally, it is interesting to see how logical atomicity notions established by program logics, such as [11,31], which has been extended to cover crashes in disk-based storage systems [9], can be adapted for establishing our correctness condition and/or for client reasoning.

[6] Since the corresponding "buffered" correctness notion is not compositional, while the refinement-based notion is (see Corollary 1), one cannot expect to have a per-object translation of a sequential implementation S into a concurrent and persistent implementation $L^{\#}_{S\,\lightning\,\mathrm{b}}$. Indeed, the addition of a single non-volatile variable that is written to by all library methods is a not a per-object translation (i.e., for two sequential library implementations implementing disjoint sets of methods and operating on disjoint variables, S_1 and S_2, we will *not* have $L^{\#}_{S_1 \cup S_2\,\lightning\,\mathrm{b}} = L^{\#}_{S_1\,\lightning\,\mathrm{b}} \cup L^{\#}_{S_2\,\lightning\,\mathrm{b}}$).

[7] See https://kiv.isse.de/projects/Durable-Queue.html.

References

1. C++ reference (std::list::pop_front explanation), https://www.cplusplus.com/reference/list/list/pop_front/ [Accessed Jan-2022]
2. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: TACAS. pp. 324–338. Springer (2013)
3. Aguilera, M.K., Frølund, S.: Strict linearizability and the power of aborting. Technical Report HPL-2003-241 (2003)
4. ARM: ARM architecture reference manual: ARMv8, for ARMv8-A architecture profile (2021), https://developer.arm.com/documentation/ddi0487/latest/ [Accessed July-2021]
5. Batty, M., Dodds, M., Gotsman, A.: Library abstraction for C/C++ concurrency. In: POPL. pp. 235–248. ACM, New York, NY, USA (2013)
6. Boehm, H.J.: Can Seqlocks get along with programming language memory models? In: MSPC. pp. 12–20. ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2247684.2247688
7. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable refinement checking for concurrent objects. In: POPL. p. 651–662. ACM, New York, NY, USA (2015), https://doi.org/10.1145/2676726.2677002
8. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent library correctness on the TSO memory model. In: ESOP. pp. 87–107. Springer, Berlin, Heidelberg (2012)
9. Chajed, T., Tassarotti, J., Theng, M., Jung, R., Kaashoek, M.F., Zeldovich, N.: Gojournal: a verified, concurrent, crash-safe journaling system. In: OSDI. pp. 423–439. USENIX Association (Jul 2021), https://www.usenix.org/conference/osdi21/presentation/chajed
10. Cho, K., Lee, S.H., Raad, A., Kang, J.: Revamping hardware persistency models: View-based and axiomatic persistency models for Intel-x86 and Armv8. In: PLDI. p. 16–31. ACM, New York, NY, USA (2021), https://doi.org/10.1145/3453483.3454027
11. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A logic for time and data abstraction. In: ECOOP. pp. 207–231. Springer (Jul 2014), https://doi.org/10.1007/978-3-662-44202-9_9
12. Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Wehrheim, H.: Verifying correctness of persistent concurrent data structures: a sound and complete method. Formal Aspects of Computing pp. 1–27 (2021)
13. Filipović, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theoretical Computer Science **411**(51), 4379–4398 (2010), https://www.sciencedirect.com/science/article/pii/S0304397510005001
14. Friedman, M., Herlihy, M., Marathe, V., Petrank, E.: A persistent lock-free queue for non-volatile memory. In: PPoPP. pp. 28–40. ACM, New York, NY, USA (2018), http://doi.acm.org/10.1145/3178487.3178490
15. Gorjiara, H., Xu, G.H., Demsky, B.: Jaaru: Efficiently model checking persistent memory programs. In: ASPLOS. p. 415–428. ACM, New York, NY, USA (2021), https://doi.org/10.1145/3445814.3446735
16. Gotsman, A., Yang, H.: Liveness-preserving atomicity abstraction. In: ICALP. pp. 453–465. Springer, Berlin, Heidelberg (2011)
17. Gotsman, A., Yang, H.: Linearizability with Ownership Transfer. Logical Methods in Computer Science **Volume 9, Issue 3** (Sep 2013), https://lmcs.episciences.org/931

18. Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X.N., Weng, S.C., Zhang, H., Guo, Y.: Deep specifications and certified abstraction layers. In: POPL. p. 595–608. ACM, New York, NY, USA (2015), https://doi.org/10.1145/2676726.2676975
19. Guerraoui, R., Levy, R.R.: Robust emulations of shared memory in a crash-recovery model. In: ICDCS. p. 400–407. IEEE Computer Society, USA (2004)
20. Hawblitzel, C., Petrank, E., Qadeer, S., Tasiran, S.: Automated and modular refinement reasoning for concurrent programs. In: CAV. pp. 449–465. Springer, Cham (2015)
21. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (Jul 1990), http://doi.acm.org/10.1145/78969.78972
22. Intel: Persistent Memory Programming (2015), http://pmem.io/
23. Intel: Intel 64 and ia-32 architectures software developer's manual (combined volumes) (May 2019), https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, order Number: 325462-069US
24. Izraelevitz, J., Mendes, H., Scott, M.L.: Linearizability of persistent memory objects under a full-system-crash failure model. In: DISC. pp. 313–327. Springer, Berlin, Heidelberg (2016)
25. Khyzha, A., Lahav, O.: Taming x86-TSO persistency. Proc. ACM Program. Lang. **5**(POPL), 47:1–47:29 (Jan 2021), https://doi.org/10.1145/3434328
26. Liang, H., Feng, X., Fu, M.: Rely-guarantee-based simulation for compositional verification of concurrent program transformations. ACM Trans. Program. Lang. Syst. **36**(1) (Mar 2014), https://doi.org/10.1145/2576235
27. Lorch, J.R., Chen, Y., Kapritsos, M., Parno, B., Qadeer, S., Sharma, U., Wilcox, J.R., Zhao, X.: Armada: Low-effort verification of high-performance concurrent programs. In: PLDI. p. 197–210. ACM, New York, NY, USA (2020), https://doi.org/10.1145/3385412.3385971
28. 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. Proc. ACM Program. Lang. **3**(POPL), 68:1–68:31 (Jan 2019), http://doi.acm.org/10.1145/3290381
29. 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
30. Raad, A., Wickerson, J., Neiger, G., Vafeiadis, V.: Persistency semantics of the Intel-x86 architecture. Proc. ACM Program. Lang. **4**(POPL) (Jan 2020), https://doi.org/10.1145/3371079
31. Svendsen, K., Birkedal, L., Parkinson, M.: Modular reasoning about separation of concurrent data structures. In: ECOOP. pp. 169–188. Springer, Berlin, Heidelberg (2013)
32. Zuriel, Y., Friedman, M., Sheffi, G., Cohen, N., Petrank, E.: Efficient lock-free durable sets. Proc. ACM Program. Lang. **3**(OOPSLA), 128:1–128:26 (Oct 2019), http://doi.acm.org/10.1145/3360554